Ref. Ares(2021)4269474 - 30/06/2021

Rational decomposition and orchestration for serverless computing

Deliverable D2.5

RADON Adoption Handbook

Version: 1.0

Publication Date: 30-June-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	D2.5
Title:	RADON Adoption Handbook
Editor(s):	D. Di Nucci, D. A. Tamburri (TJD)
Contributor(s):	E. Anastasiadis, G. Casale, Z. Niu, J. Perusquia Cortes, S. Tuli, R. Wang, L. Zhu (IMP), S. Dalla Palma, D. Di Nucci (TJD), P. Jakovits (UTR), M. Cankar, S. Dragan, A. Luzar, S. Stanovnik (XLB), G. Giotis (ATC), S. D'Agostini (ENG), T. F. Düllmann, A. van Hoorn, M. Wurster, V. Yussupov (UST), A. I. Spartalis (EFI)
Reviewers:	M. Cankar (XLB), G. Triantafyllou (ATC)
Type:	R
Version:	1.0
Date:	30-June-2021
Status:	Final
Dissemination level:	Confidential
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
EFI	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

This deliverable provides details concerning Task T2.4 and all its activities within the scope of the RADON action. More specifically, the document offers principles and practices for the adoption of RADON tools or its integrated environment and rotating around the following: (a) the perimeter of applicability of each tool and intended usage scenarios, (b) guidelines to customize RADON while retaining its core benefits; (c) cultural shift guidelines needed to adopt DevOps practices, serverless and microservice architecture within the target organization as well as barriers to adoption and customization; (d) anti-patterns emerging when implementing socio-technical processes in large-scale DevOps projects. Finally, the document contains an outline of a cross-project adoption exercise conducted jointly with the EU SODALITE project and regulated by a joint Memorandum of Understanding (MoU) between the two consortia.

Table of contents

1. Introduction 6
1.1 Deliverable objectives 6
1.2 Overview of main achievements 6
1.3 Structure of the document 7
2. RADON Framework Assumptions and Known Limitations 8
2.1 RADON IDE and CI/CD Plugin 8
2.2 Function Hub 9
2.3 Continuous Testing Tool 9
2.4 Data Pipeline Plugin 10
2.5 Defect Prediction Tool 11
2.6 Graphical Modeling Tool 12
2.7 Orchestrator 13
2.8 Decomposition Tool 14
2.9 Constraint Definition Language & Verification Tool 14
2.10 RADON Particles & Template Library 15
2.10.1 RADON Particles 15
2.10.2 Template Library 15
2.11 Monitoring Tool 16
3. RADON Framework Customization 18
3.1 Customize to run local xOpera instance 18
3.2 Customize to run other CI/CD 19
3.3 Customize to other IDEs 22
3.4 Customize IDE deployment (locally vs in the cloud) 23
3.5 Customize to other target clouds and platform services for applications 25
3.6 Customize to other modelling languages or other version of TOSCA 29
3.7 Customize defect prediction to other IaC languages 30
3.8 Customize to other orchestrators based on TOSCA or not 34
3.9 Customize data pipelines to support additional data sources 35
3.10 Customize to other trigger relationships/events 36
3.11 Customize to other Monitoring 38
3.12 Customize to other test types and tools 39
4. Adopting the RADON DevOps Process: Principles and Practices 44
4.1 Context of study 44
4.2 From RUP to CI/CD: adopted practices and emerging challenges 45

4.3 Technical and organizational adoption fallacies 46
4.4 Lessons learned and future work 47

5. DevOps Community (Anti-)Patterns on the Road to RADON: Process and Product (Re-)Design Principles 48
5.1 Software community structure types and smells explained 48
5.2 Experimental setup 49
5.3 Results of the experiments 53
5.4 RADON adoption principles 56

6. Adopting RADON for Hybrid Computing: the Joint RADON-SODALITE Use Case Report 57
6.1 Weather News front-end case study 57
6.1.1 HPC layer 58
6.1.2 Cloud and edge layers 59
6.2 Tool baselines 60
6.3 Technical achievements 62
6.3.1 Hybrid compute profile 62
6.3.2 GridFTP support 63
6.3.3 Support for edge deployment on Raspberry Pi devices 64
6.4 Open-source release 65

7. Conclusions 66

8. References 67

Appendix A: Hybrid Compute Profile 69
A.1 Artifact Types 69
A.2 Capability Types 69
A.3 Data Types 69
A.4 Node Types 70
A.5 Policy Types 73
A.6 Relationship Types 73
A.7 Service Templates 74

1. Introduction

1.1 Deliverable objectives

In this deliverable, we describe the inherent potential of the framework with respect to customization to embrace RADON within scenarios we did not originally foresee during the definition of the RADON action or the design/evaluation of its components. We consider this aspect essential for the framework to become a sustainable solution; a prerequisite is a clear roadmap for future development and adaptation processes. On the one hand, all units involved in RADON as tool owners have specific future work, and development plans fleshed out in the remainder of the document. On the other hand, tool- or scenario-specific adoption processes are tightly linked with flexible customization in which the framework becomes available and relevant for end-users with varying starting points, technical baseline, and ambitions. Central to this work would be to:

- Encourage and ease of adoption for RADON or any of its individual sub-components;
- Pinpoint any barriers for adoption and possibly offer mitigations;
- Define the necessary attributes of a company likely to adopt RADON, offering ways in which such company profile can operationally embrace the RADON organizational and technical culture;
- Define cultural shift strategies and technological transformations for the mentioned items.

1.2 Overview of main achievements

The main objective of the deliverable is to provide an adoption handbook for RADON; in particular, the deliverable provides:

- Information concerning the devops phase in which each tool is executed, the standards their leverages, the supported and/or assumed technologies, the known limitations that could prevent adopting them.
- From the premise of the IDE as a RADON baseline, which enables a flexible structure by default, we provide a list of possible customization and extensions that could be implemented to foster RADON adoption.
- An empirical ethnographic study to properly understand the process of adopting a DevOps-intensive technology, such as the RADON IDE that entails a DevOps-heavy software lifecycle.
- An empirical experiment on open-source projects closely related to the microservices and Function-as-a-Service (FaaS) design paradigm to verify the extent to which known software community anti-patterns---a.k.a., *community types and community smells*---affect the RADON adoption.

- A joint report, mutually developed by the RADON and SODALITE[1] Horizon 2020 projects that engaged a customization and mutual asset adoption experience. This provides a concrete example that demonstrates the challenges and solutions found in adopting RADON assets in situations that fell outside the scope of the project requirements.

1.3 Structure of the document

The remainder of this document is organized as follows. Section 2 presents the requirements and limitations of each RADON tool. Section 3 describes possible extensions and customizations RADON users could implement to suit RADON for their needs. Section 4 sheds light on the issues that an organization could observe when adopting a DevOps-intensive technology, such as the RADON IDE. Section 5 investigates the extent to which software community anti-patterns could harm RADON adoption. Section 6 describes the RADON extension realized in collaboration with SODALITE-H2020. Finally, Section 7 concludes the document.

[1] https://www.sodalite.eu/

2. RADON Framework Assumptions and Known Limitations

2.1 RADON IDE and CI/CD Plugin

Phase in devops cycle The RADON IDE supports several phases of the DevOps application lifecycle thanks to the available development environment and the integration of the RADON tools. The Application Development phase (i.e., planning, coding, and building) is supported by the following capabilities provided by the RADON IDE:

- Standard functionalities to support development activities (e.g., the code can be written in several programming languages, and it can be maintained by using version control tools, such as Git and SVN).
- Integration of several design-time RADON tools (i.e., GMT, VT, DT, DPT, CTT)

The Continuous Integration and Continuous Deployment phases are supported thanks to the availability of CI/CD functionalities. The CI/CD components integrated into the RADON IDE (i.e., CI/CD plugin and xOpera SaaS Orchestrator) permit to trigger CI/CD pipelines on a continuous integration platform and deploy the application blueprints to the production servers. In addition, the Monitoring phase of the DevOps lifecycle is supported by integrating the Monitoring tool in the RADON IDE.

Leveraged standards, data formats, and their versions. The integrated development environment manages the standards and data formats supported by the integrated RADON tools.

Supported/assumed technologies. The RADON IDE is based on the Eclipse Che platform, and these are the assumed technologies:

- Docker v20;
- Kubernetes infrastructure (i.e., Minikube);
- Eclipse Theia web-based IDE;
- Pre-built stacks for several programming languages (e.g., Java, Python, NodeJs);
- Visual Studio Code extensions;
- Keycloak as the Identity and Access Management system;
- Jenkins[2] as the CI/CD platform (i.e., the CI/CD plugin integrated into the IDE is configured to trigger a "job" on a Jenkins platform). The CI/CD preconditions are: (i) a configured Jenkins server; (ii) a user with execution access to jobs; (iii) a configured CI/CD pipeline.

A Che instance has been installed on a Centos 7 virtual machine (VM) hosted by FIWARE[3] and another one has been installed on Microsoft Azure public cloud.

[2] https://www.jenkins.io/
[3] https://www.fiware.org/

Known limitations. The Che instances have been deployed on the Kubernetes container infrastructure, but it is also possible to run and manage Che on an OpenShift cluster (e.g., using Minishift). By default, Che installation includes the deployment of a dedicated Keycloak instance (i.e., the Identity and Access Management system). However, using an external Keycloak is also possible. This option is useful when an existing Keycloak instance is available (e.g., a company-wide Keycloak server used by several applications). It is also possible to configure Che to work without Keycloak but only in single-user mode. The current version of the RADON IDE provides a CI/CD (Eclipse Che) plugin to trigger CI/CD pipelines on a Jenkins platform. Still, a user can customize the integrated development environment to use other CI/CD technology (e.g., CircleCI).

2.2 Function Hub

Phase in devops cycle. The Function Hub is part of the Application Development workflow (i.e., planning and coding).

Leveraged standards, data formats, and their versions.

- The user can select local files and necessary dependencies in the zip format.

Supported/assumed technologies. Function Hub client is used for creating, packaging, and deploying. Each Function is defined through the package format where the name, handler, cloud provider, and runtime are entered. The Function object is language agnostic. The Function Hub object can be passed as a source for alternative tools like the Serverless Framework.

Known limitations. In its current version, Function Hub supports storage and maintenance of artifacts sized up to 10 MB. We decided on this limit when designing our application's architecture as we want our product to be offered free of charge and open for contributions in the open-source community. As the application is hosted in Eficode's cloud premises, a cost is coming along with its use. Limiting the artifact size to 10MB still gives the user full access to our services and secures us from a potential overcharge originating from big files that naturally occupy large storage space and computing sources. Currently, Function Hub has its authentication system. In future versions, KeyCloak can be integrated and aligned with xOpera and the IDE. In addition, the collection of reusable functions could be enlarged. Finally, the package format could be improved to integrate additional fields (e.g., environment variables).

2.3 Continuous Testing Tool

Phase in devops cycle. The Continuous Testing Tool is part of the Test workflow (i.e., test).

Leveraged standards, data formats, and their versions.

- TOSCA v1.3
- TOSCA CSAR
- YAML/JSON

Supported/assumed technologies.

- CTT-defined RADON TOSCA types in Particles. CTT has been designed to be extensible with respect to new and customized test types and infrastructures.
- Existing test tools, e.g., JMeter. By design, the set of test tools can be extended due to a modular architecture. This allows developers to add their own tool-specific agents and integrate them into CTT via modules.
- Docker for IDE plugin, CTT server and agents. CTT can be run without Docker, while Docker allows an easily portable and self-contained deployment.
- Python 3 for developing CTT extensions and customization. Python is used as the implementation language for the CTT server and all currently provided modules and agents. In general, it is possible to implement agents in any programming language as long as it is properly integrated with the respective module.
- OpenAPI/REST for tool integration. It is the technology used for integrating CTT with its clients (CLI/IDE) and communication with the provided test agents. In general, it is possible to use other integrations for communicating with custom test agents as long as it is aligned with the respective modules in the CTT server.

Known limitations. Tight coupling to TOSCA standard and ecosystem (e.g., orchestrator) for test definition and deployment. Using another standard rather than TOSCA would imply changes to several core components in CTT, comprising test definition, deployment, and already provided testing tool support.

2.4 Data Pipeline Plugin

Phase in devops cycle. The Data Pipeline Plugin is part of the Application Development workflow (i.e., planning and coding).

Leveraged standards, data formats, and their versions.

- TOSCA v1.3
- TOSCA CSAR
- XML for Nifi- based pipeline templates
- YAML/JSON

Supported/assumed technologies.

- Ansible v2.0 automation engine.
- JSON for Amazon data pipeline base pipeline template.
- Apache Nifi v1.9.2 or later (latest recommended).
- Python v3.7 or later to update the JSON template while deploying Amazon data pipeline and using the data pipeline plugin as standalone.

- Docker v20.

Known limitations. The current version of the data pipeline supports open-source Apache Nifi data management and commercial AWS data pipeline. However, users can not deploy the data flow using other commercial data management solutions such as Google's Cloud Compose and Dataflow and Microsoft's Azure Data Factory solutions.

The current set of external data storage locations from/to where the users can consume/publish data includes Azure blob storage, Google storage bucket, AWS S3 bucket, SFTP, MQTT, and local directory. However, the users can not work with popular databases such as MySQL, MariaDB, MongoDB, Redis, and PostgreSQL. Further, while the current data pipeline does support strategic data flow, where the data should move based on user-specified conditions, it is somewhat limited, enabling many filter-like conditions based on data object attributes (mime type, file size, file name) and filtering based on JSON content (e.g., key "temperature" value is larger than 70).

2.5 Defect Prediction Tool

Phase in devops cycle. The Defect Prediction Tool is part of the Defect Prediction workflow (i.e., planning and coding).

Leveraged standards, data formats, and their versions.

- Ansible at least v2.0
- TOSCA v1.3
- TOSCA CSAR
- YAML/JSON

Supported/assumed technologies.

- Ansible at least v2.0
- Docker v20
- Docker Compose v3
- Python v3.6
- Firebase Cloud Storage

Known limitations. The tool currently supports only two IaC languages (i.e., Ansible and TOSCA) and five kinds of defects. However, more languages and defects could be easily integrated in the future. The defect prediction tool identifies defects at the file level. Future implementation will target lower levels of granularities (e.g., feature level, line level).

Furthermore, because the tool relies on Firebase, the end-user must stick to the Firebase billing plans. The current version is suitable for the main usage scenario and uses the Firebase Spark plan, which offers generous limits for getting started with Firebase. Therefore, it does not require the user

to subscribe to any billing plan. If the end-user wants to increase the current quota and storage limits, she can subscribe to a paid-tier plan[4] without affecting the tool's functioning.

2.6 Graphical Modeling Tool

Phase in DevOps cycle. The Graphical Modeling Tool is part of the Application Development workflow (i.e., planning, coding, and building).

Leveraged standards, data formats, and their versions.

- TOSCA v1.3
- TOSCA CSAR

Supported/assumed technologies. GMT is published as a Docker container. The image contains the two web applications for TOSCA entity management and topology modeling and the RESTful HTTP API. The provided container image can be used with any container orchestration technology, such as Kubernetes or Docker Swarm, or other container toolings such as Docker Compose. A Dockerfile and the respective Kubernetes configuration files are by default provided.

GMT's Docker container expects that the Template Library is mounted as a directory. This assumes that either the RADON Particles (RADON's public version of the Template Library) GitHub repository is cloned locally or the content from the Template Publishing Service is exported to a directory and respectively mounted into the container.

Further, GMT assumes a certain directory and file structure, which was introduced in deliverable D4.4 RADON Models II. For example, the RADON Particles follow this structure and can be directly used together with the GMT.

GMT is developed using Java (backend) and Angular (frontend). It is designed and implemented loosely coupled with users' actual contents needed to model their TOSCA application deployments. The content is provided by the Template Library containing different TOSCA entities, such as node types, policy types, requirement types, and even service templates. As a result, users may extend the content of the Template Library to introduce new types and entities for modeling. For this purpose, users may also use the GMT and its Entity Management UI to create and define new TOSCA types. Afterward, the created content can be pushed to the RADON Template Publishing Service or even pushed to the RADON Particles on GitHub by creating a respective fork and a pull request suggesting the changes.

Known limitations. Currently, GMT can only be operated based on a file-based data repository. This means that either the RADON Particles need to be cloned locally or a respective directory needs to be provided following the expected directory and file structure (full details in D4.4 RADON Models II).

[4] https://firebase.google.com/pricing

As of now, GMT only supports exporting a TOSCA service template as a CSAR file. According to the standard, the created archive contains a TOSCA-Metadata directory and a TOSCA.meta file describing the remaining entries and content of the archive. The TOSCA.meta is the entry point for a TOSCA orchestrator.

2.7 Orchestrator

Phase in DevOps cycle. The Orchestrator is part of the deployment phase.

The xOpera TOSCA orchestrator is used in the deployment (delivery and release) phase. Upon receiving IaC definitions from, e.g., the template library and/or a modeling tool, the orchestrator can execute deployment and subsequent day 2 actions, such as responding to monitoring events, project updates and teardown.

Using xOpera SaaS, users can define callback endpoints for TOSCA triggers, which any monitoring software can call to trigger custom actions. Information gleaned from this can be used in the Plan phase in the devops cycle.

Leveraged standards, data formats, and their versions.

- JSON
- TOSCA v1.3
- TOSCA CSAR

Supported/assumed technologies. The core xOpera orchestrator is written in Python and exposed as a CLI tool that can be installed as a pip package from Python Package Index (PyPI). The same goes for xOpera API. On the other hand xOpera SaaS component is more complex and its backend (including SaaS REST API, where OpenAPI specification is used) is written in Go, whereas the GUI was created with PatternFly framework. All SaaS services are launched inside Docker containers including Keycloak, which is the IAM and is used for SaaS authentication. SaaS also uses Linux containers (LXC) for spawning new projects and workspaces. Another xOpera component is the RADON IDE (Eclipse Che) plugin, which is written in TypeScript and is distributed as a Visual Studio Code extension.

Known limitations. Due to the nature of IaC deployment, the orchestrator must, at some point, be able to access user deployment credentials, e.g., cloud services. This is unavoidable as credentials are mandatory to run any deployment script, and using a tool to facilitate deployment must permit access for a nonzero time. Steps have been taken to minimize exposure, and tools have been implemented to enable users to quickly and efficiently manage credentials and projects' access to them.

The only executor currently implemented in the xOpera orchestrator is Ansible. However, this is not a technical limitation, and other execution backends can be added.

2.8 Decomposition Tool

Phase in devops cycle. The Decomposition Tool is part of the Decomposition workflow (i.e., operating).

Leveraged standards, data formats, and their versions

- TOSCA v1.3
- YAML/JSON

Supported/assumed technologies. The Decomposition Tool is developed based on layered queueing networks (LQNs), a performance formalism suitable to model most modern distributed systems. A dedicated set of data types and an abstract layer of node and relationship types are defined in the RADON Particles (see D3.3), allowing the use of LQN annotations to specify the interface and performance of an application. The Decomposition Tool has been deployed as a public service with a RESTful API[5] and can thus be invoked either from the RADON IDE or through a direct API call.

Known limitations. To simplify architecture decomposition, the Decomposition Tool ignores specific technologies in use and works on abstract RADON models, which are not deployable. As for deployment optimization, the Decomposition Tool only supports a limited range of AWS services at present. Further extension of this feature to other AWS services and cloud platforms is desirable. Extra assumptions are also made in the optimization program due to the inability of LQNs to capture the exact behavior of certain node types, for example, cold start and retrial of Lambda functions.

2.9 Constraint Definition Language & Verification Tool

Phase in devops cycle. The Verification Tool is part of the Verification workflow (i.e., planning and coding).

Leveraged standards, data formats, and their versions.

- TOSCA v1.3
- YAML/JSON

Supported/assumed technologies. The Verification Tool requires ILASP (commercial). All three modes require the clingo solver for answer set programming, which is open source and free to use.

The clingo solver can be replaced with other ASP solvers, requiring additional engineering/customization work. These include, for example, WASP[6] or DLV[7].

[5] https://github.com/radon-h2020/radon-decomposition-tool
[6] http://alviano.github.io/wasp/
[7] http://www.dlvsystem.com/dlv/

Replacing ILASP is not currently possible as no other ILP system supports learning full ASP programs.

Known limitations. The Constraint Definition Language language is Turing complete, so there is great flexibility in the language itself. However, several minor restrictions affecting the language have been outlined in deliverable D4.2, such as the requirement for arithmetic expressions to be bounded in a finite range. Although these restrictions do limit to some extent the use of the language, they have not proved to be significantly restrictive in the context of RADON application design.

2.10 RADON Particles & Template Library

To make the discussion more precise, we describe the RADON Particles and Template Publishing Service as separate subsubsections.

2.10.1 RADON Particles

Phase in DevOps cycle. The RADON Particles is part of the Application Development workflow (i.e., planning and coding).

Leveraged standards, data formats, and their versions.

- TOSCA v1.3
- TOSCA CSAR
- Ansible
- YAML
- Markdown

Supported/assumed technologies. The RADON Particles repository is hosted on GitHub; hence, Git is used as the underlying version control system (VCS). Furthermore, the repository is organized based on the requirements from RADON GMT, ensuring that it can be imported into GMT without any modifications needed. Since all service- and technology-specific information is stored in the root folder of the RADON Particles, switching the VCS or a hosting service requires minimal effort.

Known limitations. The RADON Particles relies on TOSCA, meaning that models in other languages, while still can be stored, are not of great use for tools using this repository. In addition, the specific structure required by the GMT might require additional integration effort when using other graphical modeling tools. Furthermore, currently, the deployment logic can only be implemented in Ansible due to the requirements of the RADON Orchestrator.

2.10.2 Template Library

Phase in devops cycle. The Template Library is part of the Application Development workflow (i.e., planning and coding).

Leveraged standards, data formats, and their versions.

- TOSCA v1.3
- TOSCA CSAR
- Ansible
- YAML
- Markdown

Supported/assumed technologies. The Template Library REST API is developed using Kotlin, endpoints are in camel case, for the database we use Postgres of version 12.2. TPS GUI uses PatternFly for the frontend and calls REST API using OpenAPI generated with JAR CLI generator using openapi specification YML file. CLI package is written in Python and Che TPS plugin in TypeScript and published as VSIX.

Known limitations. In addition to the limitations stated in 2.10.1, we have set some new limitations. Those are limits for the maximum size of templates set at 50MB and for readme file set at 1MB, limits of length for template names and descriptions, and regex for version name and template name. The actions supported by the Che plugin cover downloading and uploading templates and their versions, setting endpoint, logging in, and logging out. Actions for template and user groups are not supported. In the CLI package, the majority of actions are supported. In addition to actions supported in the Che plugin, users can manage and view templates and user groups and view their user information. TPS GUI is meant to search through Template Library's content and list and display templates by name, type, or template groups. Users can also download different versions and view readme files if present. All other actions are not enabled and are currently not planned.

2.11 Monitoring Tool

Phase in devops cycle. The Monitoring Tool is part of the monitoring workflow (i.e., monitoring).

Leveraged standards, data formats, and their versions.

- JSON
- TOSCA v1.3
- TOSCA CSAR

Supported/assumed technologies. The Monitoring Tool orchestrates Prometheus Server, Prometheus PushGateway, and Grafana along with its accompanying API. An accompanying core Javascript-based API is implemented using those underlying technologies to coordinate the monitored data aggregation and their respective visualizations through Grafana dashboards. Additionally, the API facilitates an Alert generation system based on a REST full API. Through this API and in combination with xOpera SaaS, the Monitoring tool offers the ability of dynamic resource scaling operations.

Known limitations. The Monitoring Tool offers great flexibility. The tool supports the monitoring and scaling of numerous types of resources (e.g., serverless, VM) and offers a great degree of streamlining/adjusting the Monitoring/Alerting/Scaling process resulting in a highly tailored solution. One of the minor limitations is that the monitoring system can be set up only on top of the particles library and has to follow the TOSCA specifications to be part of the automatic deployment through xOpera.

3. RADON Framework Customization

In this section, we provide a list of possible customizations of the RADON framework. Although not holistic, the list provides an overview of the possible extensions users could integrate before adopting RADON.

3.1 Customize to run local xOpera instance

xOpera is very versatile in the way it is set up. By default, xOpera SaaS is used as the managed multi-tenant instance of the xOpera orchestrator. To use xOpera without using xOpera SaaS, two options are available, explored in the next sections. As integration with the Template Library is done in the xOpera SaaS browser application, users will need to access it directly via its own (existing) API. Changing the xOpera SaaS to CLI or another version implies only a change in how the orchestrator is initiated to deploy the TOSCA CSAR file; therefore, the changes are quite simple, if the user is interested only in deploying.

Using the xOpera orchestrator CLI (opera). When the application design is finalised and CSAR created, the user can decide to download this CSAR and deploy the application manually. Any process currently using xOpera SaaS can use the CLI orchestrator by installing it from PyPI[8]. All operations are identical and accessible via the opera command line application. To customize the application, calls need to be rewritten into their CLI counterparts, which is a 1:1 mapping. The deployment can be done by running the `opera deploy` command. More details, how to use xOpera CLI is available on the documentation pages[9].

Using the xOpera orchestrator API (opera-api). The xOpera orchestrator API is a local HTTP API component for a single instance of the xOpera orchestrator. Installation is also done through PyPI[10] or in a clean Docker container environment, where the user can use the prepared Dockerfile[11] from the xopera-api GitHub repository to build a Docker image for the container. API call mapping is simple, as calls can be directly mapped to local endpoints, which are completely identical - the only difference is in the server prefix, omitting workspaces, project and user details. After installing the xOpera API Python package locally, the user can run the API in production mode with `python3 -m opera.api.cli` and the api will start listening on `localhost:8080`. Then the CSAR can be deployed by invoking the `localhost:8080/deploy` API endpoint with POST request containing the path to the TOSCA service template as JSON object. The API will mimic the xOpera CLI behaviour and will work in the directory where the curl command has been executed from (the .opera storage folder will be created in that directory), which means that before the

[8] https://pypi.org/project/opera/
[9] https://xlab-si.github.io/xopera-docs/
[10] https://pypi.org/project/opera-api/
[11] https://github.com/xlab-si/xopera-api/blob/master/Dockerfile

deployment the user needs to know the exact location of the CSAR to specify its relative path to the folder where the API endpoints are called with curl.

```
(.venv) $curl -XPOST localhost:8080/deploy -H "Content-Type: application/json"
-d '{ "service_template": "service.yaml"}'
{
  "clean_state": false,
  "id": "d89e2862-1068-4079-8bf2-1e3e100f63ec",
  "operation": "deploy",
  "service_template": "service.yaml",
  "state": "pending",
  "timestamp": "2021-06-15T13:28:51.485545+00:00"
}
```

Listing 3.1.1: A curl command that invokes the deployment of a service template with xOpera API.

3.2 Customize to run other CI/CD

The RADON IDE supports CI/CD functionalities so that a user can configure and trigger CI/CD pipelines through the IDE. The use of CI/CD pipelines provides more flexibility, for instance it is possible to include different tollgates in the deployment process (i.e., the job pipelines can be complex as you want).

In the current release of the RADON integrated framework, a CI/CD (che) plugin is offered in the IDE and it is used to invoke a CI/CD pipeline on a remote Jenkins platform. Therefore, it is expected the availability of a Jenkins server, an account with execution access to the jobs and a configured CI/CD pipeline as preconditions. Then, a configuration file (i.e., a yaml file), generated by the user within the RADON workspace, will be used to specify a set of properties needed to trigger the CI/CD pipelines. In our context, this configuration file specifies (i) the name and the version of the Cloud Service Achieve (CSAR) involved in the deployed process, (ii) the information of the Jenkins server and the account's credentials (i.e., Jenkins URL, username and password) and (iii) the information of the job to trigger (i.e., the URL of the job and the authentication token).

However, a user can customize the integrated development environment to use other CI/CD technologies (e.g. CircleCI). To provide CI/CD functionalities using a platform different from Jenkins a user could develop a new chePlugin to interact with the choosing CI/CD technology. The chePlugin(s) add capabilities to the Che-Theia IDE[12]. In particular, they permit extending the Eclipse Che GUI with new commands to execute functionalities or to interact with tools.

[12] https://github.com/eclipse-che/che-theia

To achieve the integration of a different CI/CD technology on the IDE the user should perform the following activities:

- Choose a CI/CD platform which permits to remotely trigger pipelines with parameters;
- Create an authentication token to grant authorization to trigger the job (the definition depends on the CI/CD platform);
- Define (in the chosen CI/CI platform) the CI/CD pipeline(s) to trigger from remote using the authentication token and the needed parameters;
- Implement and integrate a *chePlugin* which permits to trigger the pipeline(s) with specified parameters using the CURL command.

More details on how to implement and integrate a *chePlugin* to interact with the CI/CD platform and how to customize the CI/CD pipelines are reported below.

Adding a new chePlugin to interact with the CI/CD platform. The *chePlugin* can be implemented using a Visual Studio Code (VSC) extension, a kind of extension supported by the Eclipse Che technology. A Visual Studio Code[13] editor can be used to develop the extension, while the Node.js Package Manager[14] npm is needed to package the extension. The extension can be created following the instructions provided in the Visual studio Code documentation[15]. Figure 3.2.1 depicts the structure inside the project folder resulting after the creation of the extension.

```
.
├── .vscode
│   ├── launch.json     // Config for launching and debugging the extension
│   └── tasks.json      // Config for build task that compiles TypeScript
├── .gitignore          // Ignore build output and node_modules
├── README.md           // Readable description of your extension's functionality
├── src
│   └── extension.ts    // Extension source code
├── package.json        // Extension manifest
├── tsconfig.json       // TypeScript configuration
```

Figure 3.2.1: Directory structure of VSC extensions.

The file "src/extension.ts" is the entry point to implement the extension, while the file "package.json" is used to specify the metadata about the plug-in and the commands to execute, as well as the needed dependencies (for more information you can see the documentation provided in the "*demo-radon-plugins*" GitHub repository[16]). In order to trigger the pipelines remotely, the code of the extension must invoke the execution of the CURL command on which are specified the following data: the pipeline to trigger on the CI/CD platform, the authorization token, the

[13] https://code.visualstudio.com/
[14] https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/
[15] https://code.visualstudio.com/api/get-started/your-first-extension
[16] https://github.com/radon-h2020/demo-radon-plugins/blob/master/che/plugins/vsc-extensions-in-che.md

credentials of the user with execution access, the parameters taken as input by the pipeline. Listing 3.2.1 provides an example of the invocation for a job defined in Jenkins instance, while Listing 3.2.2 provides an example of the invocation for a job defined in a CircleCI instance.

```
$curl -u username:password
http://127.0.0.1:8080/job/Trigger_Remote_Demo/buildWithParameters?token=My-tok
en&para1=val1&para2=val2
```

Listing 3.2.1: CURL command to invoke remotely a Jenkins pipeline

```
curl  -u ${CIRCLECI_TOKEN}: -X POST --header "Content-Type: application/json"
-d '{
  "parameters": {
    "par1": "val1",
    "par2": "val2"
  }
}' https://circleci.com/api/v2/project/:project_slug/pipeline
```

Listing 3.2.2: CURL command to invoke remotely a CircleCI pipeline

Once the new *chePlugin* has been implemented, it can be packaged using the "vsce package" command as described in the Visual Studio Code documentation[17]. A *meta.yaml* file will be used to describe the plugin and to integrate it on the IDE (as described in the documentation provided in the "*demo-radon-plugins*" GitHub repository).

Customization of the CI/CD pipelines. Different CI/CD platforms require different configurations in order to execute the desired pipelines. RADON tool execution can be customized for different CI/CD platforms following the platform's specifications as most tools support CLI execution through PyPI packages or are publicly available as docker containers. An example of such customization of the RADON tools resides in the official Github repository[18] of CI/CD templates where all available pipeline templates initially created for the Jenkins platform were converted to suitable pipeline templates for CircleCI.

In order to parametrize the templates to a different platform a user should go through the official documentation of each tool that can be found in the RTD page. A summary of all the RTD documentations can be found in the RADON-methodology[19]. There, dedicated sections for CLI execution can be found and used as a basis for the template construction. Necessary configurations and modifications are always subject to each particular platform.

[17] https://code.visualstudio.com/api/working-with-extensions/publishing-extension#vsce
[18] https://github.com/radon-h2020/radon-cicd-templates
[19] https://github.com/radon-h2020/radon-methodology

3.3 Customize to other IDEs

Generally, the TOSCA extension developed in RADON can also be used within other IDEs, as they are rooted in the TOSCA 1.3 standard. This adoption challenge has been explored in the context of the RADON-SODALITE collaboration. The SODALITE IDE uses a TOSCA-like model definition language supported b yan ontology based semantic model which is translated to TOSCA blueprints as an intermediate execution language to provision resources, deploy and configure applications in heterogeneous environments. Limited effort was required to adapt the current tool to use RADON models, thanks to the aforementioned alignment to the TOSCA standard. This proof of concept suggests that integrating RADON TOSCA with other IDEs does not present significant obstacles. Figure 3.3.1, taken from the RADON-SODALITE joint report, gives a sense of the look-and-feel of the resulting SODALITE IDE with RADON TOSCA extensions for scaling triggers.

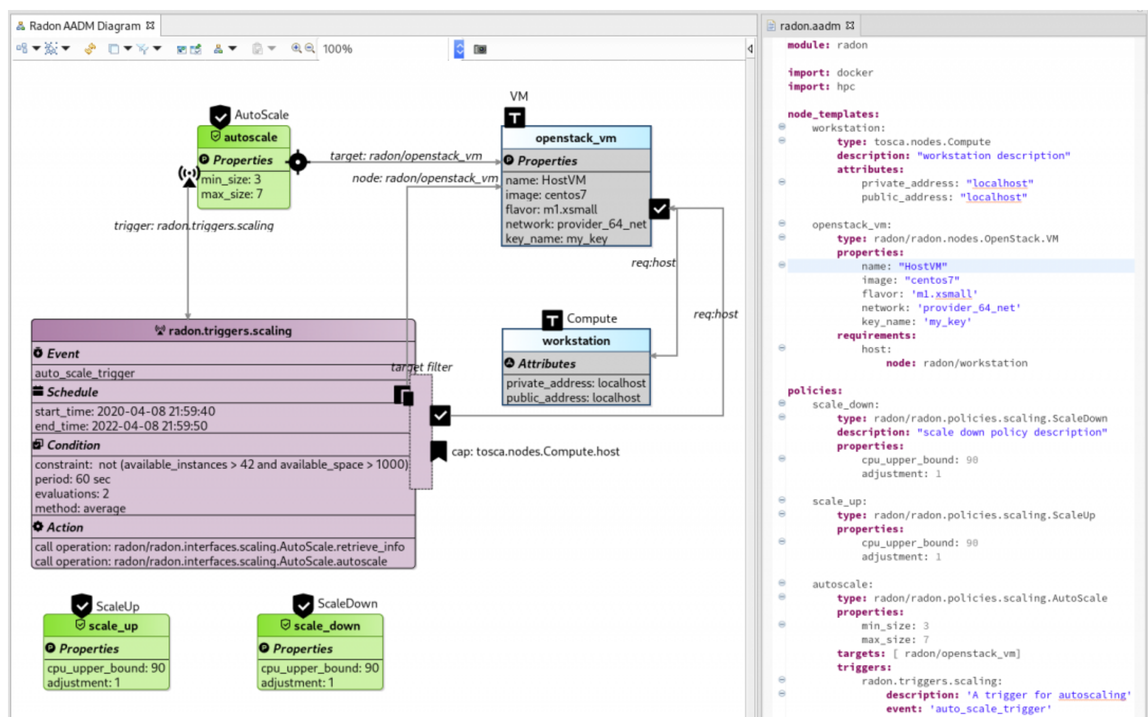

Figure 3.3.1: SODALITE IDE showing a sample RADON defined node type

In addition to integrating the RADON TOSCA within another IDE, another adoption need could be to integrate the tool plugins with a different IDE environment. Because the RADON IDE plugins are all stand-alone containers running on Kubernetes, there is a good degree of flexibility in interfacing these with other IDEs: provided that a basic plugin can issue REST calls to the container or open a new web browser tab, it would be a small effort to develop new IDE-specific plugins. Even in older environments such as the standard Eclipse, it is not difficult to issue commands to a container running on the same machine. Overall, extending RADON to other IDEs does not seem to present considerable challenges.

3.4 Customize IDE deployment (locally vs in the cloud)

The RADON IDE has been realized on top of the Eclipse Che[20] technology, a web-based development environment with multi-user support, where developers can create applications without the need to install any software on their local system. Eclipse Che is a Kubernetes[21]-native IDE, available in two modes: (i) *single-user* mode (non-authenticated Che), and (ii) *multi-user* mode (authenticated Che). The former is a lighter option most suited for personal desktop usage, whereas the latter option is present to support parallel development within a team or organization. Eclipse Che can be deployed on a single node or cloud platform that provides Kubernetes or OpenShift[22] infrastructure. More details about how to customize the local and cloud deployments of Eclipse Che are presented below.

Customize IDE deployment on a single node. During the project, an instance of Eclipse Che in multi-user mode was deployed on a Centos 7 virtual machine (VM) using Minikube[23] to set up the Kubernetes cluster. This deployment requires some minimal resources (i.e., CPUs, free memory and free disk space) on the host machine and the availability of a container manager (such as Docker[24]) as described in the Minikube documentation. Installations of the Kubernetes and Eclipse Che command-line tools[25, 26] (i.e., the "kubectl" and "chectl" tools) are also needed.

Once Minikube has been installed on the host machine, it can be started using the command reported on Listing 3.4.1 (At least 4GB of RAM should be allocated).

```
$ minikube start --memory=4096 --vm-driver=none
```

Listing 3.4.1: Command to start Minikube

Minikube runs a single-node Kubernetes cluster where the Eclipse Che instance is deployed. Listing 3.4.2 reports the command used to deploy the Che instance.

```
chectl server:deploy --platform=minikube --installer=operator --multiuser
--domain=<host-ip>.nip.io
```

Listing 3.4.2: Command to deploy Eclipse Che on Minikube

The user accounts and security policies of the Eclipse Che instance are managed by Keycloak, the identity and access management tool that authenticates users and secures interactions with external tools according to the specific *Client* configuration.

[20] https://www.eclipse.org/che/
[21] https://kubernetes.io/
[22] https://www.openshift.com/
[23] https://minikube.sigs.k8s.io/docs/
[24] https://www.docker.com/
[25] https://kubernetes.io/docs/tasks/tools/#kubectl/
[26] https://www.eclipse.org/che/docs/che-7/installation-guide/using-the-chectl-management-tool/

It is possible to deploy Eclipse Che on Kubernetes through other tools, for example MicroK8s[27], Docker Desktop[28] or kind[29]. Alternatively, a Che instance may be deployed on CodeReady Containers using OpenShift. Procedures for these local installation options are described in the Eclipse Che documentation[30].

Customize IDE deployment on a cloud platform. In the meanwhile, we have made another instance of Eclipse Che available on an Azure Kubernetes cluster for demonstrating RADON tools in outreach activities, such as workshops and meetups. This deployment requires an Azure account with an active subscription and an installation of the Azure command-line tool[31] (i.e. the "az" tool). The "kubectl" and "chectl" tools are also needed to deploy and manage the Kubernetes cluster and the Che instance respectively.

To work with a resource on the Azure platform, one must register the corresponding resource provider for their Azure subscription. Commands reported in Listing 3.4.3 can be used to register *Microsoft.ContainerService* and *Microsoft.Storage* resource providers needed by the Eclipse Che instance. Note that login to Azure from the command line is essential before executing these commands.

```
$ az provider register --namespace Microsoft.ContainerService
$ az provider register --namespace Microsoft.Storage
```

Listing 3.4.3: Commands to register the Azure resource providers

The next step is to create an Azure resource group at a specific location and the Azure Kubernetes cluster where the Eclipse Che instance is deployed. This can be done using commands reported in Listing 3.4.4. Particularly, the second command allocates 6 *Standard_DS2_v2* Azure VMs to the node pool of the Kubernetes cluster, which aims at hosting up to 32 concurrent RADON workspaces.

```
$ az group create --name eclipseCheResourceGroup --location <location-name>
$ az aks create --resource-group eclipseCheResourceGroup --name eclipse-che
--node-count 6 --generate-ssh-keys
$ az aks get-credentials --name eclipse-che --resource-group
eclipseCheResourceGroup
```

Listing 3.4.4: Commands to create the Azure resource group and the Azure Kubernetes cluster

An ingress controller is then needed to expose HTTP/HTTPS routes from outside the Azure Kubernetes cluster to services within the cluster. In this case, we choose the nginx[32] ingress controller, which is supported and maintained by Kubernetes. Listing 3.4.5 reports commands used

[27] https://microk8s.io/
[28] https://www.docker.com/products/docker-desktop
[29] https://github.com/kubernetes-sigs/kind
[30] https://www.eclipse.org/che/docs/che-7/installation-guide/installing-che-locally/
[31] https://docs.microsoft.com/en-us/cli/azure/
[32] https://nginx.org/

to install the nginx ingress controller in the Kubernetes cluster and show the external IP allocated by the Azure platform.

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.41.0/deploy/static/provider/cloud/deploy.yaml
$ kubectl get pods --namespace ingress-nginx
$ kubectl get services --namespace ingress-nginx
```

Listing 3.4.5: Commands to install the nginx ingress controller and show the allocated external IP

Finally, the Eclipse Che instance can be started using the command reported in Listing 3.4.6. We configure the Che instance through a custom resource patch file (.yaml), whose contents are reported in Listing 3.4.7. Please refer to the Eclipse Che documentation[33,34] for the format of the custom resource patch file and the details of the specified configuration options.

```
$ chectl server:start --platform=k8s --multiuser --domain=<external-ip>.nip.io
--che-operator-cr-patch-yaml=<path-to-cr-patch-yaml>
```

Listing 3.4.6: Command to start the Eclipse Che instance

```
apiVersion: org.eclipse.che/v1
kind: CheCluster
metadata:
  name: eclipse-che
spec:
  server:
    tlsSupport: false
    devfileRegistryImage: radonconsortium/che-devfile-registry:7.19.2
    customCheProperties:
      CHE_INFRA_KUBERNETES_PVC_STRATEGY: per-workspace
      CHE_INFRA_KUBERNETES_PVC_QUANTITY: 2Gi
      CHE_LIMITS_WORKSPACE_IDLE_TIMEOUT: "43200000"
      CHE_LIMITS_USER_WORKSPACES_COUNT: "1"
```

Listing 3.4.7: Contents of the custom resource patch file

Eclipse Che may be deployed on other public cloud platforms, such as AWS and Google Cloud. Alternatively, it is possible to deploy a Che instance in a private cloud using OpenShift or Kubespray. The Eclipse Che documentation[35] includes procedures for these cloud installation options.

3.5 Customize to other target clouds and platform services for applications

During the project, the consortium published a variety of TOSCA definitions to support deployment to AWS, Azure, GCP, and OpenFaaS. In particular, we support event-based application deployments involving FaaS as processing components and object storage services, such as AWS

[33] https://www.eclipse.org/che/docs/che-7/installation-guide/configuring-the-che-installation/
[34] https://www.eclipse.org/che/docs/che-7/installation-guide/advanced-configuration/
[35] https://www.eclipse.org/che/docs/che-7/installation-guide/installing-che-in-cloud/

S3, and DBaaS offerings, such as AWS DynamoDB, as respective event sources. The outcome, i.e., the required TOSCA node types, relationship types, policy types as well as example TOSCA service templates, has been published in the public Template Library so-called RADON Particles[36].

In future, the published TOSCA definitions will serve as a baseline for custom extensions. In general, the set of target clouds and platform services can be extended by developing and publishing new TOSCA definitions and their implementations designed and developed for this purpose. For example, within the RADON-SODALITE task force we migrated as well as developed several new TOSCA definints, i.e., TOSCA node types, to support HPC use cases. Further, added TOSCA definitions in this context show how to model and deploy a FaaS-based application to Kubernetes clusters running on edge components, such as Raspberry Pis.

External parties could proceed similarly to extend the RADON Framework with additional deployment targets. For example, software developers could use the RADON IDE to develop the respective TOSCA definitions using the provided editor with syntax highlighting for the TOSCA Simple Profile version 1.3. On top of that, for developers who are rather inexperienced with TOSCA, RADON offers the Graphical Modeling Tool (GMT) which also can be used to create new TOSCA definitions. GMT's Management UI provides a user-friendly interface to produce all kinds of TOSCA types in a syntax agnostic manner, while serializing the outcome in the actual TOSCA Simple Profile syntax to the RADON IDE. The custom extension can then be published either to the public RADON Template Library, to a private fork of it, or even to the RADON Template Library Publishing Service featuring lots of useful enterprise-related features, such as access-control and additional versioning.

Adding new target clouds and platform services. For new target clouds and platform services for applications new TOSCA templates have to be prepared to provide needed functionality to talk with provider's or cloud's API. Existing templates for deploying to clouds and platforms were prepared while studying APIs of already supported cloud providers. Already published TOSCA definitions can serve as a baseline for custom extensions.

In TPS CLI there is an argument `template create` to create a template with basic TOSCA and Ansible files. However customization of those files must be done manually using text editors, to adjust for the desired cloud provider.

Upon using TPS CLI package `xopera-template-library` with `template create` argument user is prompted to type in template type and template name. Created files use TOSCA definition of version 1.3. A Readme file is also generated with instructions for using the correct structure. See Figure 3.5.1 of the created structure using xopera-template-library package. As every cloud provider has a different API specifications there is no common way of preparing needed templates for orchestration, however a user can use TOSCA templates of other providers as a guideline.

[36] https://github.com/radon-h2020/radon-particles

Created templates can be uploaded to TPS through CLI or added to RADON Particles repository to share them with the rest of the users.

```
NodeExample/
├── files
│   ├── create.yml
│   └── delete.yml
├── NodeType.tosca
└── README.md
```

Figure 3.5.1: Directory structure of node type template with template create argument

Furthermore, adding support for new target clouds and services is possible via RADON GMT. When introducing new types with GMT no source code modifications are required: all actions can be performed by simply using GMT's Management UI. The underlying TOSCA definitions are automatically generated by the tool, while users only need to specify the structure for new modeling constructs, e.g., creating a Node Type, defining its properties, adding logos for improving its visual appearance in the Topology Modeler UI, etc. The process of modeling application topologies in RADON GMT is extensively described in the GMT's documentation[37]. Furthermore, more details on the extensibility of the RADON Modeling Profile can be found in the deliverable D4.4 RADON Models II.

Example: adding support for a Raspberry Pi deployment. The following customization example has been developed by RADON members to foster adoption of the solution in the context of the RADON-SODALITE working group.

New node templates were defined to deploy serverless functions on Raspberry Pi based private edge clusters. These include node definitions for MinIO data buckets and OpenFaaS deployed on a lightweight Kubernetes (k3s) cluster. MinIO is used for creating storage buckets in a private LAN node and follows APIs similar to the Amazon S3 bucket. We also integrated this with an Android based smartphone gateway to directly upload or download images to these MinIO buckets.

We then implemented custom OpenFaaS triggers that call the serverless functions on a `s3:ObjectCreated:*` event. We assume that the docker images are already available as part of a local docker registry in the master node of the k3s cluster.

To test the node templates, we created a sample application that generates thumbnails for input images, uploaded to a MinIO `SourceBucket`. This bucket triggers the `image-resize` function, hosted on an `RPi-Platform`, that saves the output image to a `TargetBucket`. We extend an open-source Android application to create a smartphone front-end UI. An overview of this demo application is shown in Figure 3.5.2.

[37] https://winery.readthedocs.io/en/latest/user/index.html

The code implementation for the demo tashk and the node templates are publicly available[38,39].

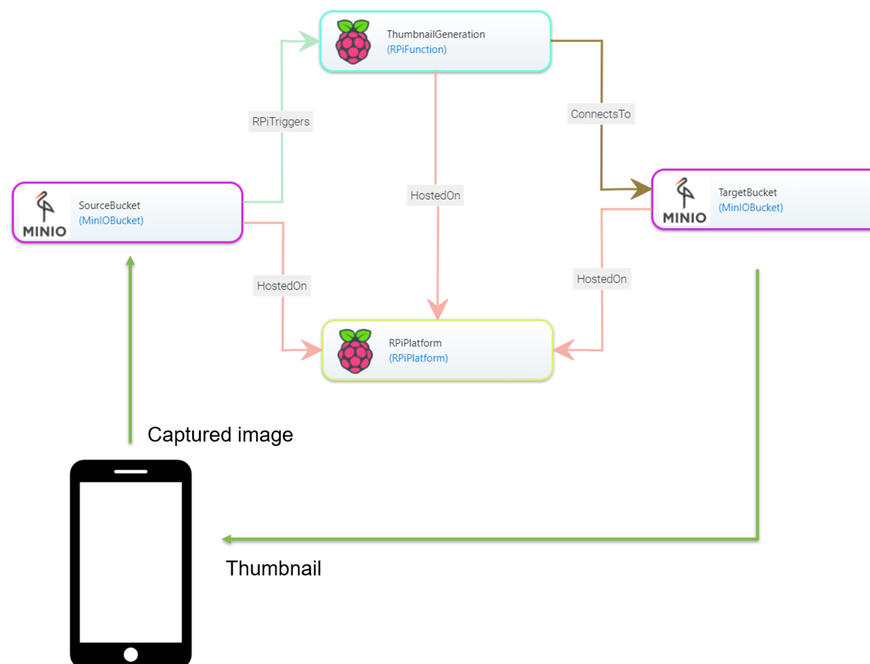

Figure 3.5.2: Winery topology of a simple thumbnail generation deployment and data flow with a smartphone gateway.

The RADON project was also verified to be customizable to build QoS management platforms. In particular, we wanted to deploy and test volatile workloads in dynamic fog computing environments using diverse scheduling schemes. The ability to automatically deploy cloud virtual machines with relevant software packages like Docker, Flask HTTP service and CRIU (Checkpoint-Restore In Userspace) of RADON was leveraged to develop the COSCO framework. COSCO is an AI based coupled-simulation and container orchestration framework for integrated Edge, Fog and Cloud Computing Environments. It's a simple python based software solution, where academics or industrialists can develop, simulate, test and deploy their scheduling policies [TPS+21]. This framework is available as an open-source GitHub repository[40].

The framework uses RADON's `AzurePlatform`, `DockerEngine`, `DockerApplication` node templates and deploys them using xOpera CLI. Further, the research work presents a novel gradient-based optimization strategy using deep neural networks as surrogate functions and co-simulations to facilitate decision making.

[38] https://github.com/RADON-SODALITE/demo-hcp-rpi
[39] https://github.com/RADON-SODALITE/hybrid-compute-profile
[40] https://github.com/imperial-qore/COSCO

3.6 Customize to other modelling languages or other version of TOSCA

While conceptually the RADON framework is not bound to a particular modeling language, the current state of implementation for the majority of the RADON tools implies the usage of the TOSCA specification for representing serverless application models. Here, it is important to highlight two different aspects: TOSCA, being a technology-agnostic standard, decouples application models from the actual deployment technologies with their underlying deployment modeling languages (typically, custom domain-specific languages such as Ansible's or Terraform DSL's). As a result, the customizability of the RADON framework can be analyzed from two different perspectives: (i) representation of deployment architectures, and (ii) modeling of the actual deployment logic. The former is related to the majority of the tools that require an application model, e.g., RADON's Graphical Modeling Tool, Decomposition Tool, or Continuous Testing Tool, and exactly this aspect is solved in the context of RADON using the TOSCA specification. On the other hand, the latter is related to implementation of reusable and deployable application building blocks and is related to the RADON Orchestrator, which is currently relying on Ansible as the underlying deployment technology.

While in theory it is possible to employ other modeling languages such as Serverless Application Model (AWS SAM) by Amazon, or Open Application Model by Microsoft, this requires modifications in all involved tools, with the complexity of such format migration to be estimated on a per-tool basis. For example, RADON Graphical Modeling tool is tightly-coupled with TOSCA: it supports both XML and YAML versions of the specification and enables transparently switching between both specifications despite their differences. The underlying canonical data model implemented in Graphical Modeling Tool also enables adding support for the upcoming TOSCA specifications. However, moving from TOSCA to other modeling languages would require significant reengineering efforts for both backend and frontend, since most interfaces are inherently coupled with TOSCA. Another important aspect to consider for such customization is whether the chosen modeling language is generic enough, e.g, AWS SAM focuses on Amazon's services. TOSCA, on the other hand, provides a generic way to represent serverless (and serverful) application models for different target environments which can then be used by RADON tools, e.g., to enable continuous testing or constraints verification.

Customization of the supported deployment technology in the RADON Framework can, on the other hand, be very beneficial, since multiple popular automation technologies such as Serverless Framework, or Terraform can then be used to implement reusable modeling constructs and RADON blueprints. This customization, however, requires significant efforts to enable technology-specific interaction on the RADON Orchestrator's side, e.g., invoking Terraform scripts instead of Ansible playbooks.

More in detail, the customization of the xOpera orchestrator to work with other versions of TOSCA includes the updates of the parser part, instantiator and executor - see Figure 3.6.1. As the xOpera

orchestrator is open source, this can be done by any community member that would require extending the xOpera orchestrator.

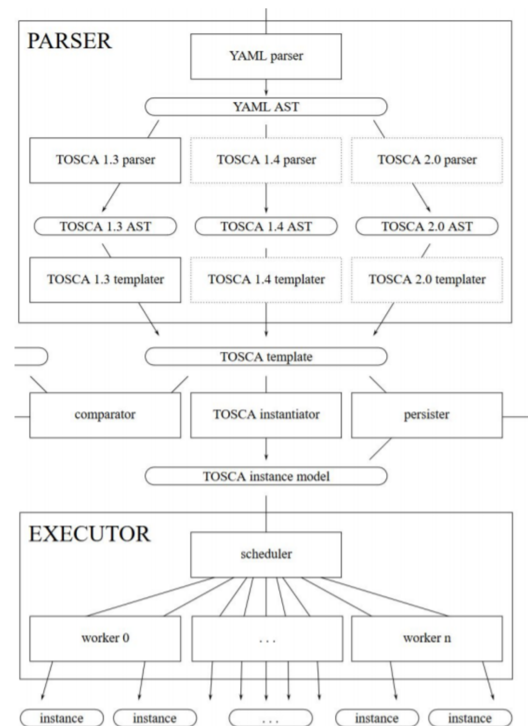

Figure 3.6.1: xOpera high-level architecture

3.7 Customize defect prediction to other IaC languages

The defect prediction tool currently supports only two IaC languages (i.e., Ansible and TOSCA) and five kinds of defects. However, more languages and defects could be easily integrated in the future.

Apart from changes to the graphical user interface (i.e., the plugin integrated with the RADON IDE), new models should be added. Please consider that the model building phase is language-agnostic. Nevertheless, new models need to be added, providing new software metrics definitions and labels that allow for representing and distinguishing the class to be classified.

This step is facilitated by *RepoMiner*, a language-agnostic tool that we provide developed to support software engineering researchers in creating datasets to support any study on defect prediction. The tool is open-source and available on GitHub[41] and the Python Package Index[42].

[41] https://github.com/radon-h2020/radon-repository-miner
[42] https://pypi.org/project/repository-miner/

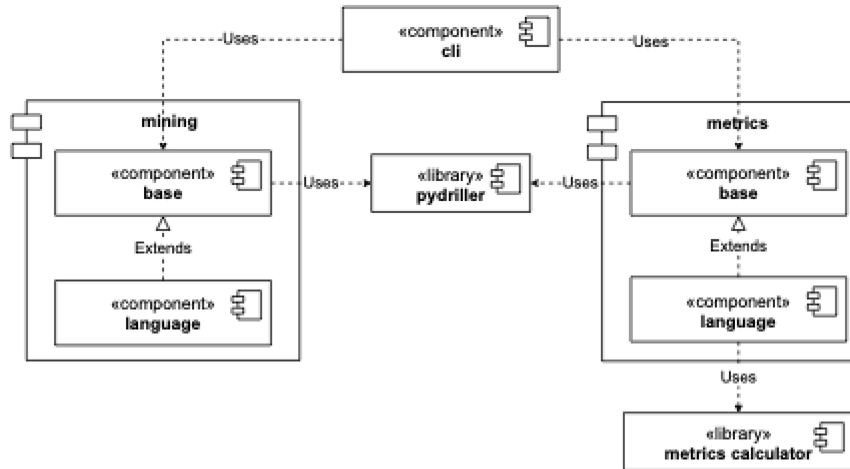

Figure 3.7.1: UML class diagram of the mining module.

RepoMiner essentially consists of two modules, namely mining and metrics, which is build on top of the PyDriller framework[43] to analyze the project history, identify relevant commits, and failure-prone files, and extract process and source code metrics from them.

As its name suggests, a *base* component provides the base functionalities for that module. Supporting a new language requires creating a *language* component that extends the *base* functionalities to fit the language.

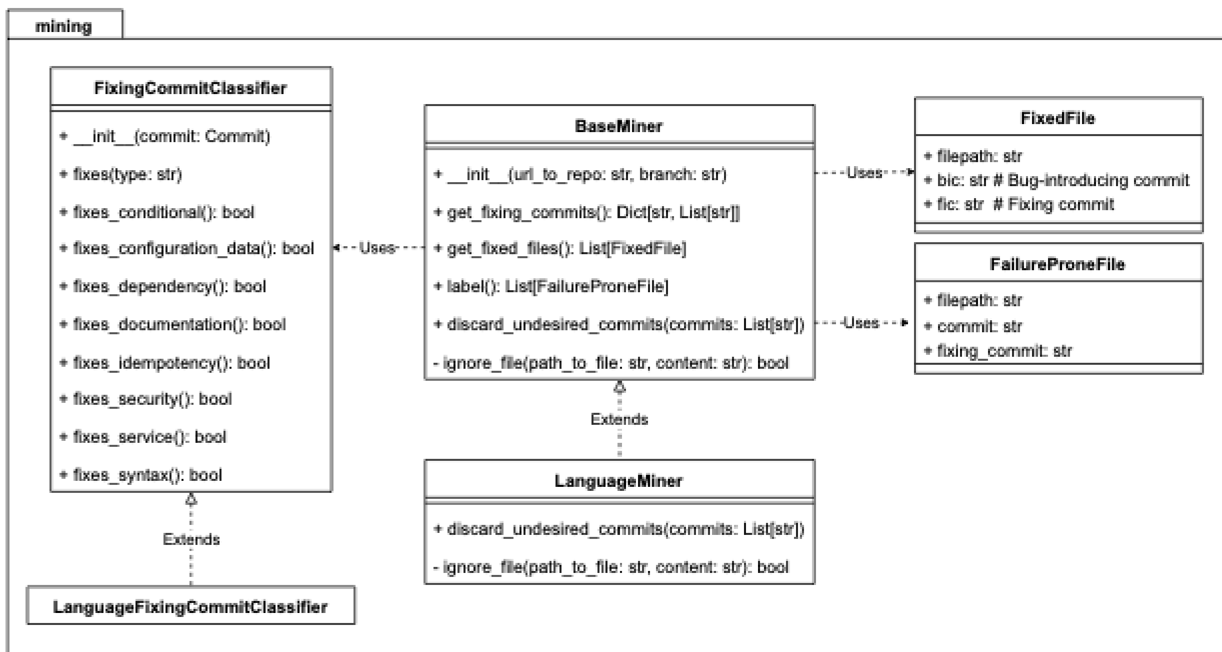

Figure 3.7.2: UML class diagram of the mining module.

[43] https://github.com/ishepard/pydriller

As depicted in Figure 3.7.2, *BaseMiner* is an abstract class that other classes extend to mine specific language-based repositories. It takes in input the URL to a remote git repository and the repository's branch to analyze. Then, the mining is made possible by the three methods below, to be executed in order: *get_fixing_commits()*, *get_fixed_files()*, *label()*.

Identifying defect-fixing commits. The *get_fixing_commits()* function identifies defect-fixing commits, if any. Defects fall under the eight categories derived by [Rahman2020] through a qualitative analysis on defect-related commits collected from the open-source software repositories of the Openstack organization: ``conditionals", ``configuration data", ``dependencies", ``documentation", ``idempotency", ``security", ``service" and ``syntax". Given a commit and a defect category, the function identifies whether the commit message or the changes to one of the modified files indicate a fix based on the rules defined in. If so, the commit hash is added to the dictionary of defect-fixing commits with the respective categories. Finally, it keeps only the commits that modify at least one file for the language at hand (line 14) and returns the dictionary of defect-fixing commits.

Identifying fixed files. The *get_fixed_files()* function identifies files of a given language modified in a defect-fixing commit and their *bug-introducing* commit; hereon, we refer to them as *fixed files*. To this end, it analyzes the commits backward from the most recent defect-fixing commit to the oldest Given a file, the function first checks if it belongs to the language analyzed and whether the file has been modified based on its content and metadata. Then, the function uses the *SZZ* algorithm [Kim2006] implemented in PyDriller[44] to automatically identify the oldest commit that introduced the defect in that file, known as the *bug-introducing* commit (bic). The hash of this commit, along with the *defect-fixing* commit (fic) hash and the *filename*, is used to create a new *FixedFile*, which is added directly to the list of fixed files. Successively, the following steps apply:

1. If the current defect-fixing commit is older than the file's previous bic, a brand new object is appended to the list of fixed files.
2. If the file's previous bic is between its current bic and fic, the existing bic is updated with the current one.

The function returns the list of *FixedFile* objects that can be used by the following function to label the files as *failure-prone* at a given point in the repository's history.

Labeling defective files. Given the list of fixed files, the *label()* function labels all the snapshots of a *FixedFile* between its bic (inclusive) and its fic as *failure-prone*. Then, for each *FixedFile* at a given point in the repository history, it yields a *FailureProneFile* object consisting of the *filepath*, the *commit hash* at that time, and the hash of its *defect-fixing commit*. It is worth noting that *BaseMiner* has two additional methods, namely *discard_undesired_commits()* and *ignore_file()*, which must be implemented by the sub-classes to filter commits and files for a given language. The former discards the fixing commits that do not modify files of the language considered. The latter

[44] Release >= 1.16.

returns a boolean value indicating whether to ignore a given file based on its path, extension, and/or content. For example, if the user aims at analyzing Python-based repositories, the method will ensure that all non *.py* files are ignored.

FixingCommitClassifier is an abstract class for categorizing fixing-commits based on the categories mentioned above. It provides base implementations for each category, although some have to be overridden by the classes extending it depending on the language. For example, the methods *fixes_configuration_data()* and *fixes_idempotency()* in Figure 3.7.1 relate to defect categories specific to language for configuration management and infrastructure provisioning and do not apply to application code languages. By contrast, methods such as *fixes_dependency()* are common to different languages, although the implementation can slightly change depending on the language syntax. Therefore, a concrete *LanguageFixingCommitClassifier* class is implemented every time a new language is added, and the related methods are overridden. Finally, an instance of *LanguageFixingCommitClassifier* must be initialized in the constructor of the respective *LanguageMiner*.

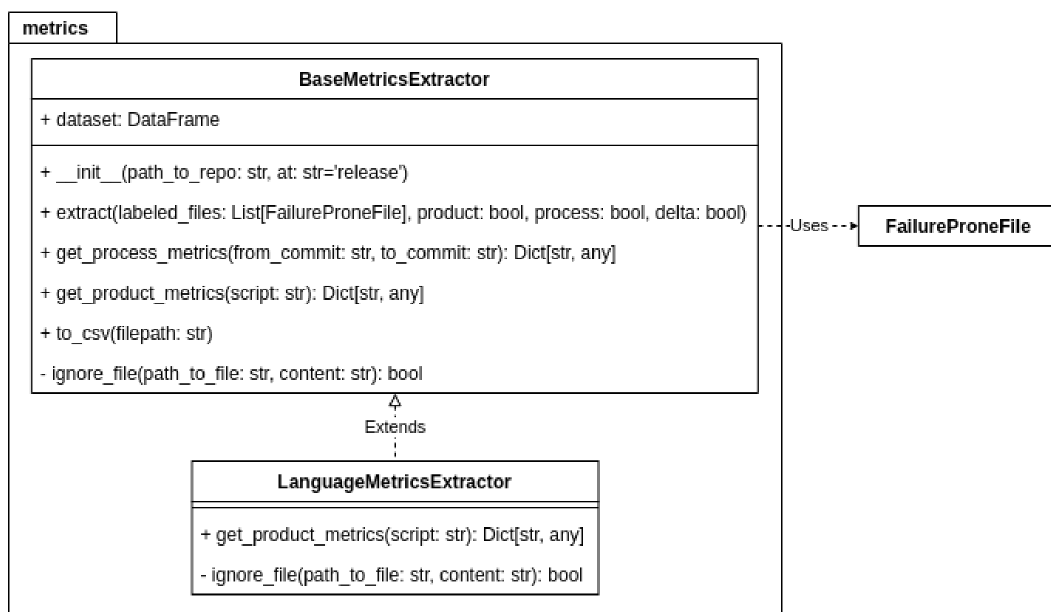

Figure 3.7.3: UML class diagram of the metrics module.

Metrics Extraction. As shown in Figure 3.7.3, *BaseMetricsExtractor* is an abstract class for extracting process and source code metrics from the collected files, and creating a dataset of *failure-prone* and *neutral* observations ready for defect prediction. It is extended by the classes accountable for extracting source code metrics related to a specific language (i.e., *LanguageMetricsExtractor*). It takes in input the path to the local or remote git repository and the repository's branch to work on. The entry point is the method *extract()*, which requires a list of *FailureProneFile* objects and the type of metrics to extract, namely process, product, delta, or a group thereof.

It uses the methods *get_process_metrics()* and *get_product_metrics()* to extract process and source code metrics, respectively. Relying on the development activity rather than the source code, process metrics are language-agnostic and do not need any further extension. In contrast, the product metrics are language-dependent because their definition and implementation change across languages. For this reason, the *LanguageMetricsExtractor* subclasses must implement that method. *BaseMetricsExtractor* provides the method *to_csv()* to save the dataset locally, albeit the user can programmatically access it. Finally, similar to the mining classes, *LanguageMetricsExtractor*s must implement the *ignore_file()* method to discard files of different languages.

3.8 Customize to other orchestrators based on TOSCA or not

The main orchestration tool that is used to deploy services within RADON is the xOpera orchestrator which supports TOSCA as its main orchestration standard (the current compatibility is with TOSCA Simple Profile in YAML Version 1.3). Therefore, it would be possible to use other orchestrators for example Yorc, Puccini or Cloudify that support the same version of TOSCA YAML to deploy RADON services. Unlike opera, which is minimal and client-based, all these alternative orchestrators usually require a server-client architecture, which means that in case of switching to one of them, we would need to set up an orchestration server before initiating client deployments. Before the customization to another TOSCA orchestrator it would also be reasonable to check, whether it supports Ansible executor because most of RADON TOSCA definitions use Ansible playbooks as actuators that implement TOSCA interface operations. Another thing to consider here is that in case of changing the main orchestrator, the users would not be able to fully use the easier way of deploying with xOpera SaaS component (API and GUI), which is now adjusted to opera and would need to be reshaped in order to prepare the necessary orchestration environments for other orchestrators. The same usage factor applies for other RADON plugins, which would need to be modified to successfully communicate with the new potential TOSCA orchestrator. Although each of alternative orchestration tools has its own specifics, it is always possible to run and test them locally and deploy RADON applications any time, without the need to refactor any of the current RADON tools.

Since RADON models (e.g. RADON Particles) are designed as reusable TOSCA YAML templates, it is required to have a TOSCA orchestrator to deploy the defined services. This dependency also applies for other RADON tools such as GMT, VT, DT, DPT, CTT which are more or less tailored to the use of TOSCA definitions and its specific language. To support one of non-TOSCA orchestrators (i.e. Terraform, AWS CloudFormation, OpenStack Heat, etc.) would mean decoupling all RADON tools and models from the TOSCA standard. This could take a bigger amount of effort as we would have to redesign all RADON TOSCA definitions to a new DSL. On the other hand, it would be possible to reconstruct the SaaS orchestrator component in a way that it would be able to set up other orchestration environments in the backend containers. It would be possible to support other orchestrators as plugins that the users could supply and use.

3.9 Customize data pipelines to support additional data sources

Adding support for additional data sources. Since RADON data pipeline blocks are implemented as TOSCA models, new data sources can be relatively easily introduced into RADON data pipeline supported services as long as the currently supported data pipelines technologies (Apache NiFi, Amazon Data Pipeline service) are compatible. AWS Data pipeline service supports a subselection of the AWS services but Apache NiFi has an extensive list of external services that can be used as data sources.

From the viewpoint of the RADON project, such a process requires designing new TOSCA node types. However, existing nodetypes can be extensively reused and only few modifications are required to adapt them for new node types. To add support for additional data sources, it is required to:

1. Decide which type of node type is required (check Figure 3.9.1)
    a. **Source Pipeline Block** - acquiring data from an external data source.
    b. **Midway Pipeline Block** - transforming data either locally or sending it to an external service (e.g., FaaS function) to be processed.
    c. **Destination Pipeline Block** - sending data to an external data source.
    d. **Standalone** - special type of data pipeline which contains everything needed and is not directly composed with other data pipeline node types.
2. Implement the specific data pipeline service using the tools of the supported technology (e.g, NiFi web interface).
3. Export the developed software artifact as a file (e.g. XML, JSON).
4. Use RADON GMT to design the respective TOSCA node type.
    a. Define all the input parameters and attributes.
    b. Existing Ansible life-cycle (create, start, configure, stop, delete) playbooks of the supported technology can be directly reused.
        i. Configure script needs to be modified for every unique user defined parameter (e.g., MQTT topic name).
        ii. Create script needs to be modified if some custom software libraries are required.

Once the node type is ready it should be validated through deployment testing by designing a data pipeline service template and using xOpera to deploy it. After that, the node type can be exported as a CSAR or committed to a git repository.

Example: Developing new gridFTP data pipeline node types using NiFi. To illustrate how the customization process works, this section outlines how to design new GridFTPdata pipeline node types. These node types were specifically designed for the RADON-SODALITE project collaboration, where we aimed to combine the outputs of the two projects into a single

demonstrative use case, and GridFTP support was required for RADON data pipelines to enable real time data migration between HPC and Cloud environments.

Adding support for additional data transfer technologies. To add additional underlying technologies in addition to Apache NiFi and AWS data pipelines, a new set of data pipeline node types ended up being created, together with new Ansible scripts. Depending on how close the chosen technology is to the current ones, this process can require significant development and testing.

This process is similar to the usual TOSCA node type design and implementation process. The main difference is that the design of the node types should follow the data pipeline methodology (illustrated on Figure 3.9.1) where the data pipeline services are divided into individual, reusable and composable node types which can be composed both at design time and runtime using the RADON orchestrator.

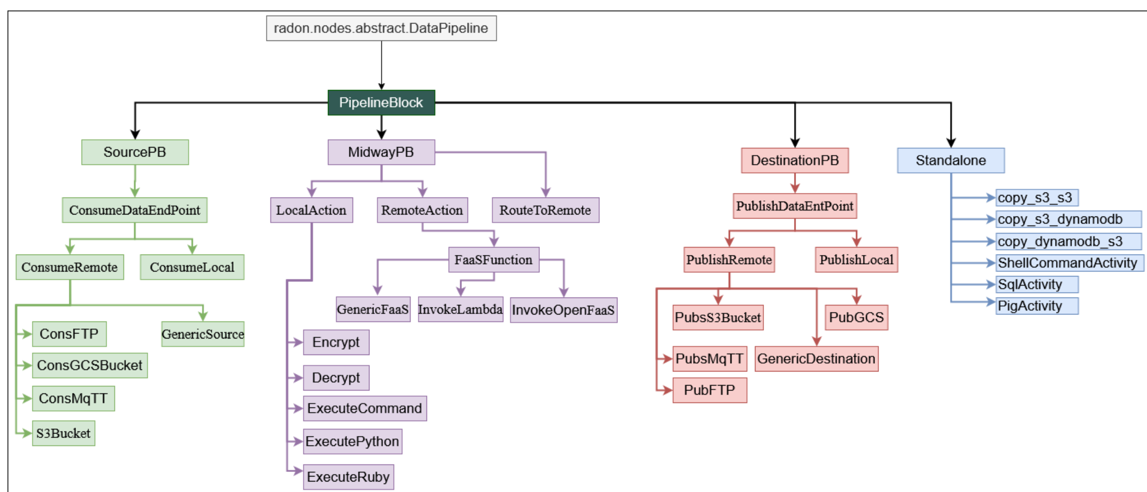

Figure 3.9.1: Data Pipeline TOSCA model hierarchy [D5.6]

Also, the data pipeline orchestration plugin will not work with a new set of data pipeline node types and would also have to be extended. This process requires adapting and testing the current TOSCA parser logic to take into account any unique characteristics of the new node types. However, as long as the TOSCA node types are designed properly, the RADON framework can be used to model, configure and deploy the new node types.

3.10 Customize to other trigger relationships/events

The definition of new trigger relationships and events is important to be able to support the extension of RADON to novel serverless platforms. The existing RADON nodes and relationships provide the insight into deploying projects for different serverless platforms. While new trigger relationships are required to support more cloud services and complicated scenarios. Users can extend their own node and relationship templates by customising RADON particles, similarly to

what described in earlier sections, in order to support user-defined use cases or services and relation between different node templates. The extension of the new relationship would involve small changes like adding new valid source nodes in the existing nodes.

For example, to deploy Azure Durable Function[45] for HTTP-triggered function chaining, new node and relationship templates are defined, including nodes for *Azure Durable Orchestrator* functions and *Azure HTTP-triggered functions*, and a relationship between an *Azure Blob Storage-triggered event* and Orchestrator function, and a relationship that connects Azure HTTP-triggered function to the Azure Orchestrator function.

The node *AzureDurableOrchestrator* is created to present the orchestrator function that can chain other Azure functions in a specific execution order. We also customise a node *AzureHttpFunction* for deploying HTTP-triggered functions. This node is used for individual functions in function chaining. The orchestrator function is triggered by uploading events in Azure Blob Storage[46]. A new defined relationship template *DurableBlobTrigger* is used to configure the app settings for the orchestrator function and start the orchestrator function when a new uploading event occurs. To connect the local HTTP-triggered functions to the orchestrator, a relationship between them is defined as *ConnectToDurable*. More in detail, this relationship helps to pass parameters like the URL of HTTP-triggered functions to *AzureDurableOrchestrator*. A logical diagram for an Azure Durable function example with two functions in the chain is shown in Figure 3.10.1.

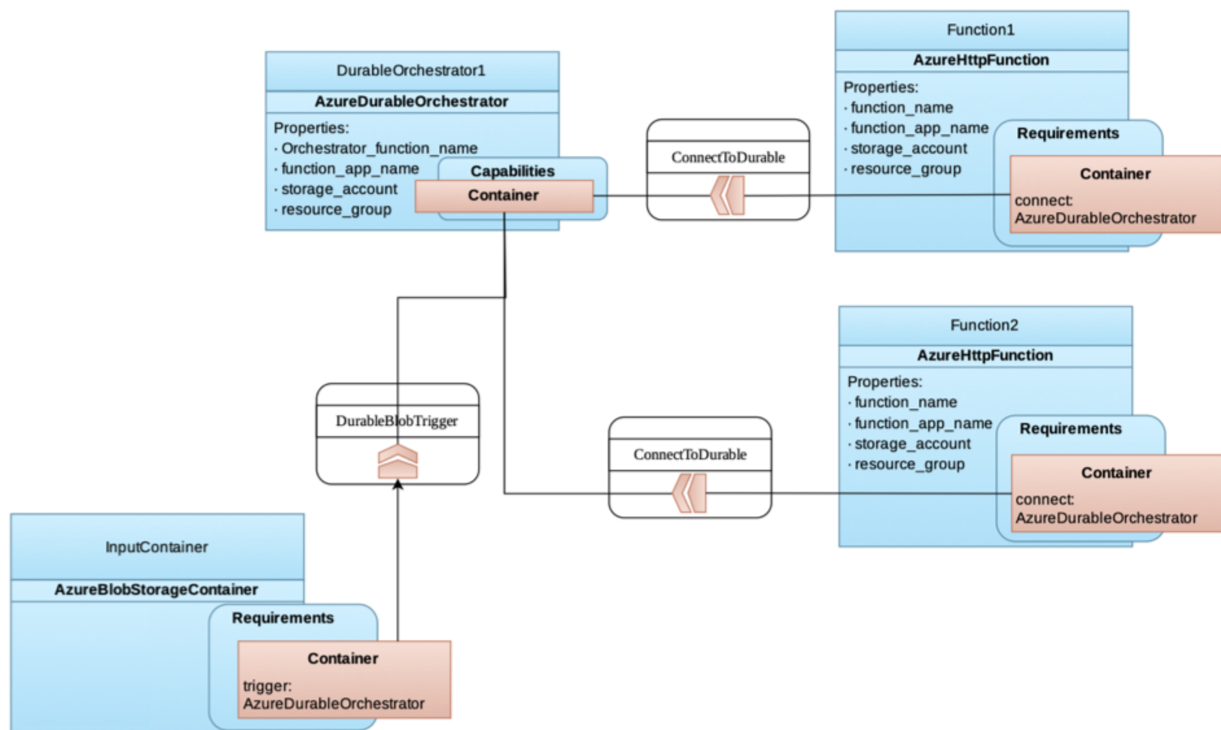

Figure 3.10.1: Logical diagram of local deployment for Azure Durable Function

[45] https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp
[46] https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-blob-trigger?tabs=csharp

3.11 Customize to other Monitoring

The customisation of other monitoring tools through external repositories, standalone libraries and toolboxes provides the decomposition tool (DT) with a more complete and robust set of tools for the user when using RADON. In this direction, the DT architecture allows a straightforward extension to arbitrary monitoring system inputs by the subtle introduction of enhancement functions in the *RootNode* and *Function* Abstract Classes. These functions should be aimed to modify the nodes' properties, and reflect the changes in the tosca model template using the results obtained from external estimation functions located in external repositories.

For example, for AWS Lambda functions, the DT has undergone a customization exercise to demonstrate typical challenges and solutions that need to be addressed in this scenario. We remark that to handle this extension the DT needs to be augmented with a specific set of functions for demand estimation that will update the .tosca service template located in the same directory as the unique .txt file containing the monitoring data traces that are used as input and for the estimation.

We have developed one such customization code for reproducibility[47], using as reference customized monitoring platform AWS X-Ray, that was not supported in earlier RADON iterations. It is important to mention that, AWS X-Ray is a distributed tracing system that records the information of a request processed by one or more services that, compared to other distributed tracing tools like Jaeger or Prometheus (the latter already integrated in RADON), X-Ray is natively supported by AWS Lambda, which can be easily enabled and instrumented during development and deployment without operating and configuration on other Lambda layers.

An additional challenge that we have experienced was to integrate the monitoring with xOpera, to ensure automated function instrumentation. Such extension has been done for AWS X-Ray as follows. Ansible playbook allows defining and executing users' tasks, in this way users can instrument on AWS Lambda API to update the tracing configuration for the Lambda function. An example of the instrument on the Ansible playbook to enable tracing is shown in Figure 3.11.1.

```
- name: Enable AWS X-Ray
  command: >-
    aws lambda update-function-configuration
      --function-name {{ function_name }}
      --tracing-config Mode=Active
```

Figure 3.11.1: An example to enable AWS X-Ray in the Ansible playbook

In order to trace requests in Lambda, users can first instrument language-specific codes with X-Ray SDK by creating custom subsegments in the Lambda handler. AWS X-Ray can be then activated during deployment with Lambda API, then users can use AWS CLI to get the trace information in a .txt file. The generated trace contains two segments, one for the Lambda service and one for the

[47] https://github.com/radon-h2020/radon-decomposition-enhancement

work done by the function. For the latter, subsegments on initialisation (only provided the first time a function is called), invocation and overhead are recorded. For a higher level of granularity, it is also possible to inject code so that the invocation subsegment contains further layers of information (such as name, start and end time, etc.) regarding particular activities (Figure 3.11.2).


```
5.038             1-60b0b8a2-4712268c6817e09437642c0a     False

SEGMENTS
{"id":"2ec9f65991d3e7e8","name":"nerl2","start_time":1.622194338306E9,"trace_id":"1-60b0b8a2-4712268c6817e09437642c0a","end_time":1.622194338341E9,"http":{"response":
{"status":202}},"aws":{"request_id":"5ce9dedc-5bed-479f-bf1d-db8eedabd7da"},"origin":"AWS::Lambda","resource_arn":"arn:aws:lambda:eu-
west-2:725152221712:function:nerl2","subsegments":[{"id":"5bf225dd63a3de94","name":"Attempt #1","start_time":1.622194338364E9,"end_time":1.622194343344E9,"http":
{"response":{"status":200}}},{"id":"63fe53843552005c","name":"Dwell Time","start_time":1.622194338306E9,"end_time":1.622194338364E9}]}     2ec9f65991d3e7e8

SEGMENTS
{"id":"5c887efe4ac4fd81","name":"nerl2","start_time":1.6221943383747206E9,"trace_id":"1-60b0b8a2-4712268c6817e09437642c0a","end_time":1.622194343342201E9,"parent_id":
"5bf225dd63a3de94","aws":{"account_id":"725152221712","function_arn":"arn:aws:lambda:eu-west-2:725152221712:function:nerl2","resource_names":
["nerl2"]},"origin":"AWS::Lambda::Function","subsegments":
[{"id":"533339642bc34612","name":"Invocation","start_time":1.6221943383747725E9,"end_time":1.622194343341808E9,"aws":{"function_arn":"arn:aws:lambda:eu-
west-2:725152221712:function:nerl2"},"subsegments":[{"id":"21815220f4af71ff","name":"nlp-
process","start_time":1.6221943383941057E9,"end_time":1.6221943429801881E9,"in_progress":false,"annotations":{"id":3.0},"namespace":"local"},
{"id":"78fc6327e2b57123","name":"parse-event","start_time":1.6221943383755536E9,"end_time":1.622194338375615E9,"in_progress":false,"annotations":
{"id":1.0},"namespace":"local"},{"id":"305ee8deeeae856c","name":"delete-
payload","start_time":1.6221943431849303E9,"end_time":1.62219434330621E9,"in_progress":false,"annotations":{"id":4.0},"namespace":"local"},
{"id":"c9399e29e4e05f76","name":"put-response","start_time":1.6221943429804592E9,"end_time":1.622194343184709E9,"in_progress":false,"annotations":
{"id":4.0},"namespace":"local"},{"id":"073c90a7fe490cc8","name":"get-
payload","start_time":1.6221943383759077E9,"end_time":1.62219433839391E9,"in_progress":false,"annotations":{"id":2.0},"namespace":"local"}]},
{"id":"4eb3a7a21ed933c7","name":"Overhead","start_time":1.6221943433418484E9,"end_time":1.6221943433421712E9,"aws":{"function_arn":"arn:aws:lambda:eu-
west-2:725152221712:function:nerl2"}}]} 5c887efe4ac4fd81
```


Figure 3.11.2: Example of an AWS X-Ray generated trace. Different segments allow extracting information for DT's accuracy enhancement feature.

3.12 Customize to other test types and tools

CTT has been designed to be extensible to custom test types and test agents, e.g., load generators. The respective extension points in the CTT architecture are described in detail in the deliverables [D3.4] and [D3.5].

- New test types and tools are defined by adding or refining (extending) TOSCA policies, node types, and service templates in RADON's TOSCA modeling type hierarchy and by providing Ansible deployment artifacts.
- The CTT server must be extended by adding a new server module that is able to interpret the new test policies and to communicate with a suitable test agent, which essentially wraps the actual test tool (e.g., load generator). The CTT test agent may be an existing one for an existing tool, or a newly developed one for new test tools.

TOSCA Types. To extend the CTT with additional custom tests and testing tools, the following steps are necessary.

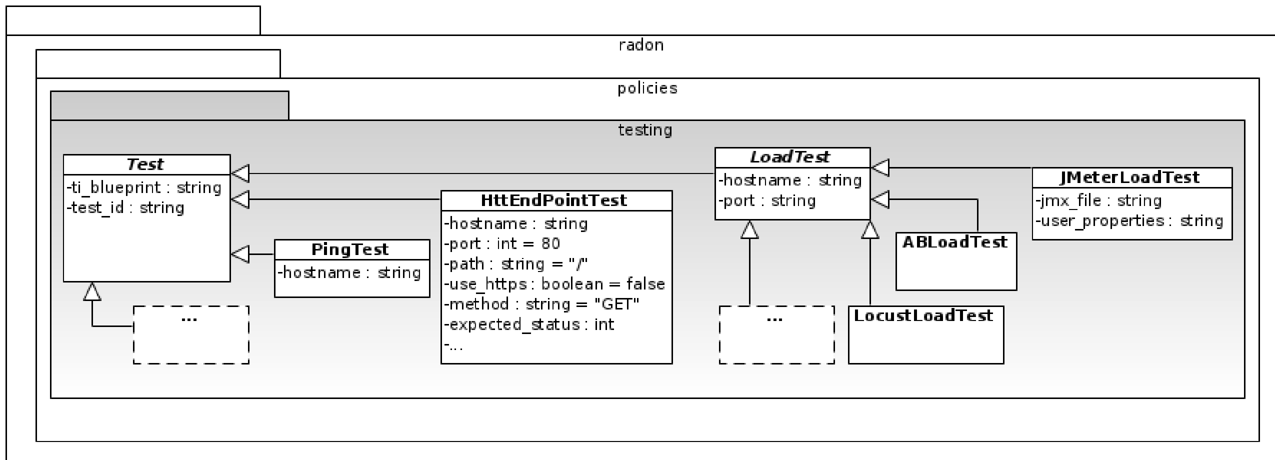

Figure 3.12.1: CTT modeling type hierarchy for test policies

1. Define custom test policies that provide details about the test parameters (e.g., test duration, concurrency, payload) your testing tool needs to configure the test execution. These types are integrated similarly to the JMeter test type (JMeterLoadTest) in the type hierarchy presented in Figure 3.12.1. Together with the definition, also a deployment description in the form of an Ansible playbook is required.

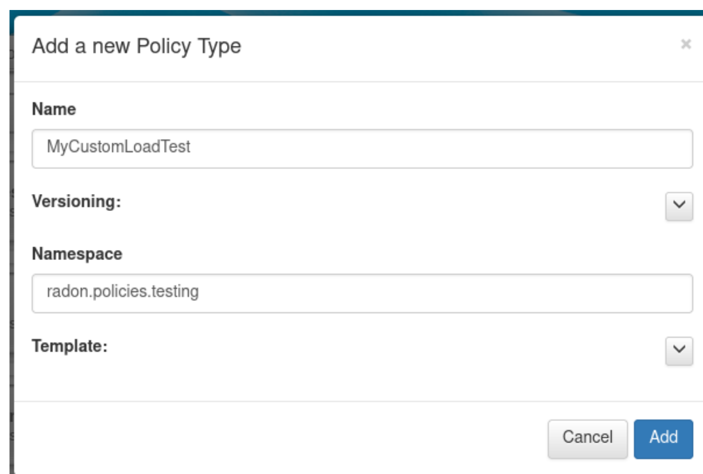

Figure 3.12.2: Add Policy Type dialog in RADON GMT

To add a custom testing policy, open the RADON GMT tool and select “Other Elements” and then “Policy Types”. On the right side, click on the button “Add new”. In order to add the new testing policy, which we will name “MyCustomLoadTest”, fill in the name and select the namespace according to Figure 3.12.2.

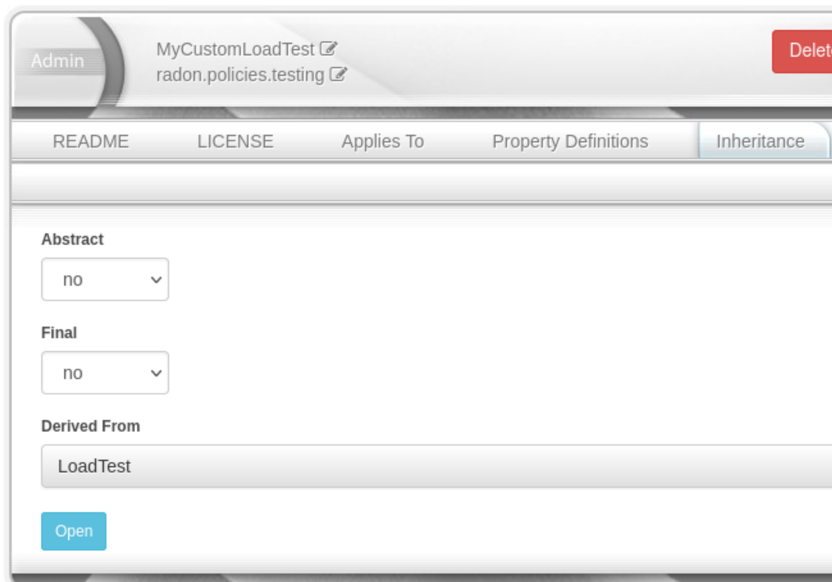

Figure 3.12.3: Inheritance view for the newly created testing policy

To inherit from an existing type (e.g., “LoadTest”), choose the respective type in the “Inheritance” tab in the drop-down menu under “Derived From” of the newly created testing policy (see Figure 3.12.3)

In the tab “Property Definitions” custom properties can be added.

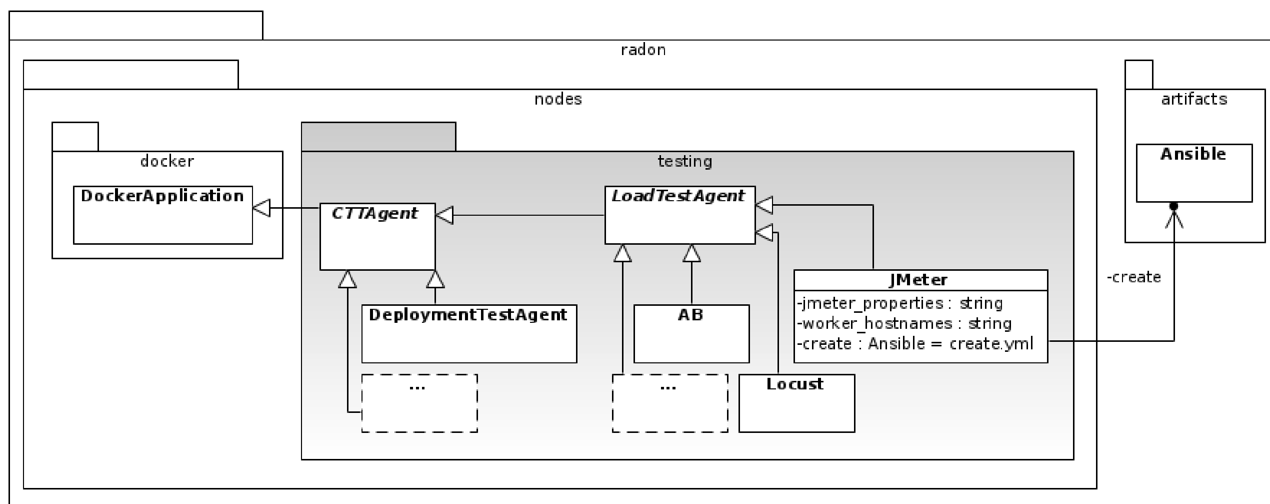

Figure 3.12.4: CTT node types for test tools

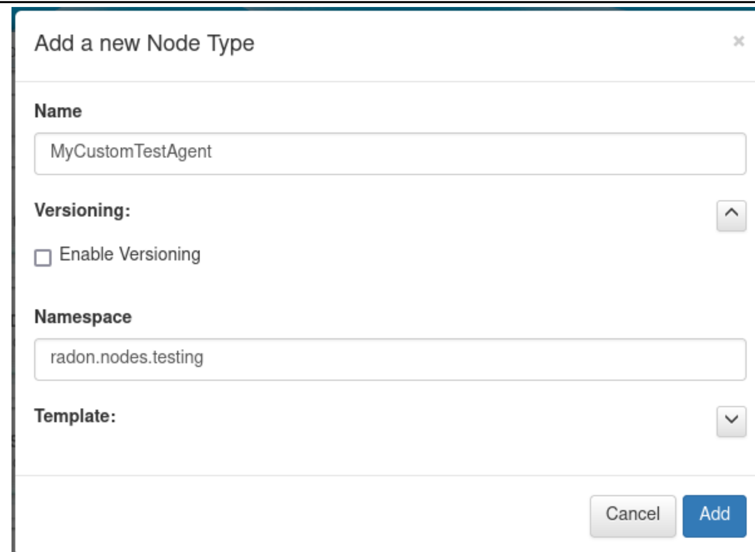

Figure 3.12.5: Create Node Type dialog in RADON GMT

2. Define a custom TOSCA node type (referencing an Ansible playbook) that represents the CTT agent including the testing tool. Figure 3.12.4 depicts the node type hierarchy including existing tools and the extension points.

   Then, similar to the steps for the testing policies, create a new node type for the test agent that inherits from the respective entities in the “radon.nodes.testing” namespace as depicted in Figure 3.12.5.

   The created node type can then be filled with the respective properties and an Ansible playbook that defines the deployment of the test agent using the previously created Docker container.

3. As a last step in the modeling context, the policies and types need to be integrated into the service templates and blueprints of the actual application that acts as the system under test (SUT) as well as the newly created test infrastructure (TI).

CTT Integration. To integrate a custom testing tool, there are two modules that need to be implemented. The CTT agent module controls the custom testing tool and acts as an interface between the CTT server and the testing tool. The other module is the CTT server module that, after reading a policy for the custom testing tool, transforms the information and controls the CTT agent with the custom agent module to act accordingly.

For both, the server and agent module, we provide a plug-in framework with examples that allow easy extension. Plugins are implemented in Python 3 and can be easily integrated using the Python Flask framework.

In order to reuse the modular structure for a CTT agent module, we recommend to clone the CTT Agent Plugins repository on GitHub[48] and create a new folder for the custom test agent based on the other examples. Each folder contains three files:

- `Dockerfile` (describing how to package the agent as a Docker container, including all necessary artifacts (e.g., executables))
- `__init__.py` (implementation of the integration between the REST endpoints and the test agent)
- `requirements.txt` (Python requirements file, listing all necessary modules for the execution)

After all parts are customized, the resulting Docker container can be built and uploaded to a container repository (e.g., DockerHub[49]). Finally, it can be referenced in the Ansible playbook for the respective test agent node type.

On the CTT server side, a fitting module to the parameters and interfaces defined in the `__init__.py` of the CTT agent module needs to be implemented. The implementation should be located in the `ctt-server/plugins` directory in the server[50], and follow the pattern of the other plugin files shipped with CTT located in the same directory.

48 https://github.com/radon-h2020/radon-ctt-agent-plugins
49 https://hub.docker.com/
50 https://github.com/radon-h2020/radon-ctt/

4. Adopting the RADON DevOps Process: Principles and Practices

To properly understand the process of adopting a DevOps-intensive technology, such as the RADON IDE and framework around it, we conducted an empirical experiment [Caprarelli2019]. We applied ethnographic methods [Willis2000] within an industrial reality in which a similar adoption scenario was unfolding as RADON was being incepted and in its mid-stages (M12 onwards). As part of the aforementioned adoption scenario, a Rational-Unified Process (RUP) shaped around five industrial-strength resource management and supply-chain management products migrated to a DevOps intensive, serverless-supported pipeline. This section recaps the fallacies and pitfalls encountered during our ethnographic study of the aforementioned scenario. We expect these lessons learned in this study to act as basic organizational and technical design principles to adopt the RADON solution properly.

4.1 Context of study

The studied organization is part of a big multinational operating in many sectors from databases to cloud products, from Infrastructure-as-a-Service (IaaS)to Software-as-a-Service (SaaS). The unit considered in the study is the consulting unit of the company. It focuses on selling and setting up EnterpriseResource Planning (ERP) and Customer-Relationship Management (CRM) for medium and big enterprises, both of which fall into the technical targets for RADON . In particular, the team that has been involved in the research study works as a system integrator to integrate the several systems involved in this context. This practice is also known as Enterprise Application Integration, and its main goals are:

- Data integration: maintain data consistency in multiple systems.
- Vendor Independence: business rules are implemented in the integration layer to avoid strict dependencies with the final applications.

The case under study features 1+5 projects: one sub-project for common and reusable components that shape the core architecture of the project (referred to as SP-C) and five sub-projects. Each of the former addresses different business areas of the main project (referred to as SP-x with x from 1 to 5). The two types of sub-projects have different team compositions, organization, and goals. Our study has focused on all the projects above throughout our longitudinal observational study. Again, all such products fall well within the RADON technical space.

Overall, the migration process was instrumented in three stages, with the incremental adoption of Continuous Integration (CI) practices, followed by Continuous Deployment (CD) practices, and finally, the instrumentation of a full-fledged DevOps pipeline. This evolution, the steps, and the required automation are recapped in Figure 4.1.1.

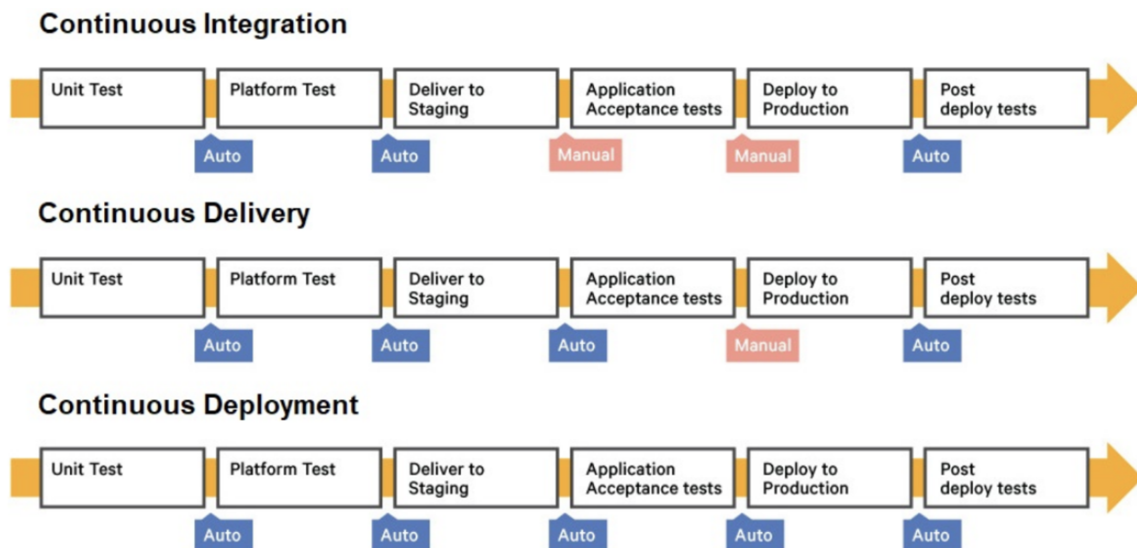

Figure 4.1.1: Continuous Integration, Delivery, and Deployment steps

For the sake of space and conciseness, in the following, we offer more details on the high-level principles, practices, benefits, and challenges emerging from the adoption process we observed in the target case study. Further details can be found in the full technical report for the same study [Caprarelli2019].

4.2 From RUP to CI/CD: adopted practices and emerging challenges

CI	Principles	Continuous integration puts a great emphasis on automation. Its core practice and enabler is the automation of the build process. On top of this, and the second main point of CI, there is test automation, which is important for checking that the application is not broken whenever new commits are integrated into the main branch. Another key practice of CI is to commit to the main code baseline daily or whenever an atomic development is completed. Each commit should automatically trigger a build and test pipeline to assure that no breaking code is merged into the main repository.
	Benefits	One of the greatest benefits of CI practices in our case is the reduced risks for developers when code integration is carried out. Bugs will always exist, but it is easier to find and remove them by having frequent tests. This faster error detection leads to higher productivity and more code quality. Such faster error detection also increases the deployment frequencies, thus reducing the time for feedback from end-users.
	Challenges	However, introducing practice was not pain-free. Organizations may face many challenges and problems in different areas, starting from the people to the technologies used, when they try to introduce more automation in the delivery pipeline. The most commonly found challenges were: (1) Smaller changes and often commit: this can be a paradigm shift for developers used to release big parts of code; (2) Maintain a fast-running set of comprehensive automated unit tests: since developers should run tests often, those tests must be optimized, and the tests suite should be easy to maintain and to enhance; (3) Set-up and maintain the build and test system: this requires effort and work, and it can become complex and costly. However, there are both proprietary and open-source software that can help with this; (4) Dealing with large monolithic applications: normally, legacy

		software is large and monolithic, and it fits with difficulty in a speed-oriented pipeline. The challenges include all the areas of a company. The process followed for delivering software may need some changes as well as the architecture of the components. Also, people need to be trained, and others with a diverse skillset may need to be hired.

CD	**Principles**	As in CI, the core principle is the automation of repetitive tasks and should be based on the execution of tests in an environment that should be as similar as possible to the production one. In addition to the tools-related practices, it also requires a change in people's mentality and process. Big releases with a long and tense integration period are discouraged, and people must collaborate more often to solve the problems that may arise quickly. The main code base should always be solid without failing tests.
	Benefits	● **Reduced Deployment Risks**: smaller changes are deployed at every push, and therefore, the possible problems and fixes came to be smaller; ● **Faster user feedback**: once a fix or feature is ready, it can be shipped to production, and feedback from end users can be received earlier. ● **More reliable releases**: at every push, the build system runs a test suite and goes further only if all tests are successful. Also, since everything is automated, rollback in case of failures is almost painless. ● **Less stress**: the responsibility of a release is distributed among many actors of the delivery pipeline, such as IT operations, testers, developers.
	Challenges	A key challenge that emerged after adopting CD was the creation of substantial and durable test cases. More specifically, in connection to CI, tests must spread from code tests (unit tests) to end-to-end tests (UI tests). However, test suites showing high coverage percentages alone cannot ensure code quality since intermittent, non-deterministic, or unknown bugs may remain hidden. Approaches such as chaos tests are being developed to cope with these.

4.3 Technical and organizational adoption fallacies

This section reports on the technical and organizational adoption fallacies encountered in the case under study and could not be overrun without additional (semi-)automated support. For the scope of this section, the labels {IC, FC, TST, DEL, OPS, INT} identify the roles of the 40+ people involved in the development and operations of products maintained in our industrial case. The fallacies are reported in the table below.

Fallacy	Description
Test Latency	Integration Consultants (IC) and Functional Consultants (FC) eventually had to wait for a long time before a release from development was deployed to the TST environment for final testing. This behavior is due to insufficient Delivery (DEL) and Operations (OPS) teams, which are a bottleneck for the delivery process. They are composed only of two people that have to serve at least 20 DEVs and a similar number of OPS.
Technical Re-skilling	The automation tools used to build and deploy require technical skills (Linux, Bash, network, DBs, cloud instances configuration) that not everybody owns. These requirements and the required intense re-training prevent opening its usage to IC and FC.

Ops Time-Waste	The workload of the people in the Delivery and Operations Team is extremely high. They spend an incredible amount of their time repeating the same instructions for deploying new components because nobody else can do it. At the same time, they should focus on more important tasks such as controlling the system health and assuring high quality of the deliverables. This issue is related to problems 1 and 2, where IC and FC wait for DEL and OPS. Furthermore, the conditions above all together reflect conditions known as community smells previously seen in the literature.
Staging Isolation	Developers do not use the automation tools for builds and deployments to the INT environment. This environment should be used for integration tests, possibly automated tests, to detect early detection problems in the release process or the deployment itself.
Manual Pre-deploy	Some components are being deployed manually, pushing the artifacts to the right environment. This action is prone to human error and time-consuming for the people in charge of deploying new releases. Also, the components that are deployed manually are not tracked in the Registry tool; therefore, no information on their deployments exists.
Build-TimeIn visibility	No information on build time is stored, not allowing for improving the tool itself or the rest of the pipeline based on the telemetry about the tool performances.
Unaccountabl e Builds	The access to the build server is shared among all the people. Therefore, there is no control over who has started previous builds and deploys.
Test Lintering	TESTERS (IC, FC) requested an automated system for doing simple integration tests at every release to save time and early detect issues such as missing configuration, wrong naming conventions, and other blocking problems.

4.4 Lessons learned and future work

According to interviews and final focus groups with the project stakeholders and teams, the solution we identified and started to implement concerns creating an automated infrastructure supported by the managed DevOps experience we gathered in applying RADON to support the work of the DevOps team. On the one hand, from a more qualitative perspective, we should enact content, root-cause, and SWOT analysis in a data-driven fashion to further elaborate on each challenge. This implies that RADON practitioners embarking on DevOps migration/adoption exercises should individually determine how every organisational and socio-technical challenge is born, evolves during the adoption, and whether eventually that challenge is reduced, solved, or even made worse. On the other hand, from a more quantitative and research perspective, we should cross data collected before and after introducing new capabilities designed to address the challenges above.

This research should bear the target of understanding whether the introduced changes had improved (or made worse) the delivery process performances and the software organizational structure around that process. The general purpose of both research streams should be to provide practices and patterns that elaborate further on the identified challenges and auto-mated ways to quantify and manage the impact of each challenge measurably.

5. DevOps Community (Anti-)Patterns on the Road to RADON: Process and Product (Re-)Design Principles

In the scope of the RADON adoption, we experimented [DeStefano2021] with open-source projects either closely-related to the RADON technical space---e.g., providing baselines for RADON adoption such as orchestration technologies, service-oriented or event-driven middleware---or who expressed the necessity or intention to adopt or support micro-grained and cloud-native architectural styles which are featured at the core of the RADON IDE and framework, namely, the microservices and Function-as-a-Service design paradigms. Within this experimentation, our research goal was discovering the extent to which known software community anti-patterns---a.k.a., *community types and community smells* [Tamburri2015,Tamburri2018]---have an effect on the adoption of RADON technologies around the aforementioned core design paradigms. This section channels the findings behind this study and the consequent distillation of three software process and product (re-)design principles for direct practical application.

5.1 Software community structure types and smells explained

Software engineering projects are now more than ever a community effort. In the recent past, researchers have shown that their success depends on source code quality and other aspects like the balance of power distance, culture, global engineering practices, and more; all these dimensions reflect the original objectives behind the RADON methodology as well as the research and innovation goals set for T2.4 for which this deliverable is intended. In such a scenario, understanding the characteristics of the community around a project and foreseeing possible problems may be the key to developing successful systems. Over the last few years, researchers have been investigating the impact of such complex social networks on the sustainability of open- and closed-source communities and source code quality, finding them to be a highly relevant factor for the success of software systems, most especially those falling in the order of RADON. For example, the so-called *socio-technical congruence* [Cataldo2008] affects distributed systems' build success [Cataldo2008], while Palomba et al. [Palomba2018] found that the so-called *community smells* negatively influence the presence and intensity of code smells and churn.

Furthermore, In recent work, Tamburri et al. [Tamburri2018] elicited a set of *community patterns*---namely a set of recurrent characteristics describing the community structure emerging from a software process evolution---which influence the general characteristics of software products and their lifecycles.

On the one hand, each type reflects one of four meta-types, namely: (a) communities, which are social constructs made for sharing (e.g., of values, norms, practices.); (b) networks, which suggest the presence of digital or technological support tools to account for (physical, cultural or otherwise) distance of some kind; (c) groups, which are tightly knit sets of people or agencies that pursue an

organizational goal; (d) teams, which emerge as specifically assembled sets of people with a diversified and complementary set of skills.

On the other hand, community types manifest themselves into a pattern, made up of two or more types---such types are known to influence the efficient adoption of specific emerging technologies such as FaaS. We start exploratively focusing on community patterns as manifestations of multiple types at once but, to give concrete inputs to the research and practitioner communities, also discuss the influence of each type over the observed characteristics.

From a deeper structural and evolutive perspective, community structures around software develop *community smells,* namely, patterns of organizational activity that reflect sub-optimal conditions. As another crucial part of our study context, community smells represent one of the possible manifestations emerging in connection to specific patterns. As such, manifestations are established already in software engineering literature (e.g., see Palomba et al. [Palomba2018]).

5.2 Experimental setup

By gathering data from 25 open-source communities, we exploited association rule learning to discover frequent co-occurrences between the two phenomena of interest, and then reasoning on the rationale behind the observed relations. Key findings of our study show that different community patterns relate to different smells, highlighting that the governance mechanisms which are put in place may potentially have consequences in terms of social debt. More specifically, we analyze the chosen systems relying on various third-party tools—namely *Yoshi* and *CodeFace4Smells* (that we have already employed in a previous study) to detect community patterns and smells, respectively, while CLOC, Multi-metrics Grimoire-Lab for extracting product and process metrics. To account for an exploratory perspective, we adopt a statistical hypothesis-testing approach to pinpoint the relations (if any) among community patterns, software processes, and product metrics in our sample. We find that specific community patterns relate to equally specific organizational conditions; at the same time, we reveal that the current understanding of software community structures and their implicit/explicit adoption of community patterns is still limited and deserves further attention.

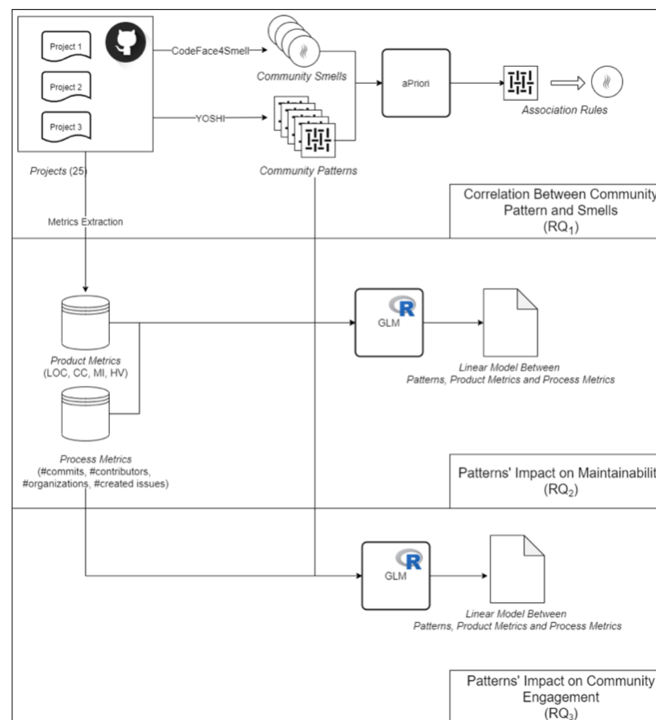

Figure 5.2.1: Workflow adopted for the empirical study

Theoretical Setup and Research Questions. The overall research design and methodology are recapped in Figure 5.2.1. The *goal* of the study recapped in this section was to investigate the relations of community patterns with community smells and their effect on both a software system's maintainability and community engagement, with the *purpose* of understanding whether the structural organization of a community may potentially affect other community-, product-or process-related factors, from the *perspective* of both researchers and practitioners, who are interested in discovering the potential impact that a community structure has on the entire project. Our study uses community patterns as the cause construct and analyzes their effects on some project aspects that we suspect may be influenced. On the one hand, given its exploratory nature, our study uses metrics and constructs that we conjecture may be influenced by the community dimensions under investigation. On the other hand, in this section, we provide an overview of the research questions and conjectures driving our study and present some relevant information on the dataset employed to address them. Although it should be noted that the selection of the constructs to examine reflects the exploratory nature of our study, we recognize the need for future work invested incoherently and systematically identify and analyze additional metrics from both the process and product front. More concretely, we addressed the following Research Questions:

- RQ1. What is the relation between community patterns and community smells?
- RQ2. To what extent do community patterns affect the maintainability of a system?
- RQ3. To what extent do community patterns affect the engagement of a community?

Tool Support and Dataset Overview. We considered 25 open-source software communities (see Figure 5.2.2) from the GitHub software community management platform, sampled according to guidelines specified at the beginning of this section. In addition, from the initial list of 81,327,803 open-source projects available, we first excluded systems having less than 10 contributors: such a filter was required to gather projects built by an actual community of developers. Then, we excluded systems with less than 100 commits—since they represent the most basic organizational manifestation of any software community—to rely on a sufficient amount of information to study how developers collaborate. This aspect is required to accurately detect community smells, as explained later in the paper. Applying these filters, we came up with a total of 44,387,266 projects. Starting from this very large number of projects, we set out additional filters to restrict the size of the sample: we considered projects having at least (i) 1000 stars, (ii) 500 commits, and (iii) 50 contributors. Applying these additional filters, we obtained 3558 projects. Due to computational constraints, we randomly selected 25 of them, which we used to run the previous study (i.e., RQ1) and all well within the scope of RADON. However, during the investigations, 4 out of 25 GitHub repositories were removed by their authors, preventing us from extracting the product and process metrics needed to answer these two research questions. We decided not to replace these projects and to continue the study restricting to the remaining 21. Table 5.2.1 summarizes the main characteristics of the extracted software projects.

Name	#COMMITS	#CONTRIBUTORS	Main Lang.	Domain
Android*	314790	759	Java	Library
Arduino	6596	262	Java	Eletronics prototype platform
BoilerPlate*	469	48	JS	Web libraries and frameworks
Bootstrap	16665	1064	JS	Web libraries and frameworks
Boto	7282	682	Python	Web libraries and frameworks
Bundler	39236	761	Ruby	Web libraries and frameworks
Cloud9	9160	82	JS	Application Software
Composer	7409	774	PHP	Software Tools
Cucumber	600	23	Java	Software Tools
Ember.Js	17594	868	JS	Web libraries and frameworks
Gollum	1970	186	Ruby	Non-Web libraries and frameworks
Hammer.Js	1282	101	JS	Web libraries and frameworks
Hawkthorne	5568	82	Lua	Software Tools
Heroku*	353	48	Ruby	Software Tools
Modernizr	2412	260	JS	Non-Web libraries and frameworks
Mongoid	6932	497	Ruby	Non-Web libraries and frameworks
Monodroid*	1462	61	PHP	Non-Web libraries and frameworks
Netty	13308	419	Java	Software Tools
PDF.Js	9735	307	JS	Web libraries and frameworks
Refinery	14003	542	Ruby	Software Tools
Salt	86637	2599	Python	Software Tools
Scikit-Learn	23110	1146	Python	Non-Web libraries and frameworks
Scrapy	7063	298	Python	Non-Web libraries and frameworks
SimpleCV	2649	108	Python	Non-Web libraries and frameworks
SocketRocket	537	81	Objective-C	Non-Web libraries and frameworks

Table 5.2.1: Software projects analyzed in the empirical study

In the context of our study, we focused on some particular instances of smells:

- **Black Cloud.** This smell arises when the community presents an information overload due to a lack of structured communications or cooperative governance.

- **Bottleneck.** In this case, one member interposes herself into every formal interaction across two or more sub-communities with little or no flexibility to introduce other parallel channels.
- **Organizational Silo.** This smell appears when there are "siloed" areas of the developer community that do not communicate, except through one or two of their respective members.
- **Lone Wolf.** Instances of this smell arise when the developer community has unsanctioned or defiant contributors carrying out their work with little consideration of their peers, decisions, and/or communication.

We detected These smells by exploiting an automated tool named *CodeFace4Smells* [Tamburri2018], a fork of CodeFace, a tool originally designed to extract coordination and communication graphs mapping the developer's relations within a community. CodeFace4Smells augments these graphs with detection rules able to identify the four community smells taken into account. As an example, the identification pattern for Lone Wolf is based on the detection of development collaborations between two community members that have intermittent communication counterparts or feature communication using an external "intruder", i.e., not involved in the collaboration. Also for this tool, it is important to comment on its accuracy. CodeFace4Smells has been empirically evaluated using surveys and/or semi-structured interviews with original industrial and open-source practitioners belonging to 60 communities. In particular, the authors of the tool showed practitioners the results obtained when running CodeFace4Smells on their communities, asking for confirmation. As an outcome, they all reported the validity and usefulness of the tool without pointing out additional problematic situations that occurred in their communities. In other words, according to developers, the community smells output by the tool are all true positives; as for false negatives, if they exist, these were not pointed out by original developers. The results achieved by the tool in previous studies make us confident of the high reliability of the tool and its suitability for our study. To address RQ1 and evaluate the relationship between community patterns and smells, we performed a three-step data collection and analysis: (1) identification of community patterns, (2) identification of community smells, and (3) association rule discovery.

The goal of the two research questions is to analyze the impact that community patterns—along with other product and process characteristics—have (i) on the maintainability of a system (RQ2) and (ii) on the community engagement in terms of contributions and issues created in the project's repository (RQ3). For RQ2, we measured maintainability through Maintainability Index (MI), an aggregate of traditional source code metrics, to indicate the degree of maintainability of a given system. It can capture different maintainability aspects (i.e., size, volume, and complexity) regardless of the programming paradigm—as we considered systems written in different languages. Since its original formulation by Oman and Hagemeister, there have been many variants of such a metric, all of them combining the Lines Of Code (LOC), the McCabe's Cyclomatic Complexity (CC), and one of the Halstead metrics (i.e., Volume Or Effort) into a polynomial equation. The

investigation of RQ3 focused on two different aspects explaining the engagement in open-source communities, i.e., the number of contributions and the created issues. We used, respectively, the number of commits (i.e., #commits) and the number of created issues (i.e.,#created_issues) until 30th April 2018 (inclusive), just as the investigation of RQ1.

5.3 Results of the experiments

RQ1. What is the relation between community patterns and community smells? Table 5.3.1 reports the association rules, grouped by community pattern, extracted after applying the *aPriori* algorithm. This section also provides some qualitative analysis of the association rules aimed at further investigating the results and possibly understanding whether the relation between community patterns and smells is causal. However, to this aim, we deeply analyzed only a subset of the systems contained in our dataset. Specifically, we focused on the two projects, namely Arduino and SaltStack (Salt in our sample). A first consideration is related to the fact that, when not filtering association rules by support and confidence, we found relationships between smells and all the communities identified by Yoshi except FormalNetworks (FNs).

Rule	Support	Confidence	Lift	p-value
Formal Group → Bottleneck	0.77	0.91	1.54	0.033
Formal Group → Lone Wolf	0.73	0.88	1.26	0.013
Informal Community → Organisational Silo	0.74	0.94	1.69	0.011
Informal Community → Lone Wolf	0.68	0.82	1.63	0.022
Informal Network → Lone Wolf	0.71	0.85	1.46	0.024
Informal Network → Black Cloud	0.69	0.83	1.55	0.043
Network of Practice → Bottleneck	0.76	0.89	1.59	0.032

Table 5.3.1: Association rules grouped by community pattern

Likely, this is due to the rigorousness used to select members in this community type: indeed, only certified and acknowledged developers can become members of these types of communities, which typically operate under strict regulatory contribution policies and codes of conduct As a consequence, developers within the community need to follow strict code of conducts to continue contributing, and this is known to address several known organizational issues, but also to manifest unexpected ones, such as higher turnover.

Finally, we found an unexpected connection between networks of Practice (NoPs) and the Bottleneck smell. By Definition, a network of practice is a community that connects communities of practice,i.e., collocated groups in which interactions are frequent and collaborative. While Such communities should theoretically be effective in communication, they are often affected by a Bottleneck due to developers who act as middlemen between two groups. For example, in version v2015.5.0 of the Salt project, a single developer managed most of the communications performed by developers, becoming de facto a bottleneck. Interestingly, the lift value reached 1.59, with a p-value equal to 0.032, confirming the relationship's significance. To broaden the scope of the discussion, the results achieved show that different community patterns are more prone to be

affected by different community smells. Therefore, practitioners can exploit the information extracted by Yoshi about the community pattern as a useful source to diagnose and understand underlying social, socio-technical, and organizational issues across their community in sight of adopting RADON complex and emerging technologies such as microservices and serverless architectures.

RQ2. To what extent do community patterns affect the maintainability of a system? The main results of the multicollinearity analysis are reported in Table 5.3.2. The models were built in a progressive manner to make sure the explanatory power of different factors in a step-by-step fashion. In particular, three models were defined for each of the three analyses: (i) the first only consider-ing the effects of the presence of community patterns, (ii)the second considering both the patterns and the selected product-related metrics, and (iii) a full model that involves both product and process metrics as confounding factors. To sum up, the model's regressors consist of 6 independent variables (i.e., the six different community patterns) and 4 confounding factors, namely LOC (only for RQ2), CC, #contributors and #organizations. The following table reports the results for our second research question. The table describes the three models we created,i.e., (i) Patterns, (ii) Patterns + Product and (iii) Full, to see the differences in terms of predictors significance.

	Patterns			Patterns + Product			Full		
Term	**Estimate**	**Std. Err.**	**Signif.**	**Estimate**	**Std. Err.**	**Signif.**	**Estimate**	**Std. Err.**	**Signif.**
(Intercept)	57.02	14.94	**	69.26	10.93	***	75.66	10.47	***
FG	−4.25	7.81		−7.50	5.56		−7.59	5.69	
FN	−8.18	10.34		−1.13	7.49		−2.05	6.62	
IC	18.51	9.00	.	11.07	6.61		6.20	6.16	
IN	−7.77	11.21		−4.26	7.86		−4.46	7.06	
NOP	7.89	10.36		5.63	7.25		3.38	6.96	
WG	25.25	9.50	*	15.80	7.13	*	15.91	6.36	*
CC				−0.43	0.14	**	−0.41	0.14	*
LOC				−0.00	0.00		0.00	0.00	
Contributors							−0.01	0.00	.
Organizations							−0.02	0.26	

Significance codes: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$, '.' $p < 0.1$

Table 5.3.2: Results of the multicollinearity analysis for RQ2

The first and more important result is the significance of the predictor WG—corresponding to the presence/absence of a community pattern featuring a Workgroup—in each of the three models. Looking at the estimated coefficient (i.e., Estimate column), we can see a positive correlation with the dependent variable MI, implying that a system developed by a community that shows the characteristics of a Workgroup (WG=1) tends to have higher MI values, and so a better degree of maintainability. This relation could be explained by the definition of the Workgroups community pattern itself. Indeed, this pattern is characterized by a strong cohesion among the team members who are all focused on a specific business area. This aspect could imply that developers collaborate a lot along the lines of adopting and maintaining technologies within the RADON technical space.

Such a degree of collaboration leads to more attention to software maintainability since different developers often make changes on the same source files.

RQ3. To what extent do community patterns affect the engagement of a community? Table 5.3.3 offers detailed results. Considering the first set of models, it is possible to appreciate that FN and IN have a high statistical significance in all the three computed models, meaning that they have a strong influence over the intensity of commits. This result is also quite understandable, taking into account the nature of the considered community patterns. On the one hand, Formal Networks (FN) are very formal communities, and by their nature, contributions (commits in this case) must follow a strict protocol to be accepted. On the other hand, Informal Networks (IN) do not have such rigid rules, so it is potentially easier to contribute to a project. In both cases, either positively or negatively, these patterns highly influence how much people contribute to a project.

	Patterns			Patterns + Product			Full		
Term	**Estimate**	**Std. Err.**	**Signif.**	**Estimate**	**Std. Err.**	**Signif.**	**Estimate**	**Std. Err.**	**Signif.**
(Intercept)	1.18	0.24	***	1.08	0.24	***	1.14	0.26	***
FG	−0.08	0.12		−0.05	0.12		−0.09	0.14	
FN	−0.80	0.16	***	−0.85	0.16	***	−0.82	0.17	***
IC	−0.15	0.14		−0.09	0.14		−0.10	0.16	
IN	−0.80	0.18	***	−0.83	0.17	***	−0.80	0.18	**
NOP	−0.13	0.16		−0.11	0.16		−0.17	0.18	
WG	−0.09	0.15		−0.01	0.15		−0.02	0.16	
CC				0.00	0.00		0.00	0.00	
Contributors							0.00	0.00	
Organizations							−0.00	0.01	

Significance codes: ‘***’$p < 0.001$, ‘**’ $p < 0.01$, ‘*’ $p < 0.05$, ‘.’ $p < 0.1$

Table 5.3.3: results of the multicollinearity analysis for RQ3

These results allow us to reject Hn2(FN) and Hn2(IN) null hypotheses in favor of the alternative hypotheses Ha2(FN) (“the community pattern FN has an impact on the number of commits”) and Ha2(IN) (“the community pattern IN has an impact on the number of commits”).

Unfortunately, none of the other patterns even became significant over the three iterations; thus, we could not reject (either confirm) the other null hypotheses, i.e., Hn2(p), where p $\in$ {IC, FG, NOP, WG}. More work needs to be enacted in this research direction and possibly more evidence from a more varied selection of open-source projects to obtain a more definitive result. When considering, however, the issues’ intensity as a dependent variable, we cannot find important conclusions regarding community patterns. Indeed, looking at the results, we can see that none of the models report any statistical significance for the community patterns.

Although these results do not allow us to reject any of the second set of *null* hypotheses Hn3(p), they suggest a good starting point for further research, aimed to investigate the mid/long term effects of community patterns on the quality of the product and the engagement of the community especially in terms of communities supportive of technologies well within the RADON technical space.

5.4 RADON adoption principles

In the scope of RADON adoption, the following principles can be distilled from our results:

Principle 1. *Different community patterns relate to different community smells. The reasons behind the presence of smells are strongly related to the characteristics and peculiarities of the community patterns.*

Therefore, adopting RADON means conducting an organizational rewiring exercise that needs to be driven by investigating previously existing organizational structures to understand whether they can cope with the adoption and continuity with an even more organizationally-intensive technical space, i.e., the RADON IDE. Likewise, depending on previously present organizational forms, the following principle applies:

Principle 2. *The presence of community patterns featuring Workgroups (WG) has a positive impact on maintainability, meaning that projects maintained by WG-like communities tend to have higher maintainability in their respective codebases.*

Therefore, teams working with RADON or engaged in its adoption are recommended to adopt the Workgroups organizational pattern, as detailed in previous work [Tamburri 2013]. Finally, for more organizationally lasque communities, the following finding applies:

Principle 3. *IN and FN community patterns have shown to have a significant impact over the inten-sity of commits made to a project. However, nocommunity pattern showed a significant impact on the intensity of created issues.*

This result could be due to the fact that we considered a limited time window and issues (introduced using a certain organization structure) could have been created later. Our conclusion is that other investigations with additional data are required to drive the organisational reinforcement using agile tools and approaches such as RADON in the mix.

6. Adopting RADON for Hybrid Computing: the Joint RADON-SODALITE Use Case Report

The present section provides a joint report[51], mutually developed by the RADON and SODALITE Horizon 2020 projects, that engaged a customization and mutual asset adoption experience. As such, this provides a concrete example that demonstrates the challenges and solutions found in adopting RADON assets in situations that fell outside the scope of the project requirements.

In February 2021, the two consortia mutually agreed with a MoU to establish a RADON-SODALITE (R-S) task force with the following core objectives:

- Definition of an integrated scenario demonstrating the ability to deploy in the SODALITE runtime environment an application modelled with the RADON framework IDE.
- Verification of correct execution of the deploying application in enabling an action in response to a cloud-generated event, such as an upload by an end user of this application.
- Verification of the ability of the application to execute in a hybrid HPC-cloud scenario in a scenario involving data transfer.

The joint task force membership consisted of all the authors of this article. The group has run bi-weekly meetings to synchronize their technical activities. A joint Github organization was also established to coordinate the releases of integrated assets:

https://github.com/RADON-SODALITE

This document reports on these technical activities and the joint experience, commenting on achievement of joint work and mutual feedback on the tools that were cross-validated as part of these activities. A joint webinar broadcasted on social media to expose the outcomes of this collaboration is planned for June 2021, coordinated by the TOSCA Technical Committee, so that the results can be exposed to a broad set of stakeholders interested in open source orchestration technologies.

6.1 Weather News front-end case study

The main aim of the integrated scenario was to demonstrate the ability to extend and customize the R-S frameworks in a complex end-to-end scenario falling outside the requirements of either project. It was observed that, on the one hand, RADON work emphasized FaaS technology and SODALITE similarly emphasized HPC, while on the other hand RADON did not consider HPC and similarly SODALITE did not focus on FaaS. This offered an opportunity for both projects to consider a joint scenario that posed the question on how to support deployment over heterogeneous cloud targets.

[51] Authors (R=RADON, S=SODALITE): Luciano Baresi (S), Giuliano Casale (R), Pelle Jakovits (R), Dragan Radolović (S), Damian A. Tamburri (R & S), Kamil Tokmakov (S), Shreshth Tuli (R), Vladimir Yussupov (R), Michael Wurster (R).

The overall technical architecture of the use case is shown in Figure 6.1.1. The goal of the use case was to be able to design and deploy this end-to-end application, called the Weather News Use Case. Weather News has the objective of demonstrating an image processing and visualization pipeline stretching across a "continuum cloud", including HPC, raspberry PI devices, and a backend public cloud with FaaS support.

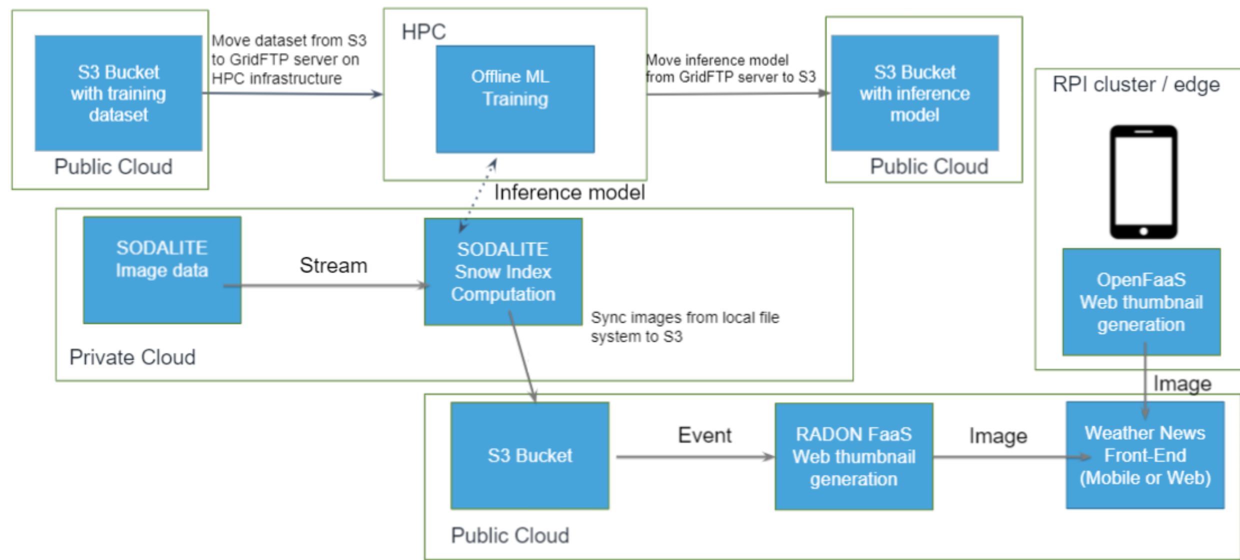

Figure 6.1.1: R-S Weather News case study consisting of HPC, Cloud and edge deployment layers

A set of technical requirements was drafted to describe the planned work:

- R-SR-01: Move snow index mask and values from executing VM to remote S3 bucket
- R-SR-02: Move snow index mask and values from executing VM to remote Google Cloud storage
- R-SR-03: Define S3 thumbnail generation FaaS pipeline
- R-SR-04: Define Google Cloud thumbnail generation FaaS pipeline
- R-SR-05: Create news front-end website
- R-SR-06: Training data pipeline between cloud storage and HPC
- R-SR-07: Develop node templates for MinIO storage buckets and OpenFaaS functions.
- R-SR-08: Develop OpenFaaS function triggers.

We now discuss individually the components of the Weather News application across the different deployment environments.

6.1.1 HPC layer

This section of the Weather News application is an adaptation of the Snow Use Case (Snow UC) developed within SODALITE as one of its use cases. The original Snow UC goal is to exploit the operational value of information derived from public web media content, specifically from mountain images contributed by users and existing webcams, to support environmental decision

making in a snow-dominated context. At a technical level, Snow UC consists of an automatic system that crawls geo-located images from heterogeneous sources at scale, checks the presence of mountains in each photo and extracts a snow mask from the portion of the image denoting a mountain. Two main image sources are used: touristic webcams in the Alpine area and geo-tagged user-generated mountain photos in Flickr in a 300 x 160 km Alpine region. Figure 6.1.2 shows the different components of the original Snow UC pipeline. Ultimately, the pipeline consists of three sub-pipelines, where certain image processing steps are applied. As such, there are two sub-pipelines that filter and classify source images (user generated photos and webcam images) and a computational sub-pipeline that at the end computes a snow index based on the processed source images. All the components are deployed on VMs of the private OpenStack cloud.

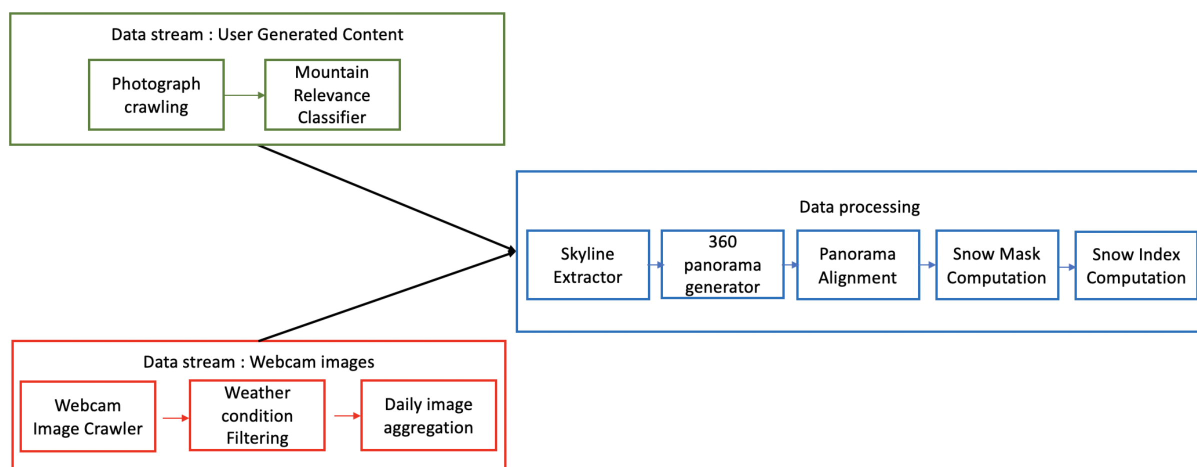

Figure 6.1.2: Components of the Snow Use Case pipeline

The task force leveraged Snow UC to implement the Weather News case study. The original Snow UC pipeline was extended with the offline ML-training of one of the components of Snow UC (Skyline Extractor) and with a thumbnail generation for the resulting images of Snow UC, as presented in Figure 6.1.2. These thumbnails are then eventually consumed by Weather News Front-End, which displays images to the end user. The ML-training demonstrates deployment on an HPC cluster with the GPU accelerators. The training dataset is stored in an Amazon S3 bucket and then moved from the bucket to the HPC infrastructure. Once the ML-training is executed, the resulting inference model is moved from the HPC cluster into another S3 bucket, from where it can be further consumed by the Snow UC Skyline Extraction component.

6.1.2 Cloud and edge layers

The cloud layer consists of a hybrid setup, with some services hosted on OpenStack in a private cloud and some other services, in particular S3 buckets and functions, hosted on Amazon AWS.

The images made available on the S3 bucket can then be picked up by FaaS functions for further processing, which in the Weather News Case study focuses on thumbnail generation.

The thumbnail generation is implemented in two cases, both relying on Function-as-a-Service (FaaS) technology. In the first case, the thumbnails are generated in the Cloud with Amazon Lambda service and stored in Amazon S3. In this classic FaaS application scenario, a function hosted on AWS Lambda generates thumbnails for each image uploaded to a source AWS S3 bucket. More precisely, an image upload event triggers the function, which generates a thumbnail for the uploaded image and stores it in the target AWS S3 bucket.

In the second case, the same function is deployed on Edge (a Raspberry Pi cluster) with OpenFaaS and the results are stored in MinIO (an S3-compatible storage). With similar APIs of both MinIO and S3, the pipeline is integrated with Android smartphone gateway. The weather photos can be uploaded by an android application using access and secret keys. Similar to geo-tagged mountain photos, images with geo-tagging information can be uploaded within the app to apply the UC pipeline. Basic image filtration and processing like image-subsampling can be performed at the edge to minimize the response time of the workloads.

6.2 Tool baselines

The task force involved in the development of the use case the following tools and assets from the two projects:

- RADON Graphical Modelling Tool (GMT)[52]
- RADON Particles[53]
- RADON Data Pipelines[54]
- xOpera orchestration (both projects)[55]
- SODALITE IaC Modules[56]
- SODALITE Snow Use Case (Image data, Snow Index Computation)[57]

A number of limitations where found at start for the baseline tools that presented obstacles towards meeting the requirements:

- The two projects developed different TOSCA modelling profiles, both based on TOSCA 1.3, and encoded in the RADON Particles and SODALITE IaC Modules, tailored to the different deployment environments target. The feasibility of their integration had to be established through initial tests and be followed by integration of the TOSCA service templates of the two projects in a unified hierarchy.

---

[52] https://github.com/radon-h2020/radon-gmt
[53] https://github.com/radon-h2020/radon-particles
[54] https://github.com/radon-h2020/radon-datapipeline-plugin
[55] https://github.com/xlab-si/xopera-opera
[56] https://github.com/SODALITE-EU/iac-modules
[57] https://www.sodalite.eu/sites/default/files/sodalite/public/content-files/articles/D6_2-Initial_implementation_and_evaluation_of_the_SODALITE_platform_and_use_cases.pdf section 4.1: POLIMI Snow UC.

- The SODALITE IDE uses a TOSCA-like model definition language supported by ontology based semantic modelling which is translated to TOSCA blueprints as an intermediate execution language to provision resources, deploy and configure applications in heterogeneous environments. Supporting the RADON node type definition in SODALITE IDE was a cross project validation of approaches used in both projects. At the same time, while RADON GMT can address the need to have a graphical TOSCA node and template representation by design, it could not show at start SODALITE node templates as part of the visualization.

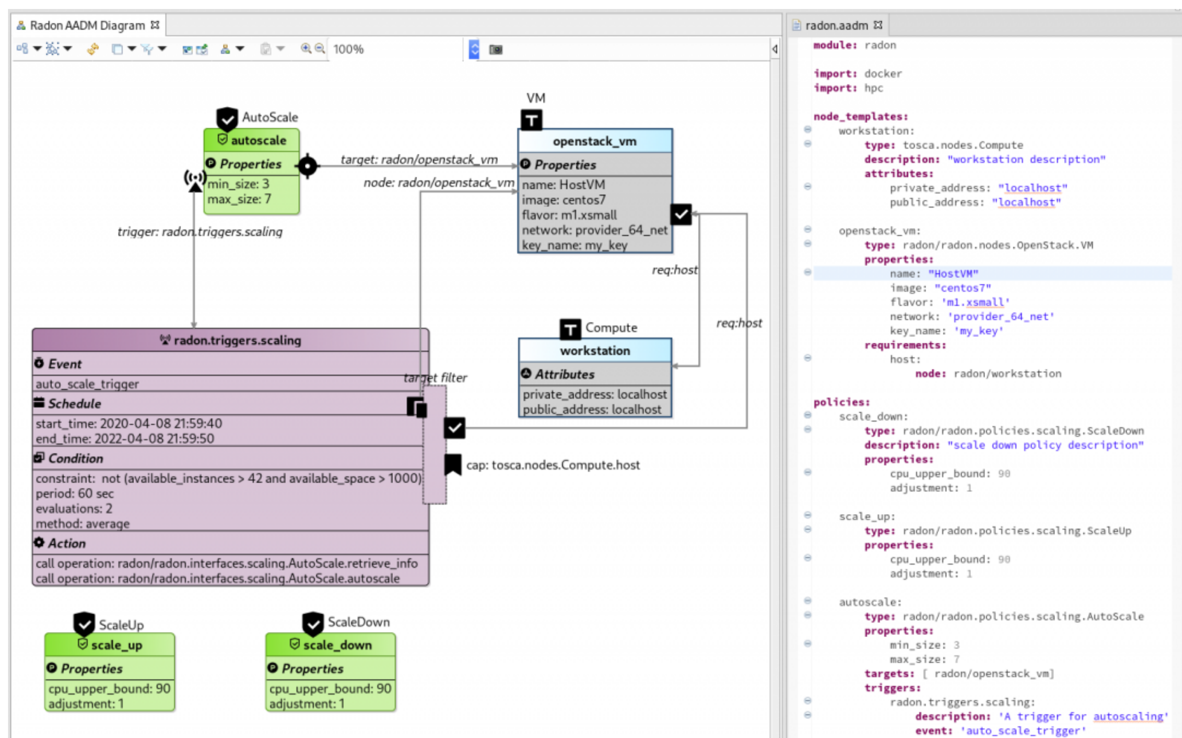

Figure 6.2.1: SODALITE IDE showing a sample RADON defined node type

- The RADON Modeling Profile had neither abstract nor concrete node types to support HPC, as this was out of the scope of the project. As it was decided to integrate the SODALITE types into the RADON Particles repository, this required to introduce a completely new hierarchy of node types, resulting in a corresponding palette of components available for modeling in the RADON GMT.
- The RADON Data pipelines offered a method to move data using TOSCA and Apache NiFi. However, the solution was not designed to operate in HPC context, where tools such as GridFTP are required to transfer data out of HPC nodes.
- RADON has developed a specialized set of raspberry PI node templates used within the Assisted Living UC, however these were used internally to the industrial use case owner and not available for repeatable use as part of the RADON Particles hierarchy.

Overall, the above limitations of the baselines prompted sustained actions across the two projects to extend and integrate the baseline across multiple fronts, which were discussed and regularly reported on during the joint task force calls; the main outcomes are described in the next subsection.

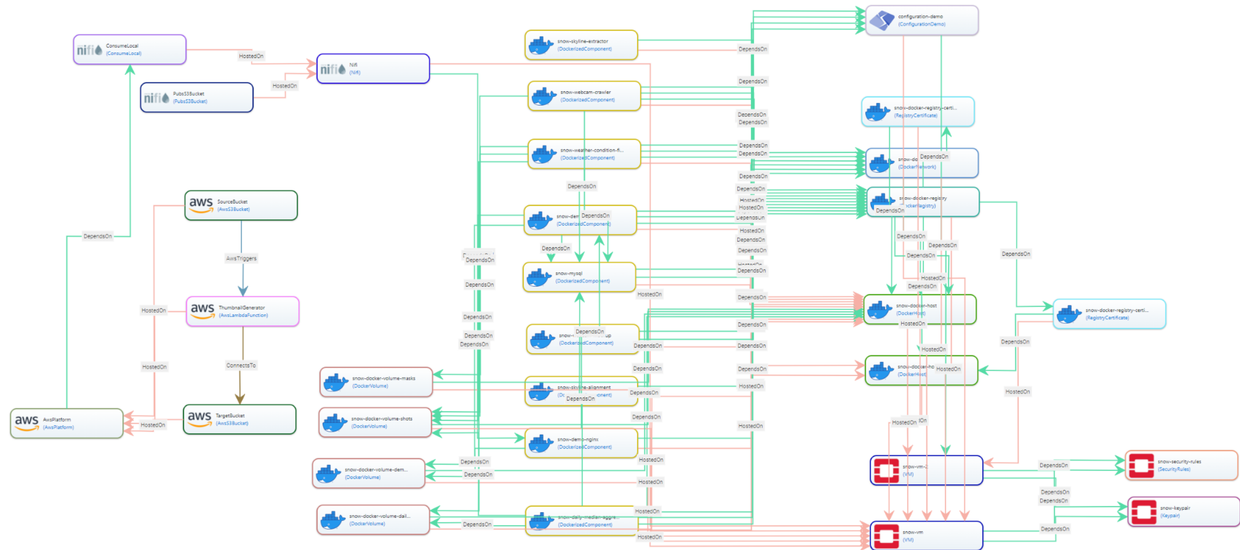

Figure 6.2.2: RADON GMT-based modeling using SODALITE node types

6.3 Technical achievements

The main technical achievements to complete the scenario are a hybrid compute profile, support for GridFTP, and support for edge deployment on Raspberry Pi devices, which are summarized below.

6.3.1 Hybrid compute profile

Since both RADON and SODALITE rely on TOSCA for modeling application topologies, most integration efforts were focused on porting the modeling constructs introduced on the SODALITE side into the format compatible with RADON tools, and RADON GMT in particular. This mainly involved modularizing introduced types and templates and organizing them according to the RADON GMT's conventions: all distinct modeling constructs such as Node Types and Relationship Types have to be stored in dedicated folders and grouped by corresponding namespaces. Furthermore, GMT employs the full TOSCA notation, meaning that modifications were needed whenever shorthands were used in the SODALITE modeling constructs. To promote reusability, GMT also requires importing all used types inside Service Templates instead of embedding these definitions directly, which also required splitting such combined models into reusable building blocks and storing them as described previously.

Another important part of the integration is organization of the deployment logic, i.e., Ansible playbooks enabling the deployment of corresponding modeling constructs. RADON GMT requires

to store these implementations also following specific file organization conventions: inside the files/{artifact_name} folder related to the corresponding modeling construct such as Node Type.

A summary of the hybrid compute profile can be found in Appendix A.

6.3.2 GridFTP support

There was a need to continuously migrate data between the HPC and Cloud layers to implement the selected use case. In most HPC systems, the main protocol for data transfer between the Grid servers and the outside world is GridFTP due to its performance, reliability and security. Data pipeline TOSCA node types have been previously developed inside the RADON project for deploying and orchestrating data migration service across Cloud and on-premise environments. However, GridFTP was not supported by both the existing TOSCA models and also by the underlying technologies (Apache NiFi and AWS).

To solve this issue a new GridFTP data pipeline node types were designed and implemented to set up real-time data migration services. While a custom approach was needed (as the underlying technology did not officially support GridFTP) using low level GridFTP Linux client libraries, it demonstrated that RADON data pipeline node types can be adapted for custom data sources with a medium amount of effort. Two GridFTP related node types were created:

1. ConsumeGridFtp - listens for new files in a specific GridFTP server folder
2. PublishGridFtp - transfers files into a specific GridFTP server folder

Both node types require GridFTP certificates to be available and must be hosted on a NiFi node type, which can be installed either on OpenStack VM, AWS EC2 VM or as a Docker container. Figure 6.3.1 illustrates a sample data pipeline, which receives data through ConsumeGridFTP and transfers files into an AWS S3 bucket using a PubsS3Bucket data pipeline node type.

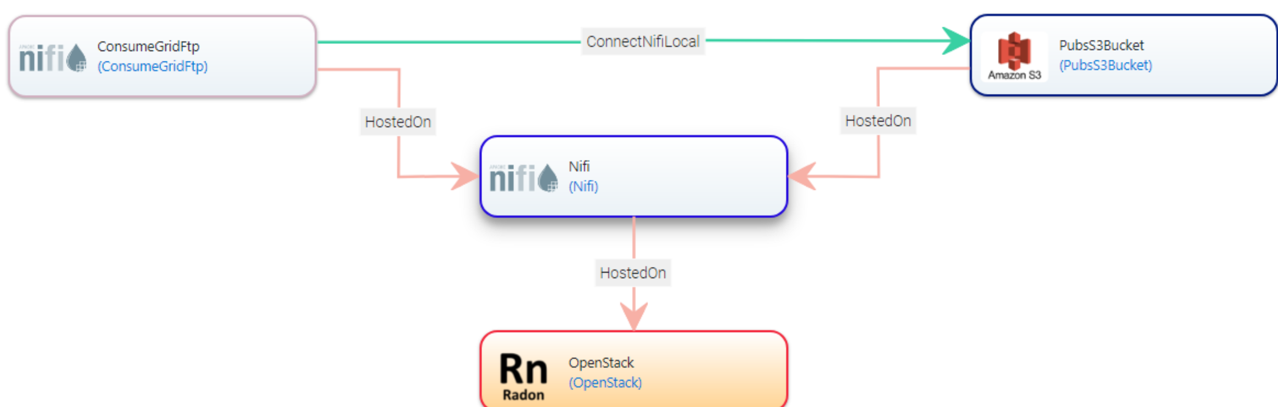

Figure 6.3.1: Example TOSCA service template which uses the new ConsumeGridFTP node type

The unified data management services, such as FTS3 and Globus, do provide a similar real-time data migration between HPC and Cloud storages, however the variety of supported storage types

and platforms is limited. Using the RADON data pipeline node types means that data can be transferred not only between the protocols such as GridFTP and S3, but also to other supported data pipeline node types (e.g., MQTT, Azure storage, Google Cloud storage, FTP) and can be further extended for support of additional storage services (e.g. Kafka, SQL databases). Furthermore, the data pipeline node types can be used to migrate data to multiple external datastores at the same time and data can be transformed or processed on the way using FaaS functions. Therefore, the developed RADON data pipelines enable data migration between heterogeneous platforms, such as HPC, Cloud storages, data streams and serverless platforms.

6.3.3 Support for edge deployment on Raspberry Pi devices

New node templates were defined to deploy serverless functions on Raspberry Pi based private edge clusters. These include node definitions for MinIO data buckets and OpenFaaS deployed on a lightweight Kubernetes (k3s) cluster. MinIO is used for creating storage buckets in a private LAN node and follows APIs similar to the Amazon S3 bucket. We also integrated this with an Android based smartphone gateway to directly upload or download images to these MinIO buckets.

We then implemented custom OpenFaaS triggers that call the serverless functions on a `s3:ObjectCreated:*` event. We assume that the docker images are already available as part of a local docker registry in the master node of the k3s cluster.

To test the node templates, we created a sample application that generates thumbnails for input images, uploaded to a MinIO `SourceBucket`. This bucket triggers the `image-resize` function, hosted on an `RPi-Platform`, that saves the output image to a `TargetBucket`. We extend an open-source Android application to create a smartphone front-end UI.

An overview of this demo application is shown in Figure 6.3.2. As visible from the figure, the RADON GMT has been extended to display customized icons for the new Raspberry Pi and MinIO data buckets.

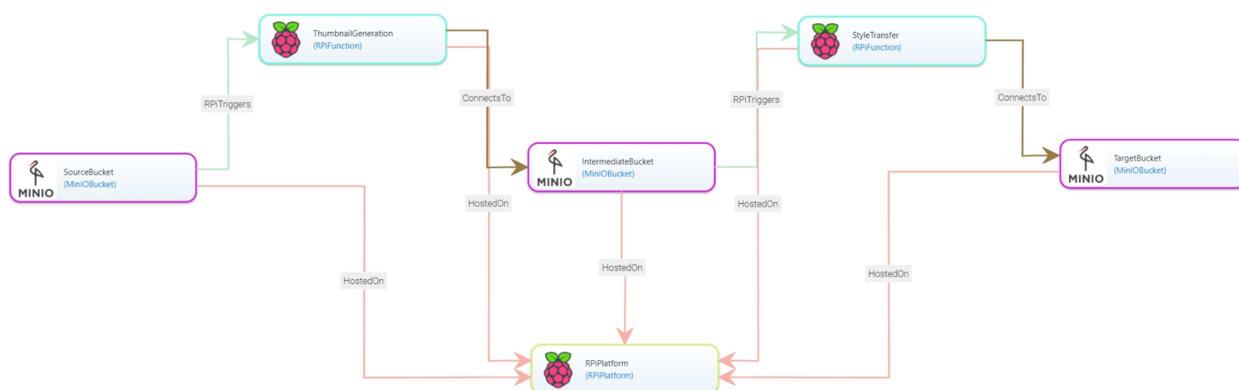

Figure 6.3.2: Edge thumbnail generation based on Raspberry Pi and MinIO buckets

6.4 Open-source release

The assets developed in this collaboration have been released and documented in the organization repositories[58]. In particular, the integrated RADON-SODALITE node templates, that we have termed the Hybrid Compute Profile, are available at:

https://github.com/RADON-SODALITE/hybrid-compute-profile

The release of the asset is mutually agreed by the two consortia to be open source under Apache 2.0 licensing, which is compatible with all the baselines.

Other original repositories included in the Github organization are as follows:

- demo-hcp-rpi: RADON based Serverless deployment on Raspberry Pi Edge Cluster;
- demo-snow-cloud: service templates for Snow demonstrator on cloud side;
- demo-snow-hpc-training: service templates for Snow HPC training.

[58] https://github.com/RADON-SODALITE/

7. Conclusions

This deliverable has introduced guidelines for adopting RADON from several perspectives: (1) in the specific context of serverless development, we outlined how RADON tools can be extended/tailored; (2) from a process perspective, we offered a study that details how software lifecycle management can be tuned to welcome the DevOps way of working intended in RADON; (3) finally, in the scope of a hybrid computing scenario which not only features serverless development but also other more classical or even more advanced cloud compute modes are featured. This document is a technical guide whose companion is the RADON booklet that provides a more narrative guide for new users. Details about the RADON booklet can be found in D7.6.

More specifically, first, in Chapter 2, we described the assumptions made by each tool and the limitations that could prevent their adoption. Given these limitations, in Chapter 3, we distilled a list of foreseeable customization and extensions that could be implemented to foster the framework adoption in the near future. Adopting RADON entails adopting a DevOps-heavy software lifecycle. Subsequently, in Chapter 4, we provide an empirical ethnographic study to understand the process of adopting a DevOps-intensive technology properly. Adopting such technology could be hindered by software community anti-patterns---a.k.a., community types and community smells. In Chapter 5, we conducted an empirical study on open-source projects closely related to RADON to verify the extent of this impact. Finally, in Chapter 6, we reported on the collaboration with H2020-Sodalite to illustrate how RADON can be smoothly extended to other contexts.

Overall, our work in RADON highlights the massive potential of serverless computing technologies and this deliverable reflects such potential with the adoptability and adaptability perspectives represented by the RADON solution. We therefore expect the contributions in this deliverable to act as a guide for all practitioners willing to embrace the serverless way both from an architectural perspective and in terms of software process management.

8. References

[Rahman2020] Rahman, A., Farhana, E., Parnin, C., & Williams, L. (2020, October). Gang of eight: a defect taxonomy for infrastructure as code scripts. In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE) (pp. 752-764). IEEE.

[Kim2006] Kim, S., Zimmermann, T., Pan, K., & James Jr, E. (2006, September). Automatic identification of bug-introducing changes. In 21st IEEE/ACM international conference on automated software engineering (ASE'06) (pp. 81-90). IEEE.

[Tamburri2015] Tamburri, D.A., Kruchten, P., Lago, P., van Vliet, H., 2015c. Social debt in software engineering: insights from industry. Springer J. Internet Services and Applications 6, 10:1–10:17.

[Cataldo2008] Cataldo, M., Herbsleb, J.D., Carley, K.M., 2008b. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity, in: ESEM ’08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineer-ing and measurement, ACM, New York, NY, USA. pp. 2–11.

[DeStefano2021] Manuel De Stefano, Fabiano Pecorelli, Damian A. Tamburri, Fabio Palomba, and Andrea De Lucia. 2020. Splicing Community Patterns and Smells: A Preliminary Study. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20). Association for Computing Machinery, New York, NY, USA, 703–710.

[Palomba2018] Palomba, F., Tamburri, D.A.A., Fontana, F.A., Oliveto, R.,Zaidman, A., Serebrenik, A., 2018b. Beyond technical aspects:How do community smells influence the intensity of code smells? IEEE Transactions on Software Engineering.

[Tamburri2018] Tamburri, D.A., Palomba, F., Serebrenik, A., Zaidman, A., 2018. Discovering community patterns in open-source: a sys-tematic approach and its evaluation. Empirical Software Engineering 24, 1369–1417.

[Tamburri2013] Tamburri, D.A., Lago, P., van Vliet, H., 2013b. Organizational Social structures for software engineering. ACM Comput. Surv .46, 3.

[Willis2000] Willis, P. & Trondman, M. (2000). Manifesto for ethnography. *Ethnography*, 1, 5--16.

[Caprarelli2019] Caprarelli, A., Nitto, E. D. & Tamburri, D. A. (2019). Fallacies and Pitfalls on the Road to DevOps: A Longitudinal Industrial Study.. In J.-M. Bruel, M. Mazzara & B. Meyer (eds.), *DEVOPS* (p./pp. 200-210), : Springer. ISBN: 978-3-030-39306-9

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

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

[D4.4] RADON Consortium, “Deliverable D4.4: RADON Models II”, 2020

[D5.6] RADON Consortium, “Deliverable D5.6: Data pipeline orchestration II”, 2020

[D7.6] RADON Consortium, “Deliverable D7.6: Community building report II”, 2021

[TPS+21] Shreshth Tuli, Shivananda Poojara, Satish Srirama, Giuliano Casale, N. Jennings. *COSCO: Container Orchestration using Co-Simulation and Gradient Based Optimization for Fog Computing Environments*, IEEE Trans. on Parallel and Distributed Systems, to appear in 2021.

Appendix A: Hybrid Compute Profile

A.1 Artifact Types

Namespace	Types
radon.artifacts	• Ansible • Repository
radon.artifacts.archive	• JAR • Zip
radon.artifacts.datapipeline	• InstallPipeline
radon.artifacts.docker	• DockerImage

A.2 Capability Types

Namespace	Types
radon.capabilities	• Invocable
radon.capabilities.container	• DockerRuntime • JavaRuntime
radon.capabilities.datapipeline	• ConnectToPipeline
radon.capabilities.kafka	• KafkaHosting • KafkaTopic
radon.capabilities.monitoring	• Monitor

A.3 Data Types

Namespace	Types
radon.datatypes	• Activity • Entry • Event • Interaction • Precedence • RandomVariable
radon.datatypes.function	• Entries
radon.datatypes.workload	• Entries

sodalite.datatypes.OpenStack	• env • SecurityRule
sodalite.datatypes.OpenStack.env	• OS
sodalite.datatypes.OpenStack.env.Token	• EGI

A.4 Node Types

Namespace	Types
radon.nodes	• SockShop • Workstation
radon.nodes.abstract	• ApiGateway • CloudPlatform • ContainerApplication • ContainerRuntime • DataPipeline • Function • ObjectStorage • Service • WebApplication • WebServer • Workload
radon.nodes.abstract.workload	• ClosedWorkload • OpenWorkload
radon.nodes.apache.kafka	• KafkaBroker • KafkaTopic
radon.nodes.apache.openwhisk	• OpenWhiskFunction • OpenWhiskPlatform
radon.nodes.aws	• AwsApiGateway • AwsDynamoDBTable • AwsLambdaFunction • AwsLambdaFunctionFromS3 • AwsPlatform • AwsRoute53 • AwsS3Bucket
radon.nodes.azure	• AzureCosmosDB • AzureFunction • AzureHttpTriggeredFunction • AzurePlatform • AzureResource • AzureResourceTriggeredFunction

	• AzureTimerTriggeredFunction
radon.nodes.datapipeline	• DestinationPB • MidwayPB • PipelineBlock • SourcePB • Standalone
radon.nodes.datapipeline.destination	• PubGCS • PublishDataEndPoint • PublishGridFtp • PublishLocal • PublishRemote • PubsAzureBlob • PubsMQTT • PubsS3Bucket • PubsSFTP
radon.nodes.datapipeline.process	• Decrypt • Encrypt • FaaSFunction • InvokeLambda • InvokeOpenFaaS • LocalAction • RemoteAction • RouteToRemote
radon.nodes.datapipeline.source	• ConsAzureBlob • ConsGCSBucket • ConsMQTT • ConsS3Bucket • ConsSFTP • ConsumeDataEndPoint • ConsumeGridFtp • ConsumeLocal • ConsumeRemote
radon.nodes.datapipeline.Standalone	• AWSCopyDynamodbToS3 • AWSCopyS3ToDynamodb • AWSCopyS3ToS3 • AWSEMRactivity • AWSShellCommand • AWSSqlActivity
radon.nodes.docker	• DockerApplication • DockerEngine
radon.nodes.google	• GoogleCloudBucket • GoogleCloudBucketTriggeredFunction

	• GoogleCloudFunction • GoogleCloudPlatform • GoogleCloudResource
radon.nodes.java	• JavaApplication • JavaRuntime
radon.nodes.mongodb	• MongoDBDatabase • MongoDBMS
radon.nodes.monitoring	• NodeExporter • PushGateway
radon.nodes.mqtt	• MosquittoBroker
radon.nodes.mysql	• MySQLDatabase • MySQLDBMS
radon.nodes.nifi	• Nifi • NiFiDocker • Pipeline
radon.nodes.nodejs	• NodeJSApplication
radon.nodes.openfaas	• OpenFaasFunction • OpenFaasPlatform
radon.nodes.testing	• AB • CTTAgent • DataPipeline • DeploymentTestAgent • JMeter • LoadTestAgent • Locust • QT
radon.nodes.VM	• EC2 • OpenStack
sodalite.nodes	• ConfigurationDemo • DockerHost • DockerizedComponent • DockerNetwork • DockerRegistry • DockerVolume • RegistryCertificate • RegistryServerCertificate • TestComponent

sodalite.nodes.OpenStack	• Keypair • SecurityRules • VM

A.5 Policy Types

Namespace	Types
radon.policies	• Performance
radon.policies.performance	• MeanResponseTime • MeanTotalResponseTime
radon.policies.scaling	• AutoScale • ScaleIn • ScaleOut
radon.policies.testing	• ABLoadTest • DataPipelineLoadTest • HttpEndpointTest • JMeterLoadTest • LoadTest • LocustLoadTest • PingTest • QTLoadTest • TcpPingTest • Test

A.6 Relationship Types

Namespace	Types
radon.relationships	• ConnectsTo • Triggers
radon.relationships.apache.kafka	• PublishToKafkaTopic
radon.relationships.apache.openwhisk	• OpenWhiskKafkaTriggers
radon.relationships.aws	• ApiGatewayTriggers • AwsTriggers
radon.relationships.azure	• AzureCosmosDBTriggers • AzureTriggers
radon.relationships.datapipeline	• ConnectNifiLocal • ConnectNifiRemote
radon.relationships.google	• GoogleTriggers

radon.relationships.monitoring	• AWSIsMonitoredBy • GCPIsMonitoredBy
radon.relationships.openfaas	• OpenFaasKafkaTriggers

A.7 Service Templates

Namespace	Types
example.org.tosca.servicetemplates	• NiFiDocker
radon.blueprints	• ServerlessToDoListAPI • ServerlessToDoListAPI_withDNS_w1-wip1 • SockShop • SockShopTestingExample • ThumbnailGeneration • ThumbnailGeneration_CDL-w1-wip1 • ThumbnailGeneration_FromS3-w1-wip1 • ThumbnailGeneration_GCP-w1-wip2
radon.blueprints.datapipeline	• ToyExampleNifi
radon.blueprints.examples	• AWS_EMR_Example • Azure_Blob_Datapipeline_Example • DataPipelineExample • EC2_on_AWS • gridFTPtoS3pipeline • MQTT_Data_Pipeline_Encryption_Decryption_Example • S3toGridFTPpipeline • SFTP_Datapipeline_Example • TestPython
radon.blueprints.monitoring	• GCP_Monitoring_Example
radon.blueprints.testing	• ABMasterOnly • DataPipelineAWSdemoSUT • DeploymentTestAgent • DeploymentTestAgentEC2 • JMeterMasterOnly • JMeterMasterOnlyEC2 • LocustMasterOnly • NiFiTIDocker • QTMasterOnly • ServerlessToDoListAPITestingExample
sodalite.blueprints	• demo-snow-cloud