H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable D4.6

Graphical Modeling Tool II

Version: 1.0

Publication Date: 31-October-2020

Deliverable Card

Deliverable	D4.6
Title:	Graphical Modeling Tool II
Editor(s):	Michael Wurster (UST), Vladimir Yussupov (UST)
Contributor(s):	Michael Wurster (UST), Vladimir Yussupov (UST)
Reviewers:	George Triantafyllou (ATC), Alexandros Ilias Spartalis (PRQ)
Type:	Report
Version:	1.0
Date:	31-October-2020
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

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

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 document is the final deliverable which presents the final architecture and its implementation of the Graphical Modeling Tool (GMT). It draws the connections with the previous deliverables and presents in detail the rationale of fulfilling the requirements. In the course of this document, we elaborate on the switch to canonical data model introduced as a substitute of the approach discussed in the first iteration of the deliverable. Furthermore, we describe which core functionalities were reengineered to support RADON use cases. This includes such features as custom data types support, export and import of CSARs, additional export mode aiming at integration with the RADON IDE, and artifacts referencing. Moreover, we elaborate on other UI and backend enhancements related to general modeling experience and integration with RADON IDE. The implementation described in this document are publicly available on GitHub[1], which will be proposed to be merged into the official Eclipse Winery[2] project. Further, the discussed modeling constructs that were presented in the RADON Models deliverables [D4.3, D4.4] are available in the RADON Particles[3] repository.

[1] https://github.com/OpenTOSCA/winery/tree/project/radon
[2] https://github.com/eclipse/winery
[3] https://github.com/radon-h2020/radon-particles

Glossary

CSAR	Cloud Service Archive
FaaS	Function as a Service
GMT	Graphical Modeling Tool
GUI	Graphical User Interface
IDE	Integrated Development Environment
MSA	Microservice Architecture
NFR	Non-Functional Requirement
TOSCA	Topology and Orchestration Specification for Cloud Applications
VM	Virtual Machine
WP	Work Package
Tw.n	Task n in Work Package w
Yn	Year n
YAML	YAML Ain't Markup Language
XML	Extensible Markup Language
CDL	Constraint Definition Language
IaC	Infrastructure as Code
CI/CD	Continuous Integration / Continuous Delivery

Table of contents

1 Introduction

The RADON project focuses on facilitating DevOps processes in software engineering which involve microservices, serverless and FaaS-based architecture elements, as well as data pipelines design, development, and operations. One part of such processes is to speed-up the design of desired application deployment models and its contained components. To accomplish this task, in the context of the "RADON Models" task [D4.3, D4.4], we introduced a set of modeling constructs allowing the representation of application topologies using a standard cloud modeling language, namely, OASIS TOSCA. These modeling constructs extend the existing types of the TOSCA standard in order to model serverless FaaS and data-driven microservice applications. To support modelers with means to use these modeling constructs in an interactive manner, a Graphical Modeling Tool (GMT) is introduced. In this document, we elaborate on the final version of GMT developed in the scope of WP4. This deliverable presents the final architecture and its implementation details by extending the open source TOSCA modeling tool Eclipse Winery. The described software in this deliverable is open source and available online on GitHub[4].

1.1 Deliverable Objectives

This deliverable has the main objective to provide the final architecture and implementation to support the vision and use cases of RADON. Further, we address integration tasks by preparing essential GMT components to be easily integrated into Eclipse Che or with other RADON tools. Lastly, we provide an update on established practices and quality gates guiding us through the software engineering processes to guarantee a certain quality level of the resulting code base.

1.2 Overview of Main Achievements

The main achievements of this final deliverable can be broken down into the following parts:

1. Reengineer the intermediate solution introduced in the first iteration of the deliverable to use a canonical model, thus decoupling XML- and YAML-based specifications and making the code more readable and maintainable.
2. Enable support of custom data types in GMT, e.g., definition of composite properties with user-provided schema. This is one of the core requirements needed by some of the RADON tools.
3. Provide support for export and import of executable CSARs by extending the existing XML-centric functionalities.
4. Enable referencing of deployment artifacts attached to RADON Models and ensure the self-containment of CSARs by extending the export and import functionalities.

[4] https://github.com/OpenTOSCA/winery/tree/project/radon

5. Provide support for features required in the context of integration with the RADON IDE by extending the corresponding UI and backend components.

1.3 Structure of the Document

The current report is structured as follows: **Section 1** provides an overview of this deliverable by presenting the overall objective. **Section 2** gives an overview of requirements guiding this deliverable and presents updates after M22. **Section 3** elaborates on development activities for the final GMT architecture, while **Section 4** addresses activities in the context of integrating GMT with other tools. Moreover, **Section 5** presents the software engineering practices employed during developing the GMT and, finally, **Section 6** concludes the report and reflects on the requirements by discussing their level of fulfillment.

2 Requirements

The initial version of RADON requirements was provided in the deliverable D2.1 “Initial Requirements and baselines” [D2.1], whereas the deliverable D2.2 “Final Requirements” [D2.2] finalizes RADON requirements according to the emerging challenges arising over the course of the project. This section outlines the final RADON requirements relevant for the GMT.

The full description and rationale of each requirement is available on GitHub[5].

Table 1. RADON requirements relevant to the GMT.

ID	Category	Title	Description
R-T4.3-1	Must have	Integration into IDE	The graphical modeling tool (GMT, Winery) must provide a GUI which is integrated into the IDE.
R-T4.3-2	Must have	Navigation to business logic	In the GMT, it must be possible to navigate to the respective workspace where the source code is maintained.
R-T4.3-3	Should have	Navigation to deployment logic	In the GMT, it should be possible to navigate to a workspace where the deployment logic is located.
R-T4.3-4	Should have	Definition of constraints	In the GMT, a user must be able to define a set of constraints (based on the CDL) for one or more components.
R-T4.3-5	Could have	Import existing models	A user could be able to import existing models that can then be reused when creating new ones.
R-T4.3-6	Should have	Integration with other RADON tools	A user should be able to trigger certain tools from the modeling tool.
R-T4.3-7	Should have	Present verification result	Given a RADON model which does not comply with a set of hard constraints, the graphical modelling tool should be able to graphically represent the explanation generated by the verification tool.
R-T4.3-8	Must have	Test case specification	In the GMT, it must be able to use predefined or to create new test case specifications that a user can use to annotate modeled components.

[5] https://github.com/radon-h2020/radon-gmt/issues

R-T4.3-9	Must have	Modeling lots of elements	In the GMT, it must be possible to model an amount of up to two hundred elements (i.e., nodes, relations).
R-T4.3-10	Could have	Group modeling elements	The GMT could provide a feature to group or abstract certain elements in order to reduce the visual complexity of tens or hundreds of FaaS components.
R-T4.3-11	Must have	Referencing files as URLs	There must be the possibility to reference business logic as a URL to make it possible to change the business logic of the application without changing the RADON Model.
R-T4.3-12	Should have	Manage company-specific RADON Models and TOSCA entity type	GMT must be initialized and start based on the RADON Particles, but company-specific RADON Models as well as custom TOSCA entity types must be stored differently.
R-T4.3-13	Must have	Save exported CSAR to the filesystem	Save generated CSARs to Eclipse Che's workspace to enable further processing by other RADON tools.
R-T4.3-14	Could have	Manage RADON Models from RADON Template Publishing Service	Push and load RADON Models to or from the RADON Template Publishing Service once users decide to finalize a model or want to modify an existing one.
R-T4.4-1	Must have	Export executable deployment model	The bundle which is exported from the modeling tool must be processeable by the RADON orchestrator.
R-T4.4-2	Could have	Export of different deployment model formats	The GMT could provide an option to export a blueprint in different formats to use other orchestration tools, such as OpenTOSCA or Terraform.
R-T4.4-3	Could have	Import of model in different format	The GMT could provide the possibility to import different output formats produced by the integrated RADON tools.

2.1 Requirement Updates After M22

This section gives an overview and its rationale of requirements that have changed since the deliverable D2.2 “Final Requirements” [D2.2] at M22. Based on lessons learned and a better technical understanding of the envisioned user workflows the consortium agreed to change the following requirements:

R-T4.3-4: Definition of constraints

Change: Reduced priority from MUST to SHOULD

Rationale: The consortium agreed to define the actual constraints based on the CDL directly in the IDE. Therefore, GMT will offer the possibility to open the IDE and navigate to the respective TOSCA file of a selected RADON blueprint (aka. TOSCA Service Template). Then, users can utilize the capabilities of the IDE to create a corresponding CDL file and directly define the CDL statements. Hereby, the GMT employs a plugin inside Eclipse Che in order to navigate to the specific folder inside the project explorer based on a passed URL parameter.

R-T4.3-14: Manage RADON Models from RADON Template Publishing Service

Change: Reduced priority from MUST to COULD

Rationale: The consortium agreed to integrate the RADON Template Library Publishing Service (TLS) by employing an Eclipse Che plugin to query and publish RADON types and models directly from the IDE. We agreed that it makes more sense to use GMT only as an editor and the IDE to commit, publish, and update types and models with the TLS. This follows a similar workflow as in traditional software engineering where users employ text editors to write code and external tools, such as Git, to commit and push the changes to share it with others.

3 Engineering the final RADON GMT Architecture

In this section we elaborate on the RADON GMT architecture refinements introduced in the scope of RADON toolchain evolution. The main parts include fundamental changes in the Eclipse Winery's underlying data model and the core functionalities needed for RADON GMT in the context of TOSCA YAML-based specification support and integration with the RADON toolchain.

3.1 Supporting All TOSCA Specifications: A Canonical Data Model

In the former GMT deliverable [D4.5] we presented a first step to process TOSCA modeling files according to the latest TOSCA YAML standard. In this context, we introduced two separate domain models, one for TOSCA XML and another one for TOSCA YAML, while the TOSCA XML domain model is used by all internal components as well as the RESTful HTTP API. Therefore, a *Model Converter* component was introduced being able to map between the TOSCA XML and TOSCA YAML domain whenever the YAML-based repository implementation tries to read or write TOSCA entities from the file-based datastore. Despite the fact that a major achievement of the former deliverable was to fully reuse the existing TOSCA XML domain model in all backend components and as an interface to the frontend, it also introduced some drawbacks.

The biggest drawback is that the TOSCA XML domain model has been polluted with elements and aspects that should not belong to it. Due to the fact that there is no 1:1 mapping for both of the standards it was necessary to build-in specific aspects of the TOSCA YAML domain model, e.g., the simplified declaration of TOSCA artifacts. As a result, there was no longer a clear separation of concern between these two standards in the code base, which makes it more difficult to understand the connections and mappings between the standards and decreased the overall maintainability.

For the aforementioned reasons we introduced a *Canonical Data Model* (CDM) to represent both standards in a common and unified manner. With a CDM we can clearly separate the specifics of each standard and provide a common definition to all internal GMT components. Figure 1 shows the resulting component architecture. In the context of RADON, the GMT is started by using the "YAML Repository" strategy (introduced in the first deliverable [D4.5]) to process TOSCA entities according to the latest TOSCA YAML standard. Internally, each GMT component operates based on a respectively instantiated repository interface to load and save data. Thereby, the repository interface expects and returns the respective entities of the CDM. Inside the "YAML Repository", the respective domain model of the standard is employed for reading and writing TOSCA entities into and from the file-based datastore, while it uses the improved *Model Converter* to translate between the data models.

In addition, Figure 2 shows the conversion process between the CDM and the underlying serialization model (e.g., TOSCA YAML model) based on the instantiation of the "YAML Repository" strategy.

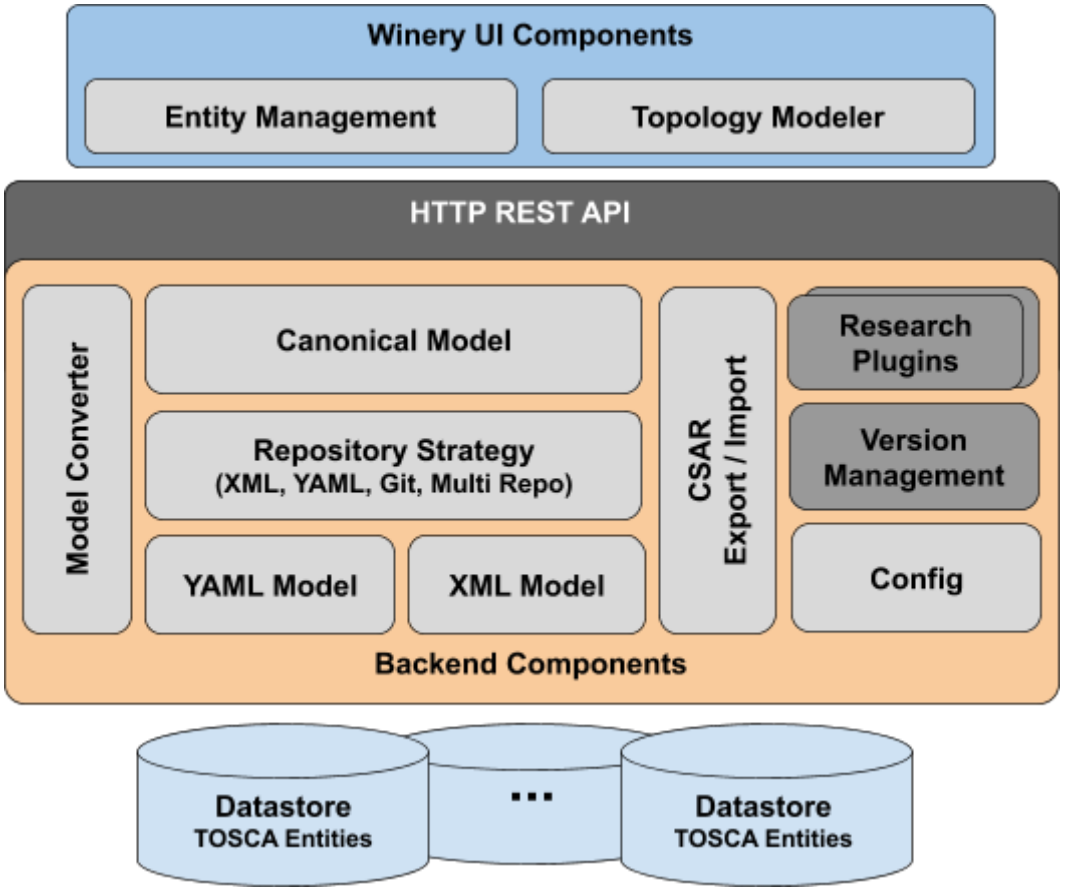

Figure 1: Final architecture for the RADON GMT.

Apart from the clear separation of the models used for serialization, it also improves the maintenance as all internal GMT components, such as the HTTP REST API, are implemented against the CDM and do not have explicit dependencies to a specific standard. While being backward compatible with the former TOSCA XML standard, the CDM also simplifies the support of future TOSCA specifications by implementing a new repository strategy and the respective converter logic.

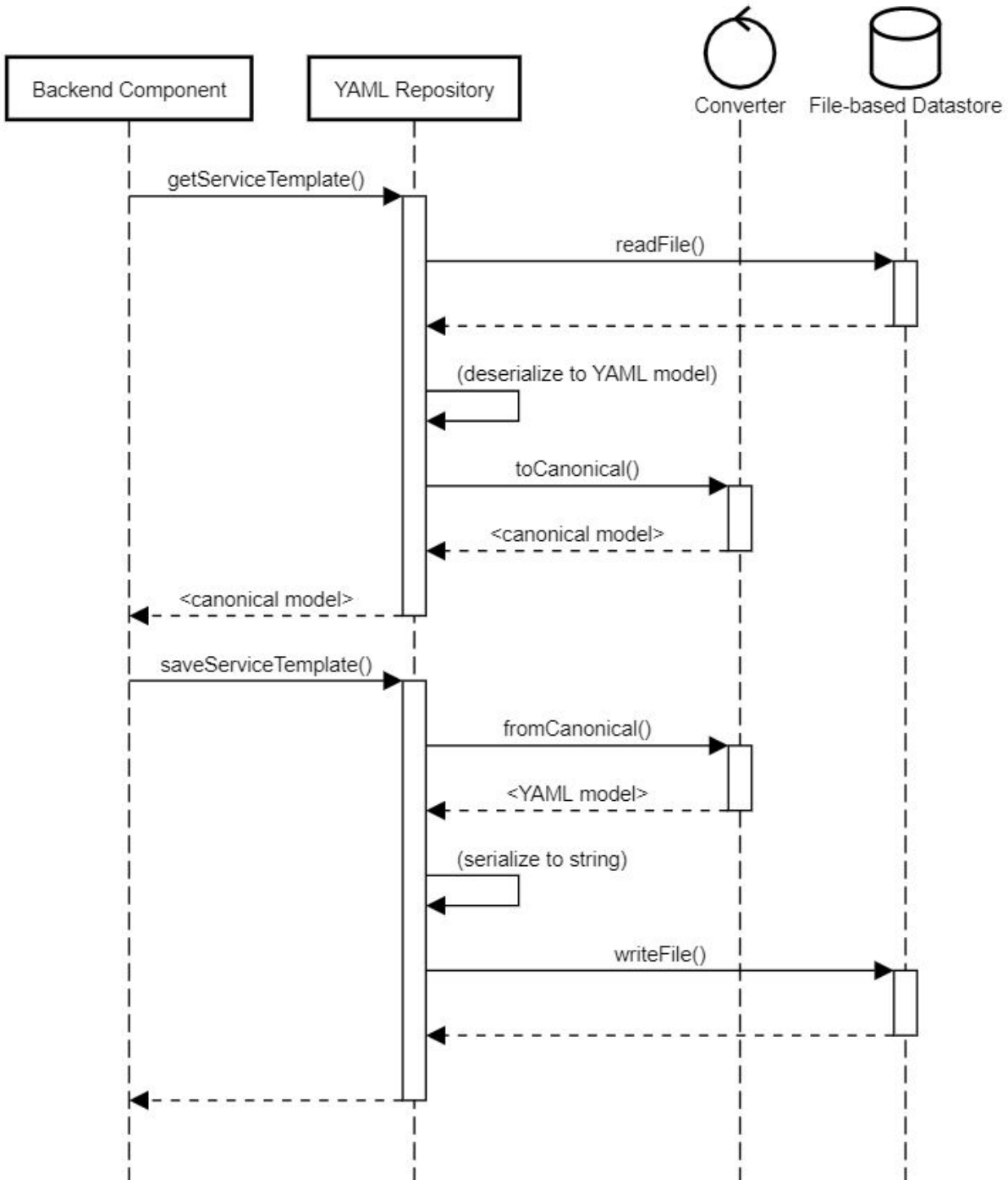

Figure 2: Conversion between domain-specific (TOSCA YAML) and canonical data model.

3.2 Core Functionalities Revised

As discussed in the previous iteration of the deliverable, several features required in the scope of RADON were not usable as-is due to the Eclipse Winery's focus on XML-based TOSCA specification. Apart from the data model issues discussed in Section 3.1, the obvious features required for working with the YAML-based specification are the export and the import of modeled TOSCA elements. Moreover, to enable modeling of arbitrary, user-defined properties in RADON GMT, the support for custom data types was needed. Finally, one feature required by several RADON use cases is to enable referencing artifacts instead of attaching them as archives directly to the model. In the following section, we describe the core points related to the implementation of the aforementioned features in RADON GMT.

3.2.1 Complex Data Types Support

One of the most important functionalities introduced in the RADON GMT after the first deliverable is the support for complex data types. Firstly, Eclipse Winery did not provide complete support for TOSCA types such as *map* or *list*, making it more complex to define properties containing collections of elements. The introduced canonical data model also resulted in modifications related to definition of properties, attributes, and inputs. Figure 3 demonstrates how a property of type map can be set using the Topology Modeler UI. In the current implementation, it is possible to supply a JSON string defining the value for a property of a given type. In addition, the user interface also provides basic validation of the inserted value against the type definition in the corresponding type, e.g., a property of the respective Node Type.

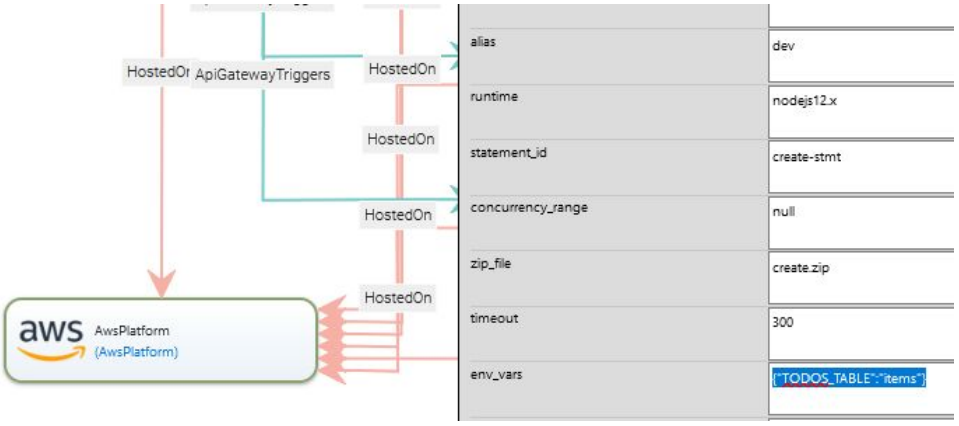

Figure 3. Definition of a property with data type "map" in the Topology Modeler UI

Another important task related to complex data types is support for user-defined Data Types allowed by the TOSCA specification. This feature facilitates specifying types with custom schemas and using such types, e.g., for properties. In fact, support for such user-defined data types was needed to enable modeling of properties required by RADON Decomposition Tool. Listing 1 shows an excerpt from the TOSCA definition of a custom data type called *Entry* in *radon.datatypes*

namespace. As can be seen, the properties follow a specific schema which combines a map of activities and a list of precedences.

```
data_types:
  radon.datatypes.Entry:
  properties:
    activities:
      type: map
      entry_schema:
        type: radon.datatypes.Activity
    precedences:
      type: list
      required: false
      entry_schema:
        type: radon.datatypes.Precedence
```

Listing 1. A custom data type radon.datatypes.Entry required by RADON Decomposition Tool.

The property of custom data types can be defined on the type level, e.g., Node or Relationship Types, using RADON GMT's Management UI as shown in Figure 4.

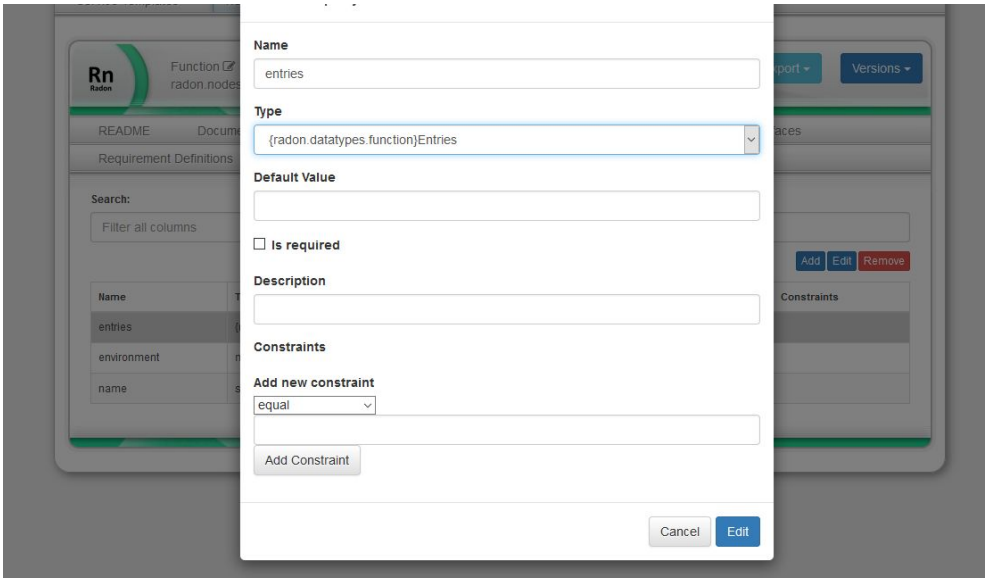

Figure 4. Definition of a property with custom data type in the Management UI

After instantiating TOSCA constructs having properties with custom data types, modelers are able to set property values by supplying a JSON string corresponding to the custom schema. At export time supplied custom values are serialized into a valid YAML representation. This decision was made to simplify the UI generation, since a large amount of nesting for custom data types might introduce unnecessary clutter, thus reducing the readability. The overall flow for working with

custom data types requires users to initially define custom data types in the corresponding tab in GMT's Management UI. Afterwards, modelers can use these data types by selecting them from the list of available data types which also includes normative TOSCA data types such as version, range, or map, as shown in Figure 5.

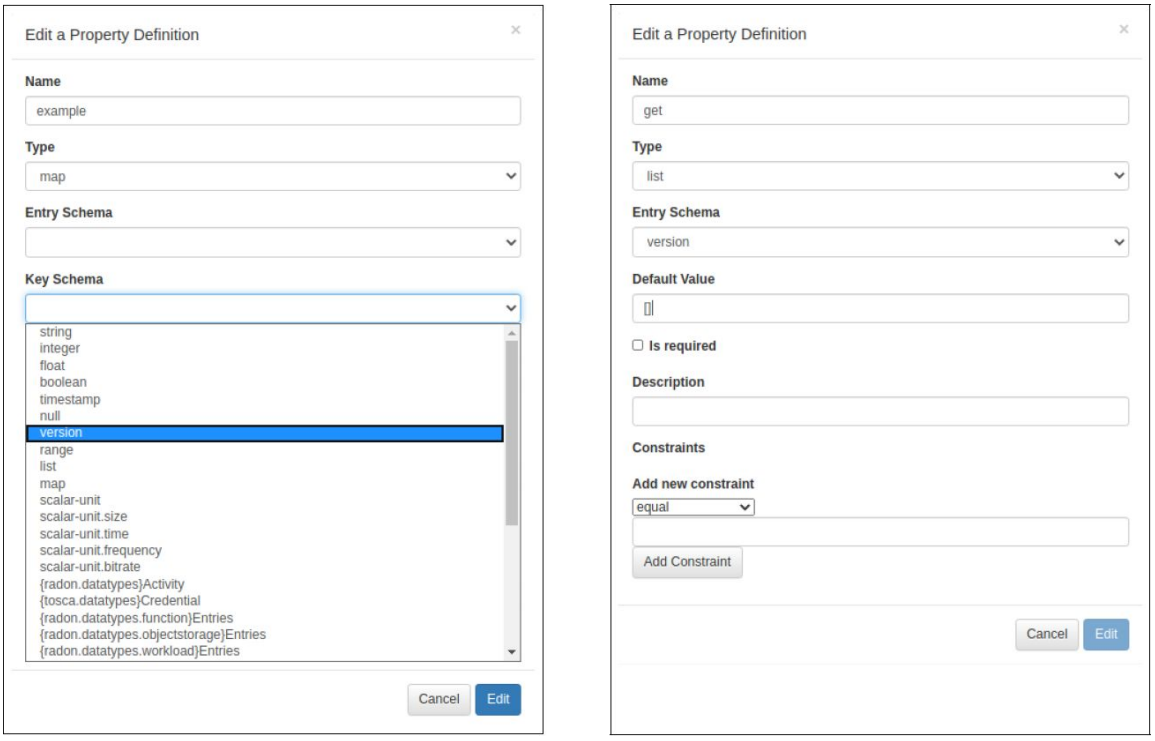

Figure 5. Definition of a property with custom data type in the Management UI

3.2.2 Export of TOSCA YAML

While the overall content of CSARs has to comply with the TOSCA standard, e.g., TOSCA.meta file that lists all included files that must be present, the actual folder structure inside CSAR can be implemented differently. For the implementation of YAML-based CSARs export, we followed the same approach used by Eclipse Winery for XML-based export. In particular, exported CSARs include several distinct types of content, namely (i) TOSCA definitions, (ii) attached artifacts, (iii) visual appearance elements, (iv) README.md, (v) LICENSE, and (vi) TOSCA.meta file. Essentially, these types are stored as follows:

- all TOSCA definitions are placed inside *_definitions* folder with CSAR,
- TOSCA.meta file is stored in *TOSCA-Metadata* folder, and
- all other files are stored in a dedicated folder representing the underlying TOSCA entity, e.g., Service Template, Node Type, etc.

To illustrate this structure in a better way, consider an example of a Service Template exported using the GMT. Figure 6 depicts the structure of ServerlessToDoAPI CSAR that represents a typical serverless use case - a serverless API, obtained from the RADON Particles repository[6]. As discussed previously, all definitions used in the model are stored in the *_definitions* folder. This includes both, explicitly used definitions as well as all referenced constructs, e.g., types with inheritance. At export time, the GMT recursively traverses all types and extracts also the parent types. This is required to export self-contained CSARs. As an additional option, it was also made possible to skip export of TOSCA normative types in case the underlying orchestrator does not require embedding such types in CSARs. The folder *TOSCA-Metadata* contains only the TOSCA.meta file, which follows the structure required by the TOSCA standard [OASIS2020] and lists all files in the CSAR. This file allows other RADON tools to easily access the CSAR's entry point, e.g., a path to the Service Template within CSAR.

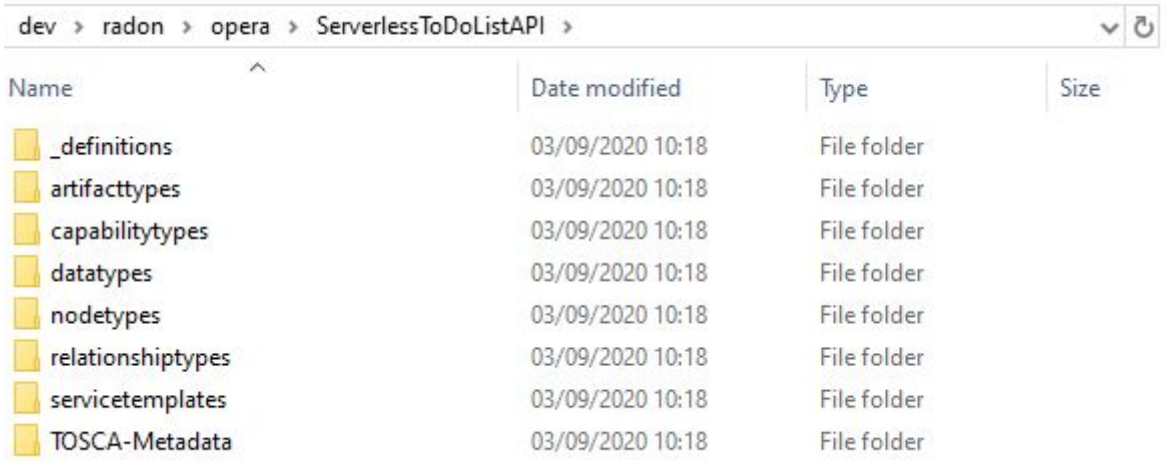

Figure 6. The general structure of CSAR for ServerlessToDoListAPI Service Template.

Next, all other files are grouped inside the folder related to the corresponding TOSCA entity such as Node Type or Relationship Type. As described in both iterations of RADON Models deliverable [D4.3, D4.4], this structure adheres to the template library's persistence rules. In the ServerlessToDoListAPI Service Template, one of the main node types is AwsLambdaFunction that defines FaaS-hosted functions hosted on AWS Lambda. Figure 7 shows the files' organization for this type within the exported CSAR: license and readme files are stored in the root of the folder representing the type, whereas all attached artifacts are stored with the *files* folder.

[6] https://github.com/radon-h2020/radon-particles/tree/master/servicetemplates/radon.blueprints/ServerlessToDoListAPI

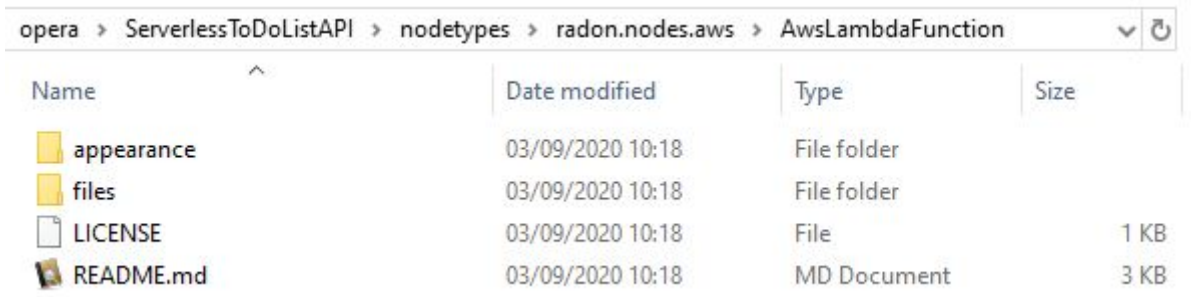

Figure 7. The file organisation for AwsLambdaFunction Node Type.

The implementation of CSAR export functionality depends on the underlying TOSCA specification and the corresponding GMT's mode, i.e., YAML mode in our case. As a result, the export functionality relies on YAML-specific classes located in *org.eclipse.winery.repository.yaml* package. In particular *YamlExporter* and *YamlToscaExportUtil* classes contain the main logic for generating the structure and serializing the respective definitions into YAML. Both of these classes extend the original, XML-centric classes *CsarExporter* and *ToscaExportUtil* respectively, and override only the YAML-specific parts. In future releases of RADON GMT, the class structure might be refactored to fully rely on the canonical model discussed in Section 3.1 and only address XML- and YAML-specific classes when it is needed.

Another important part of the implementation is the set of classes representing different entry types in a CSAR. All these classes are contained in *org.eclipse.winery.repository.export.entries* package and implement the *CsarEntry* interface which allows flexibly organizing the access to the content of a given entry, e.g., a TOSCA definition file in YAML and a binary file attached as an artifact are accessed and processed differently. This design of separation of concerns (SoC) also simplified implementation of other features such as referencing artifacts in TOSCA models which is discussed in the following section.

The original export functionality in Eclipse Winery was also available as a REST endpoint to enable programmatic access. For YAML-based CSARs we followed the same approach by extending the corresponding resources in the backend and UI elements in the frontend. Moreover, RADON GMT currently supports two export modes: the *download* and *save* modes. The *download mode* is a standard functionality which triggers the CSAR export and receives it as a response. As a result, CSARs can be downloaded either by using the UI or issuing the REST calls programmatically.

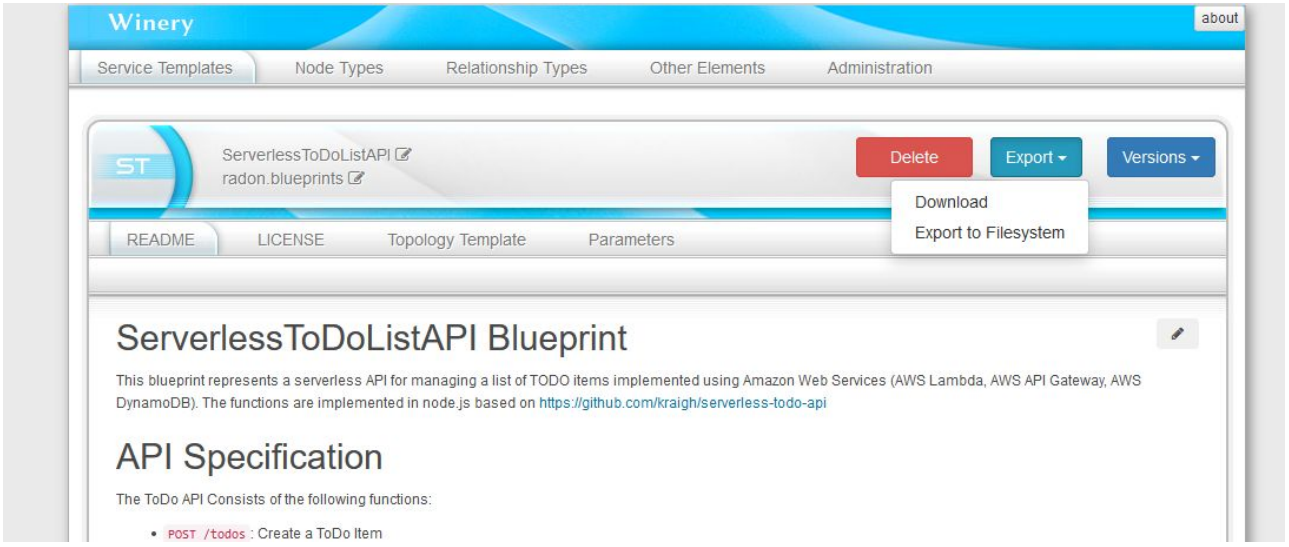

Figure 8. Two export modes using Management UI.

On the other hand, the *save mode* is introduced to facilitate using RADON GMT as part of the RADON IDE, which is based on Eclipse Che. Using this mode, the chosen model is exported as a CSAR and stored in the file system within Eclipse Che. This allows flexibly using other RADON tools within the IDE using CSARs as inputs. More information on using CSARs as the main integration point for RADON tools can be found in the deliverable "RADON Models II" [D4.4]. This functionality also required introducing modifications to Winery's REST API, meaning that this functionality can also be triggered programmatically and be used outside Eclipse Che context.

One technical aspect worth mentioning is related to persistence of artifact paths inside CSARs. Essentially, deployable Service Templates in the majority of cases require attaching deployment artifacts, e.g., function's source code for a modeled FaaS function. When modeled and persisted in RADON GMT, all such file attachments are persisted using relative paths following Eclipse Winery's repository structure. However, as discussed previously, during export all CSAR contents are stored using a slightly different structure. This requires updating all the paths for deployment artifacts *in the exported definitions* following the CSAR's folder structure. As a result, when model files, e.g., a Service Template for ServerlessToDoAPI, are executed by the RADON Orchestrator, all attached artifacts are accessible.

3.2.3 Import of TOSCA YAML

To simplify reusability and enable collaborative modeling using the RADON GMT, the import functionality of Eclipse Winery had to be extended to support import of YAML-based CSARs. Similar to export, import depends on the corresponding TOSCA specification. As a result, the YAML-specific functionalities are implemented in *YamlCsarImporter*, which extends *org.eclipse.winery.repository.importing.CsarImporter*. Similar to export functionality, import of CSARs in Eclipse Winery is achieved by means of a dedicated REST endpoint defined in *org.eclipse.winery.repository.rest.resources.MainResource*. To enable YAML-based CSARs import, this endpoint was extended to also use YamlCsarImporter class when needed. Additional modifications were also introduced in the corresponding UI elements in the Management UI. Figure 9 depicts the import dialog in RADON GMT working in YAML mode.

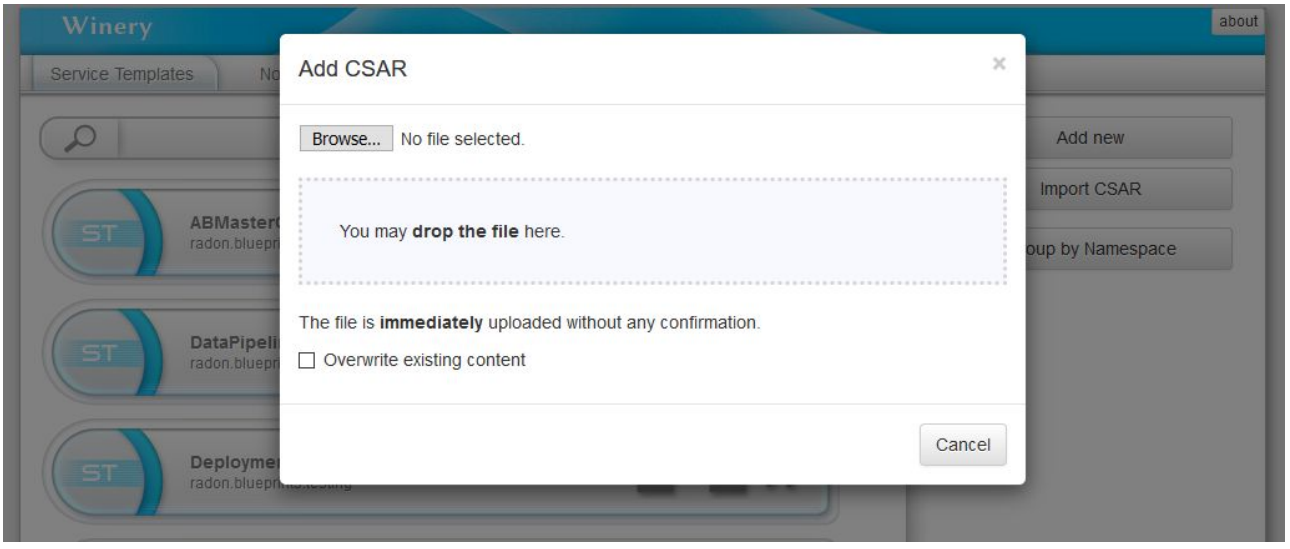

Figure 9. CSAR import using Management UI.

Since the repository layout in RADON GMT is different from the CSAR structure described in Section 3.2.2, the *YamlCsarImporter* is responsible for correct placement of all included TOSCA definitions together with the attached artifacts. Moreover, since the models in exported CSARs have modified paths for each attached deployment artifacts, during import the paths are adjusted in the imported models (Service Templates, for example) to comply with the GMT's repository layout. Additionally, it is also possible to overwrite existing models at import time, in case the corresponding option is selected. If the overwrite option is not chosen, the entities already existing in the repository are not imported. For example, Node Types such as AwsPlatform or AwsLambdaFunction that are available in the RADON GMT by default, will not be imported when importing CSARs that use these types. One pitfall here is the import of modified types with the same namespace and name. In such cases, the overwrite option can also be used. The versioning available in RADON GMT helps tackle such issues, as modified versions will be processed separately at import.

3.2.4 Artifacts Referencing

On the one hand, as mentioned previously, exported CSARs are required to be self-contained to ensure their deployability using the RADON Orchestrator or any other TOSCA-compliant orchestrator. On the other hand, artifacts attached to templates might be large in size, increasing the overall size of the resulting CSAR. To provide users with more lightweight modeling experience, one of the new updates in RADON GMT allows attaching file references as deployment artifacts. Figure 10 shows the UI modifications added to enable this artifact's attachment option.

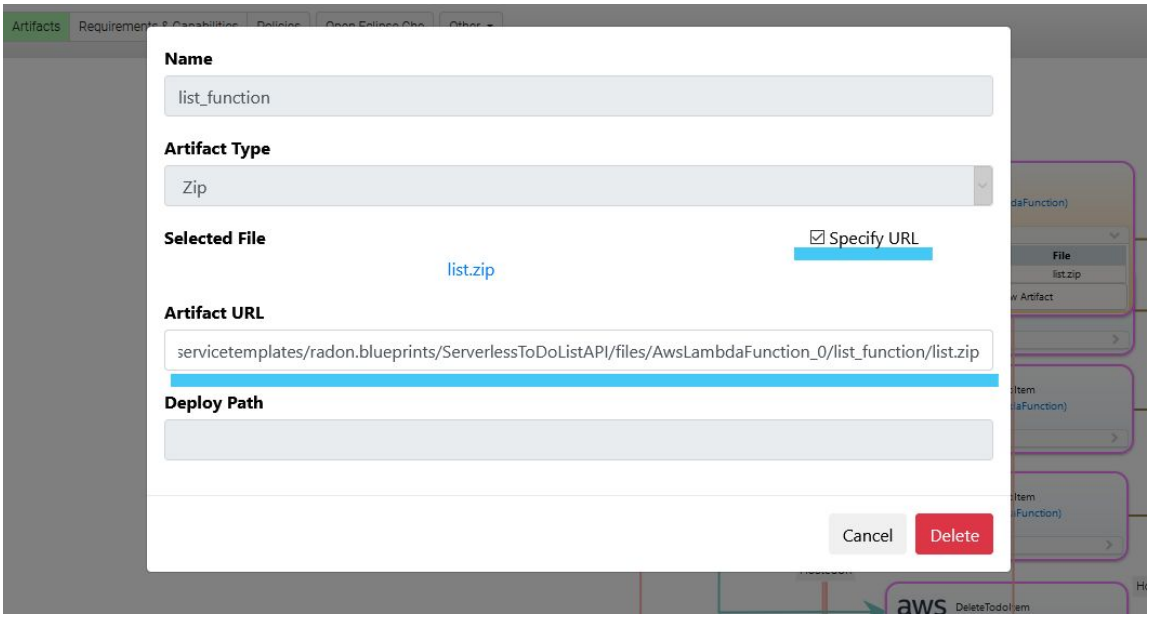

Figure 10. Artifacts referencing using Topology Modeler UI.

More specifically, the path value for referenced artifacts within Service Templates does not store a relative path within the GMT's repository, but a remote link to a downloadable file. The resulting templates can be shared among GMT instances for collaborative modeling without the need to have actual files. At export time, all referenced artifacts are downloaded by the GMT and packed within the CSAR to guarantee its self-containment. Technically, this is implemented by introducing a new CSAR entry type *RepositoryRefBasedCsarEntry* which is stored together with other entry types in the *org.eclipse.winery.repository.export.entries* package. The current assumption that RADON GMT relies on is that modelers provide valid references (also considering HTTP redirects) allowing to download files.

3.3 User Interface Enhancements

Apart from the enhancements described in the previous sections, the implementation of user feedback by the consortium, and general bug fixes, we want to highlight two additional enhancements concerning GMT's user interface: (1) a dynamic table-based data UI component and (2) the possibility to define groups of TOSCA Node Types.

Dynamic table-based data UI component. The GMT, especially in the Management UI, presents several TOSCA entities in a table-based data component. Figure 11 shows an example of the use in the context of TOSCA Property Definitions on a TOSCA Node Type. In the previous GMT version it was only possible to "add" and "remove" entries from the underlying data table. Further, for each data table a custom pop-up with the respective form elements had to be implemented individually.

We enhanced the table-based data UI component and implemented a "DynamicTable" component which is now used by several Angular components in GMT's code base. This component contains the functionality commonly used when presenting data within tables and is able to generate modal windows for "add", "edit", and "remove" actions. This is realized such that GMT developers can define metadata about the underlying data structure. For example, it is possible to define that the "name" attribute of a TOSCA Property Definition must be of type "string" and entered by standard text input form element. Based on this definition, it is possible to fully automatically generate the "add" and "edit" pop-ups and its input form elements. This enhancement eases the implementation of further data tables. Moverover, it reduced the complexity of GMT's code base as lots of boilerplate code to create pop-ups and form elements could be removed.

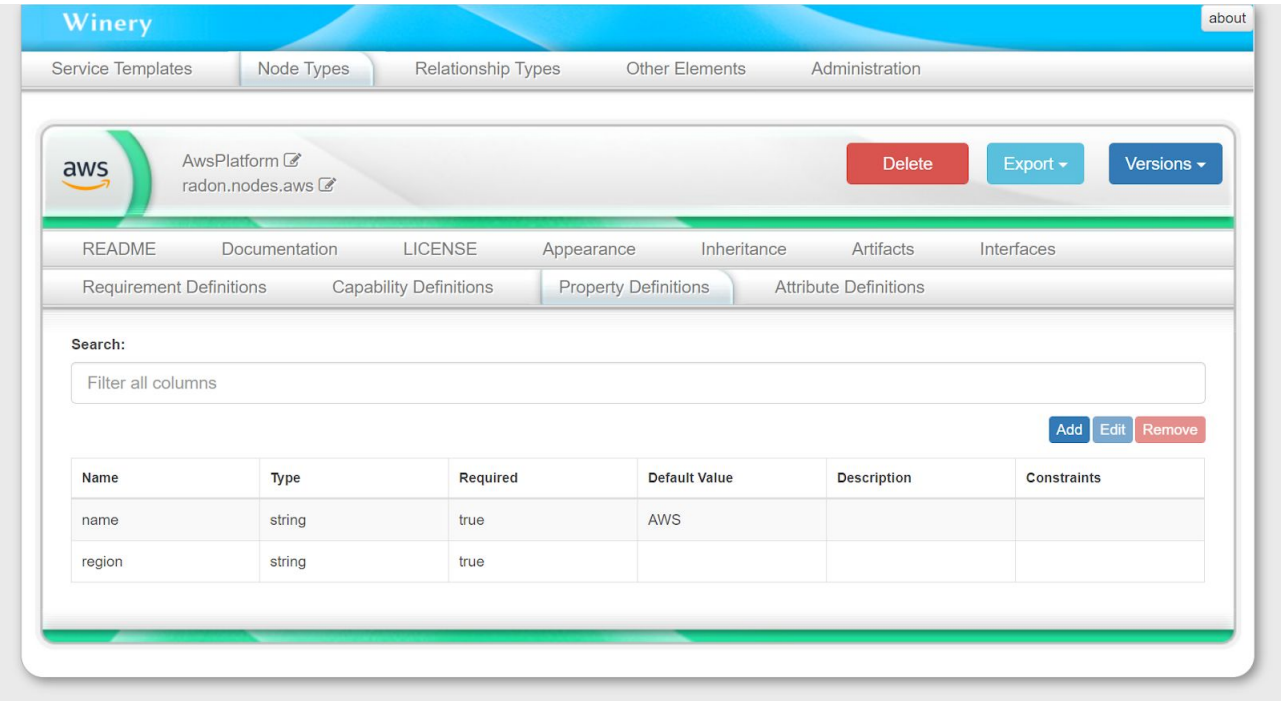

Figure 11: Dynamic table-based data UI component.

Define groups of TOSCA node types. The TOSCA YAML standard defines the ability to group node templates of a modeled topology. Therefore, to provide full TOSCA YAML support with the GMT, users must be able to define groups of TOSCA Node Types inside a TOSCA Service Template. For example, users may define a "webserver_group" in the groups section of the resulting topology template and add a "apache" node template and a "server" node template as its members. Such groups can be used to assign TOSCA policies to express that such a policy must be enforced to the complete group. For example, a use case could be to assign a scaling policy to express that the group as a whole should scale up or down under certain conditions. However, the assignment of policies is optional.

For the final version of the GMT we implemented a feature to create new groups for a given TOSCA Service Template. Hereby, users employ the Topology Modeler UI of the GMT and create such groups interactively for a modeled application. Further, the Topology Modeler UI allows to add several modeled nodes to the created groups. For example, in Figure 12 the VM node is assigned to the defined "autoscale" group. Moreover, users can assign TOSCA policies to the defined groups in order to express that such policies must be applied to all nodes of the group.

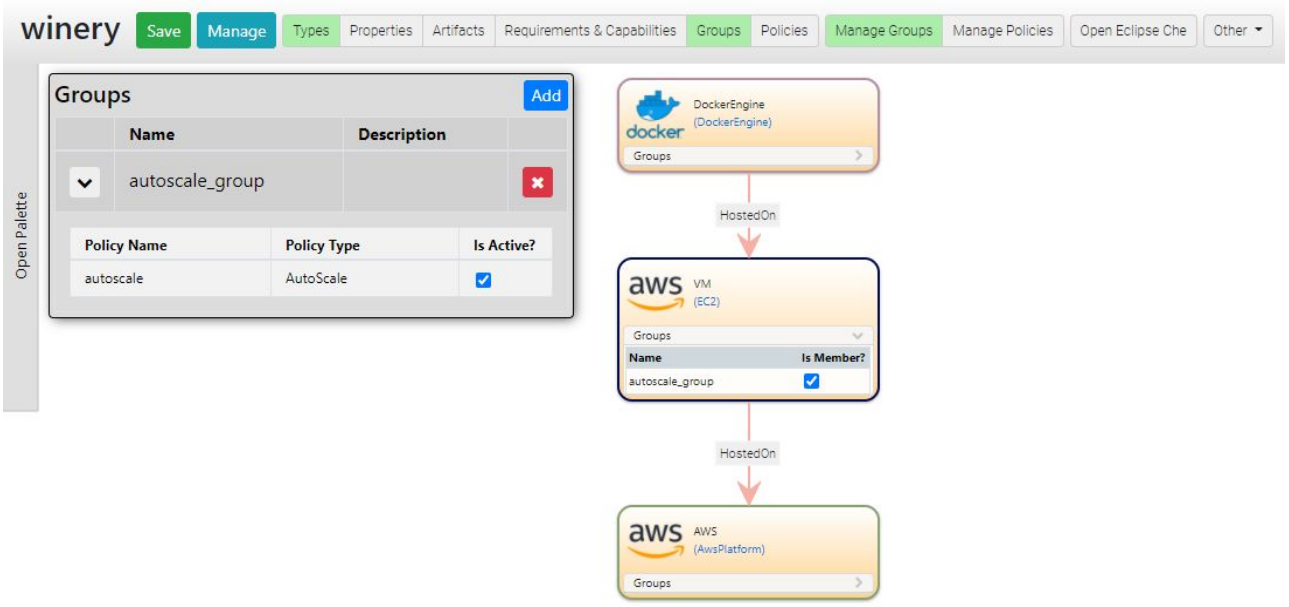

Figure 12: TOSCA group management in GMT.

4 Addressing Integration Tasks

This section highlights GMT improvements concerning the main integration activities in the context of enabling RADON GMT as a part of the RADON toolchain. This includes such aspects as modeling within RADON IDE using RADON GMT, switching between the GMT and IDE using jump to code behavior, and the interaction between the RADON GMT and other RADON tools.

4.1 TOSCA CSAR as the Point of Integration

As discussed in the deliverable RADON Models II [D4.4], CSARs can be used as the main point of integration for tools working with resulting application models. Apart from RADON Orchestrator, multiple RADON tools such as Decomposition tool or Verification tool work with TOSCA models. Since CSAR is essentially a zip archive containing at least two directories, the TOSCA-Metadata directory and the Definitions directory as described in Section 3.2.2, processing it is a straightforward task not requiring significant efforts. Essentially, the TOSCA-Metadata directory containing a TOSCA.meta file describing the contents with relative paths and archive entry points can be processed to extract the necessary information, e.g., how to access specific TOSCA Definition files such as node type definitions, policy type definitions, or the actual service template describing the structure of the application.

To simplify using CSARs not only for orchestration but also as an exchange format for RADON tools, the alternative export mode in Eclipse Winery was implemented as discussed in Section 3.2. This design decision was made so that it avoids strong coupling with the RADON tools, i.e., no tool-specific endpoints and UI elements were introduced for each tool but a standard way of accessing application models was enhanced to support more interaction scenarios.

4.2 Enhance Container-based Integration with Eclipse Che

One important aspect in this area was the enhancement of GMT's automatically built Docker image. The respective `Dockerfile` has been refactored so it does not require the usage of a `root` user inside the container. This improves security and provides the possibility to fine-tune permission for mounted volumes. Apart from that, such Docker images are appreciated by Sysadmin personnel as they reduce problems and, especially in the context of Eclipse Che, it is possible to start the GMT container under a configured Eclipse Che user with individual permissions.

Listing 2 shows an excerpt of the enhanced `Dockerfile` highlighting the main contributions in this period. First of all, we employ the recommended multi-stage build pattern to reduce the final size of the Docker image as it does not "pollute" the image with non-required build output files. Further, by the line `USER winery` we strictly define that the container runs as a non-root user

(including all the required permission settings) to easily integrate it into Eclipse Che.

```
# multi-stage build to reduce final image size
FROM maven:3-jdk-8 as builder
COPY . /tmp/winery
WORKDIR /tmp/winery
RUN mvn package -DskipTests=true -Dmaven.javadoc.skip=true -B

# build final Docker image
FROM tomcat:9-jdk8

ENV WINERY_USER_ID 1724
ENV ... # omitted for brevity

RUN apt-get ... # omitted for brevity

# copy build artifacts from builder stage
COPY --from=builder ... # omitted for brevity

# create Winery user and home dir
RUN mkdir ${WINERY_USER_HOME} \
    && groupadd -g ${WINERY_USER_ID} winery \
    && useradd -s /bin/nologin -u ${WINERY_USER_ID} -g winery -d ${WINERY_USER_HOME} \
    && chmod a+rwx ${WINERY_USER_HOME} \
    && chown winery: ${WINERY_USER_HOME} \
    && echo "winery:winery" | chpasswd \
    && usermod -aG sudo winery

# create repository dir and change ownership
... # omitted for brevity

USER winery
WORKDIR ${WINERY_USER_HOME}

EXPOSE 8080

CMD ["/docker-entrypoint.sh"]
```

Listing 2. Excerpt of the refactored Dockerfile.

4.3 “Jump to Code” Behavior

Firstly, it is worth clarifying that the term *code* in the context of RADON models might refer to several different artifact types. Obviously, the main aspect we focus on is the source code related to the deployment artifacts attached to nodes in RADON applications, e.g., function code for FaaS-hosted functions. This “jump to code” behavior is described by the (must have) requirement R-T4.3-2. On the other hand, apart from deployment artifacts, RADON applications also contain the infrastructure-as-code artifacts that make applications deployable. More specifically, each deployable TOSCA Type has Ansible code attached on the type level, which can also be modified by modelers in specific cases, e.g., when the deployment logic has to be extended. This “jump to code” behavior is described by the (should have) requirement R-T4.3-3.

The current version of jump to code behavior’s implementation focuses more on jumping to the business logic due to the priority of the corresponding requirement. Essentially, the new modifications to RADON GMT include several features that simplify switching from the artifact representations in GMT’s UI to the corresponding project folders in RADON IDE. Firstly, it is possible to switch to RADON IDE’s Project Explorer from GMT’s UI by clicking a corresponding button. Moreover, RADON GMT now supports linking source code projects from Eclipse Che, e.g., the source code of a FaaS function, to the respective node in the RADON Model, i.e., the TOSCA node template that represents the FaaS function. To achieve this, we extended GMT, besides the possibility to reference a binary artifact, with the capability to link the respective source code root folder by an URL. It is worth emphasizing that this feature is not bound only to Eclipse Che, which, however, is used as the means of validation. Essentially, any source code project that can be accessed by an URL, e.g., GitHub or GitLab repositories, can be linked this way. On a technical level, this feature was implemented by introducing a new type extending the *CsarEntry* class (discussed in Section 3.2.2) that represents linked source code location and using these entry types in Topology Modeler UI to enable opening browser tabs with the given URL. Figure 13 demonstrates how an artifact of type *Repository* added to enable jumping to the linked repository.

As the current step, we are working on the Eclipse Theia plugin prototype that is required to implement the envisioned “jump to code” behavior related also to deployment logic as discussed previously. With this additional feature, the GMT is envisioned to launch the IDE in a new browser window with an additional URL parameter containing the information to which type or template to navigate. The plugin in Theia will take this information to directly open the folder in Theia’s project/file explorer. This also integrates GMT with the CDL and VT such that modelers can easily switch to the TOSCA blueprint location where it would be possible to define new or adapt existing files containing CDL statements.

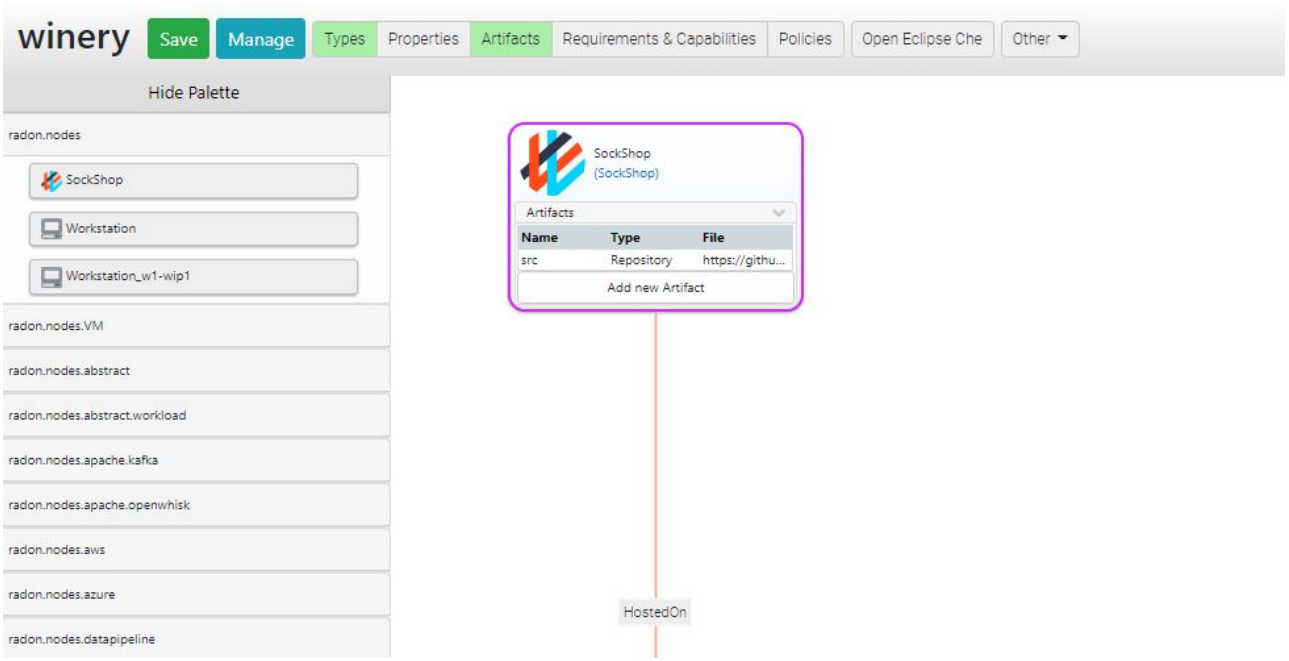

Figure 13: Using an artifact of type *Repository* to enable jump to code behavior for business logic.

5 Software engineering practices, code quality, and documentation

During the development of the GMT we established several practices and quality gates guiding us through the software engineering processes to guarantee a certain quality level of the resulting code base. In this section, we briefly describe which software engineering and code quality practices we applied. Hereby, we distinguish between practices in the context of "managing the codebase", "code reviews", "automated testing", "automated builds", and "documentation".

Managing the codebase: The GMT is developed based on Eclipse Winery, a web-based modeling environment for TOSCA application deployments. All extensions and improvements in the context of RADON must be added and merged to the official repository at the Eclipse Foundation. The official repository is publicly available at GitHub[7]. However, for the daily development we created a fork[8] of this repository where we created the branch `project/radon` based on Eclipse Winery's `master` branch. This branch is used to bring the developed features and improvements together. We created this branch since the fundamental and structural changes of the GMT were too risky to perform directly on the official `master` branch. Further, we use this branch as a central point of integrating all RADON related changes before merging them into the official `master`. The Eclipse Foundation requires that each change to the official repository is checked such that changes are not violating intellectual property, which results in slightly more administrative effort and slower merge cycles. Hence, we use the `project/radon` branch to provide the developed advancements to the RADON consortium even faster.

An additional repository[9] has been created as a mirror inside the RADON GitHub organization, which is only used to present the progress and activities in developing the features and improvements for the GMT.

Code reviews: Generally, we employ a Feature-Branch-Workflow during daily development. Developers create new branches based on the latest `project/radon` branch and propose their changes to be merged into it. All changes must be provided as pull requests to be reviewed and discussed on GitHub. Each pull request is reviewed by the GMT lead development team to check whether the changes comply with Eclipse Winery's coding standard. Further, we employ Codacy[10], a static code analysis tool, which is built into our GitHub workflow to additionally check the proposed code for errors, duplication, complexity, and style violations. Finally, the changes of the `project/radon` branch will be proposed to be merged in the context of a pull request into Eclipse Winery's `master` branch, which already took place for the changes in the context of the first GMT deliverable. Hereby, the official committer of Eclipse Winery reviews and verifies the proposed changes and then accepts the final merge.

[7] Official Eclipse Foundation repository: https://github.com/eclipse/winery
[8] Project RADON development branch: https://github.com/OpenTOSCA/winery
[9] RADON mirror: https://github.com/radon-h2020/radon-gmt
[10] Codacy dashboard: https://app.codacy.com/gh/OpenTOSCA/winery/dashboard

Automated testing: More recently, testing software regularly and automatically become common in software engineering due to the Test Driven Development (TDD) aspects of Extreme Programming in the context of agile software development. In the development of GMT, we mainly made use of unit tests. On the one hand, we used unit tests to test and verify newly developed features. On the other hand, we used them for "regression testing" to ensure that previously developed features and improvements still perform as intended after newly introduced changes.

Eclipse Winery already employed up to 500 unit tests when we started with the advancements for RADON. Beginning the year we started to track the number of unit tests for GMT's nightly automated build. Figure 14 shows the evolution of the number of unit tests since then. GMT used 526 individual unit tests to verify the functionality of the code base in the beginning. The number dropped slightly down to 517 as we had to adapt, reengineer, and change several unit tests to comply with the architectural changes described in this deliverable. However, we managed to increase the number back up 567 in the course of the recent GMT development activities.

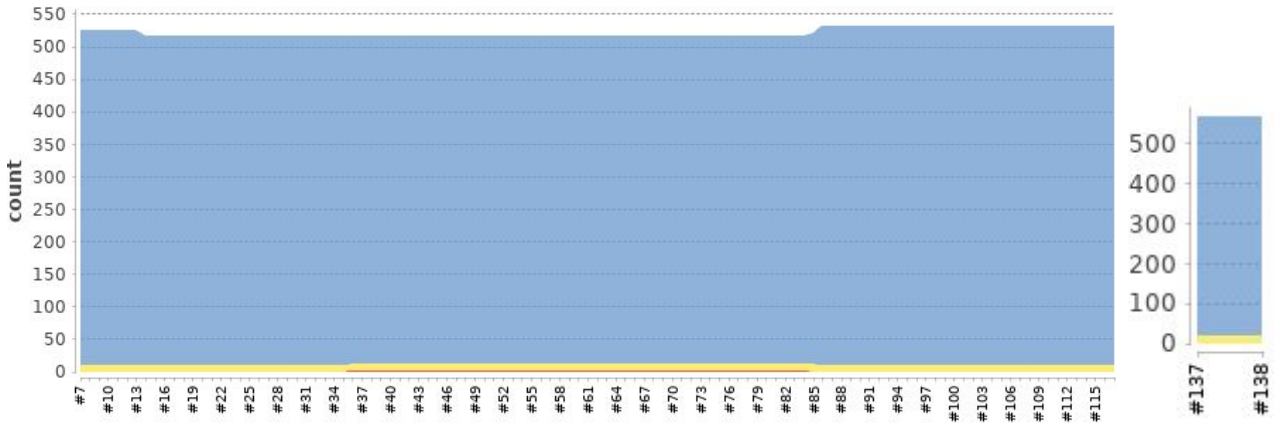

Figure 14: Evolution of the number of unit tests (since beginning of the year).

Automated builds: Working software is a crucial aspect in the era of agile software development. Therefore, Eclipse Winery has established and maintains an automated process to build and package a ready-to-use Docker image[11] of the current `master` branch. Similar to the official repository, we set-up a separate automated deployment process (Figure 15 shows the builds' activity) to publish a specific Docker image containing the advancements for the GMT. The Docker image is publicly available through Docker Hub[12]. It is automatically built whenever changes are made to the `project/radon` branch. In this way we are always able to provide the GMT as a working Docker container.

[11] Eclipse Winery Docker image: https://hub.docker.com/r/opentosca/winery
[12] RADON GMT Docker image: https://hub.docker.com/r/opentosca/radon-gmt

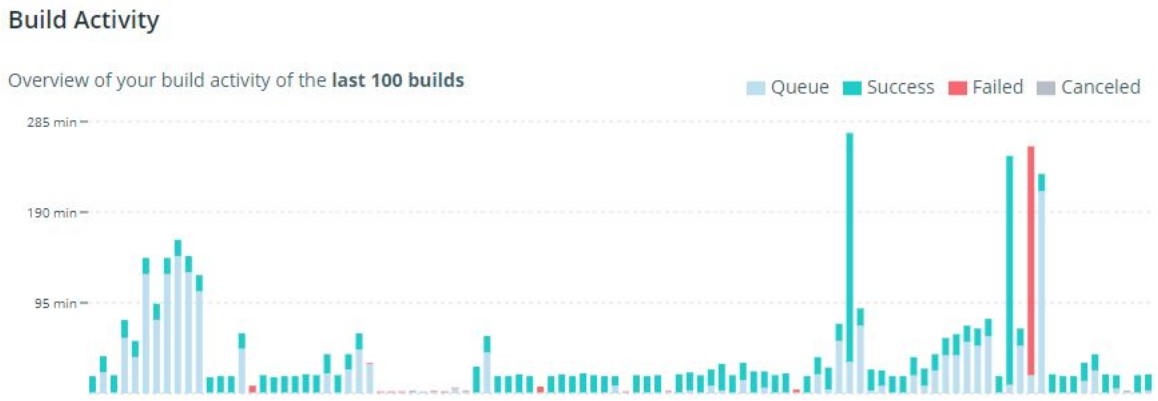

Figure 15: Automated Docker build activity.

Documentation: Another significant improvement is the updated documentation for RADON GMT which is now available using Readthedocs[13] as shown in Figure 16, a free documentation hosting service. This transition simplifies updating and versioning the documentation as well as generating multiple document formats using the documentation files. The documentation includes a comprehensive user guide as well as a dedicated developers guide which is aimed to simplify contributions from existing and new users; and is automatically built whenever new content is published to the GMT GitHub repository.

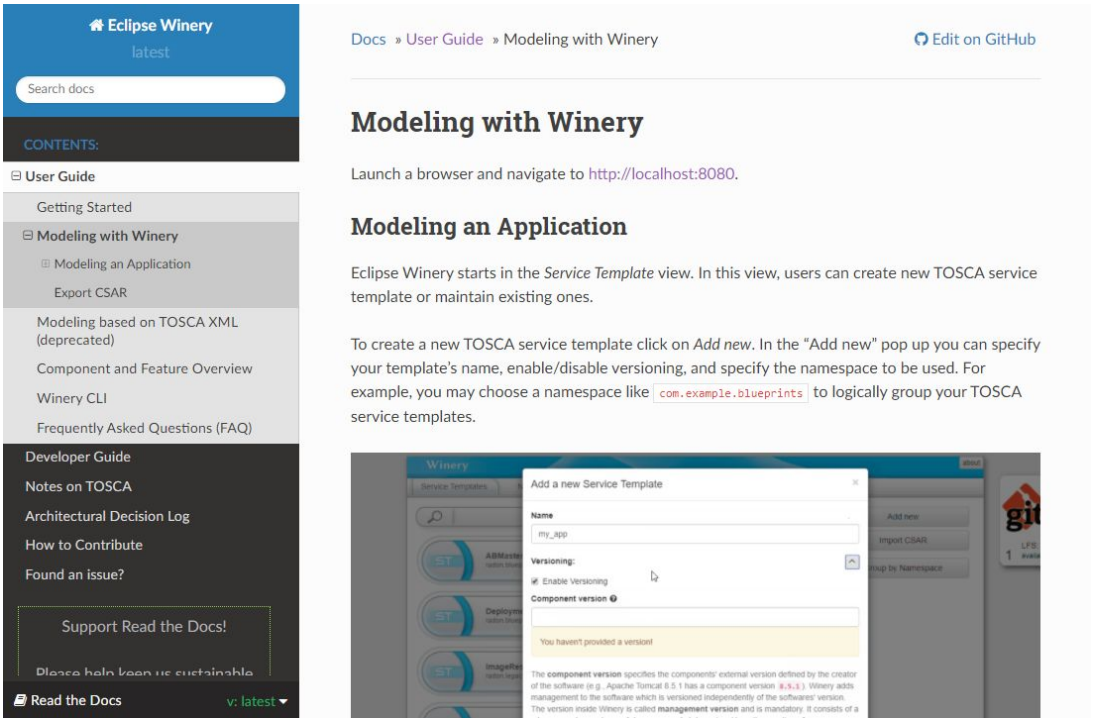

Figure 16: Automatically generated documentation for RADON GMT.

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

6 Conclusion

This deliverable presents the final architecture and implementation details towards the RADON GMT. We revised several core functionalities of Eclipse Winery to support the vision and use cases in RADON. Specifically, GMT provides full support for all TOSCA YAML modeling entities, including to define and specify custom TOSCA data types, while still being backward compatible with the former TOSCA XML standard. Further, we enabled export and import of TOSCA CSAR files to (a) share a modeled application with other RADON users and (b) enable other RADON tools to analyze and process modeled application blueprints. Among other enhancements, we integrated GMT into Eclipse Che and its underlying Kubernetes cluster. For example, GMT enables users to navigate from a graphically modeled TOSCA service template to the location in the RADON IDE where the corresponding TOSCA syntax files and Ansible implementations reside. All changes and enhancements in the course of this deliverable are proposed to be merged into the official Eclipse Winery code base. Project RADON turns Eclipse Winery into a full-fledged modeling tool for TOSCA applications and supports sustainably its open source community.

Table 2 shows an overview of the level of fulfillment for each of the agreed requirements. The labels specifying the "Level of fulfillment" are defined as follows:

(i) ✗ (unsupported): the requirement is not fulfilled by the current version

(ii) ✔ (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version

(iii) ✔✔ (partially-high supported): most of the aspects of the requirement is fulfilled by the current version

(iv) ✔✔✔ (fully supported): the requirement is fulfilled by the current version.

Table 2. Achieved level of compliance to GMT's requirements.

Id	Requirement Title	Priority	Level of fulfillment
R-T4.3-1	Integration into IDE	Must have	✔✔✔
R-T4.3-2	Navigation to business logic	Must have	✔✔✔
R-T4.3-3	Navigation to deployment logic	Should have	✔✔
R-T4.3-4	Definition of constraints	Should have	✔
R-T4.3-5	Import existing models	Could have	✔✔✔

R-T4.3-6	Integration with other RADON tools	Should have	✔
R-T4.3-7	Present verification result	Should have	✔✔
R-T4.3-8	Test case specification	Must have	✔✔✔
R-T4.3-9	Modeling lots of elements	Must have	✔✔✔
R-T4.3-10	Group modeling elements	Could have	✔
R-T4.3-11	Referencing files as URLs	Must have	✔✔✔
R-T4.3-12	Manage company-specific RADON Models and TOSCA entity type	Should have	✔✔
R-T4.3-13	Save exported CSAR to the filesystem	Must have	✔✔✔
R-T4.3-14	Manage RADON Models from RADON Template Publishing Service	Could have	✗
R-T4.4-1	Export executable deployment model	Must have	✔✔✔
R-T4.4-2	Export of different deployment model formats	Could have	✔
R-T4.4-3	Import of model in different format	Could have	✗

We hereafter discuss for each requirement briefly how it has been addressed:

- R-T4.3-1: This requirement is fully supported by two main contributions. First, a Docker image is automatically built based on the latest code base (master) of GMT. Therefore, the master branch of GMT is maintained to deliver always working software (backed with automated CI builds and unit tests). Second, a respective Kubernetes configuration allows starting the Docker image for each RADON workspace that is created in Eclipse Che. Therefore, the RADON's workspace definition (*devfile* in Eclipse Che terms) specifies a Kubernetes component employing the above mentioned configuration.
- R-T4.3-2: GMT is aware of the current Eclipse Che workspace and supports opening the RADON IDE in a new browser window in case the source code of a modeled node is maintained in Eclipse Che. Further, a special TOSCA artifact type (Repository) can be used to define arbitrary source code locations for a modeled node. GMT is also able to open these repository locations in a new browser window.
- R-T4.3-3: Similar to the previous requirement, a user can open a TOSCA service template or TOSCA node type in the IDE by using the "Open in IDE" button in the GMT. The current context (name of the TOSCA service template or node type) is passed as an URL parameter and picked up by an Eclipse Che plugin. The plugin is able to automatically select the respective directory in IDE's project explorer. This allows users to validate and

maintain the resulting TOSCA syntax and its Ansible implementations directly inside Eclipse Che.

- R-T4.3-4: In the course of the project it was decided by the consortium not to provide graphical support for defining CDL statements. Instead, GMT allows users to navigate to the respective TOSCA service template in Che's project explorer where users are able to use the IDE's text editor to define constraints specifically for a certain model. From here, the Verification Tool (VT) is used to verify the constraints for the model.
- R-T4.3-5: This requirement is fully supported as GMT offers the possibility to import a TOSCA CSAR file containing a TOSCA service template and its used TOSCA types, i.e., node types, policy types, or referenced artifacts. This way, users can share their RADON models in the form of a TOSCA CSAR file.
- R-T4.3-6: The implementation of this requirement shifted from the initial architectural decision in the direction of having a more loosely-coupled integration. The consortium agreed not to use the GMT as the point of such integrations as it introduces more restrictions on the way users interact with the RADON tools. Instead, the integration of tools is enabled by implementing another export type in the RADON GMT: TOSCA CSARs can now be stored directly in the IDE (complementary to the download option), and other tools can use exported CSARs as inputs. For example, users can generate a TOSCA CSAR at any point in time and can utilize tools such as the Defect Prediction Tool (DPT) to search for defects in its Ansible implementations. Further, tools such as the Decomposition Tool (DT) or Verification Tool (VT) can operate directly on the produced TOSCA files available through the IDE's project explorer.
- R-T4.3-7: The Verification Tool (VT) is capable of producing a *corrected* TOSCA service template of a verified model. VT creates a new version of the initial TOSCA service template containing the corrections. Users can compare the two versions by using GMT's diff features which presents a textual representation of the changes. This requirement is only partially supported as we do not visualise the difference of such versions.
- R-T4.3-8: Test cases are modeled using TOSCA policies. GMT provides the capability to maintain certain TOSCA policy types representing different testing strategies, e.g., HTTP endpoint tests. Further, such test-related policies can be attached to modeled application components using GMT's modeling frontend. The Continuous Testing Tool (CTT) processes these policies and translates them into test cases that are executed after the actual application deployment. All aspects of these test cases can be modeled and maintained using the GMT.
- R-T4.3-9: Use case providers heavily used the GMT while modeling and implementing their use cases, which showed that it was feasible to model complex application deployments consisting of many individual components. Further, an artificial RADON model (aka. TOSCA service template) was used to validate that GMT can process a topology with up to 200 TOSCA modeling entities.

- R-T4.3-10: GMT enables users to define TOSCA node groups. However, this requirement is only partially-low supported as GMT does not support a drill-down and drill-up feature, e.g., to only show the groups or to zoom into the full application topology.
- R-T4.3-11: GMT is capable of referencing artifacts for modeled application components as URLs, either staged in accessible locations or in build artifact management tools. This enables to create more lightweight models while maintaining and versioning the actual artifacts in separate repositories. GMT is also able to download such artifacts whenever a user wants to generate a TOSCA CSAR file in order to provide a self-contained package.
- R-T4.3-12: Company-specific TOSCA service templates cannot be maintained and shared in a public repository like the RADON Particles. Companies require to maintain their internal TOSCA types and blueprints in private Git repositories, such as GitLab. To fulfill this requirement, GMT provides a "multi repository" mode. In this mode, GMT is started and initialized based on a publicly available modeling repository (e.g, the RADON Particles). Types and blueprints that are developed individually are stored in a separate workspace by the GMT and can be versioned using different Git remote servers.
- R-T4.3-13: GMT enables two export modes: (1) users can download the generated CSAR to their workstation (using UI or API), and (2) the CSAR file can be saved to a location onto GMT's filesystem. The second option fully enables the loosely-coupled integration of RADON tools. This way, GMT is able to save a generated CSAR file to Eclipse Che's workspace, which enables other RADON tools to further process the CSAR files.
- R-T4.3-14: This requirement is not fulfilled as the consortium agreed to integrate the RADON Template Library Publishing Service (TLS) by employing an Eclipse Che plugin to query and publish RADON types and models directly from the IDE.
- R-T4.4-1: This requirement is fully supported by generating a self-contained CSAR file according to the latest TOSCA standard. This archive contains all required artifacts, TOSCA types, and blueprints to deploy the modeled application using the RADON Orchestrator. Further, it enables other RADON tools to analyze the modeled application and its components.
- R-T4.4-2: Initially we thought about generating and exporting additional deployment model format on top of TOSCA. Internally, the GMT is able to process, generate, and output TOSCA syntax according to the former XML standard and the latest YAML standard. In the early stage of the project we envisioned to offer support for both TOSCA standards at runtime. However, we have not pursued this further as the use case providers did not require it and RADON in general does not employ an orchestration layer for other deployment model formats. This requirement is only partially-low supported as the GMT uses a configuration parameter to decide at start-up which TOSCA syntax can be produced and processed during runtime; at least to guarantee backward compatibility with the former version of the TOSCA standard.
- R-T4.4-3: In the beginning of the project it was planned to integrate different RADON tools directly with the GMT. Due to the decision in the direction of having a more

loosely-coupled integration based on a TOSCA CSAR it was no longer necessary to support different import formats to be consumed by the GMT.

Future work. For the remaining project period we will focus on supporting the use case providers ATC, PRQ, and ENG to implement their envisioned use cases by using the GMT. Specifically, we consult and help to further enhance their RADON Models. Beside that, we work continuously on updates for "could have" requirement implementations, defect fixes, and provide an automatically built Docker image of the latest GMT code base, which is also used by each RADON workspace when started in Eclipse Che.

References

[D2.1] RADON Consortium, “Deliverable D2.1: Initial Requirements and baselines”, 2019

[D2.2] RADON Consortium, “Deliverable D2.2: Final requirements”, 2020

[D2.6] RADON Consortium, “Deliverable D2.6: RADON Integrated Framework I”, 2020

[D4.3] RADON Consortium, “Deliverable D4.3: RADON Models I”, 2019

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

[D4.5] RADON Consortium, “Deliverable D4.5: Graphical Modelling Tool I”, 2019

[OASIS2020] OASIS, TOSCA Simple Profile in YAML Version 1.3, 2020

[Casale2019] G. Casale et al.: “Rational Decomposition and Orchestration for Serverless Computing”, The Symposium and Summer School on Service-Oriented Computing (SummerSoc), 2019, (accepted for publication)

[Kopp2013] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Service-Oriented Computing: 11th International Conference“. In: Springer Berlin Heidelberg, 2013. Kap. Winery – A Modeling Tool for TOSCA-Based Cloud Applications, S. 700–704. doi: 10.1007/978-3-642-45005-1_64

[Lipton2018] Lipton et al.: “TOSCA Solves Big Problems in the Cloud and Beyond!”, IEEE Cloud Computing, 2018