



Sodalite

Software Defined AppLication Infrastructures management and Engineering

Final version of ontologies and semantic repository

D3.2

CERTH

31.10.2021



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825480.

Deliverable data			
Deliverable	Final version of ontologies and semantic repository		
Authors	Georgios Meditskos (CERTH), Zoe Vasileiou (CERTH), Savvas Tzanakis (CERTH), Anastasios Karakostas (CERTH), Stefanos Vrochidis (CERTH), Xefteris Vasileios-Rafail (CERTH), Grigoris Tzionis (CERTH), Stylianios Andreadis (CERTH), Dourvas Nikolaos (CERTH), Stathis Nikolaidis (CERTH), Jesús Gorroñoigoitia (Atos)		
Reviewers	Elisabetta Di Nitto (POLIMI) Nejc Bat (XLAB)		
Dissemination level	Public		
History changes of	Zoe Vasileiou (CERTH)	Outline created	31/08/21
	All	Initial contribution to sections	05/09/21
	Jesús Gorroñoigoitia (Atos)	Section 6 and subsection 7.3 written	15/09/21
	All	Initial version of the deliverable finalized	09/10/21
	Zoe Vasileiou (CERTH)	Sent for internal review	15/10/21
	Zoe Vasileiou (CERTH)	Addressing internal review	26/10/21

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action SODALITE (project no. 825480), started in February 2019, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-16-2018: Software Technologies)



Table of Contents

Executive Summary	7
Glossary	9
List of figures	11
List of Tables	12
1 Introduction	13
1.1 Deliverable goal	13
1.2 Overall objectives of the project	13
1.3 Progress beyond the state of the art and potential impact	15
1.4 Work performed from the beginning of the project	15
1.5 Structure of the Document	16
2 Final SODALITE Conceptual Models	17
2.1 New concepts added to the ontologies	17
2.1.1 Scaling Policies example	21
2.1.2 Topology	24
2.1.3 Resource Model	25
2.1.4 Inputs	25
2.1.5 Outputs	27
2.1.5 Optimization	28
2.1.5 KB architecture	29
2.2 Evaluation of SODALITE Ontologies	31
2.2.1 Assessment	31
2.2.2 Comparison with other ontologies	37
3 Updated Ontology Population and Checking	39
4 Advanced Reasoning services	42
5 SODALITE IDE	43
5.1 Domain Specific Languages	43
5.1.1 AADMs	44
5.1.2 RMs	44
5.1.3 Optimization Models	44
5.1.4 Ansible Models	45
5.1.5 Alerting Rule Models	45
5.2 New and extended features (M18 - M33)	45
5.2.1 Multiview representation of the AADM	45
5.2.2 Extended AADM Deployment Support	49
5.2.3 Deployment Governance View	50
5.2.4 KB Browser view	52
5.2.5 Improved Content Assistance	53



5.2.6 Semantic Validation (Reporting and Quick fixes)	55
5.2.7 AADM Versioning	56
5.2.8 IAM/Secrets Management	57
5.2.9 Image Builder Integration	58
5.2.10 PDS Integration	58
6 Updated Modelling Layer architecture	59
7 Final version of the components	61
7.1 Semantic Reasoner	61
7.1.1 Semantic Reasoning Engine (SRE)	63
7.1.1.1 Advanced Reasoning services - examples	64
Validation	64
Matchmaking and reuse	67
Abstraction DSL	69
Optimization suggestions	70
7.1.2 Semantic Population Engine (SPE)	72
7.1.2.1 Workspaces and Versioning	72
7.1.3 CI/CD Integration	73
7.2 Semantic Knowledge Base	73
7.2.1 RDF Triple Store	73
7.2.2 Domain Models	73
7.2.3 CI/CD Integration	73
7.3 SODALITE IDE	74
7.3.1 Domain Specific Languages	76
7.3.2 New and extended features (M18 - M33)	77
7.3.3 Multiview representation of the AADM	77
7.3.4 Extended AADM Deployment Support.	78
7.3.5 Deployment Governance and KB Browser Views	78
7.3.6 Improved Content Assistance	79
7.3.7 Semantic Validation (Reporting and Quick fixes)	79
7.3.8 AADM Versioning	79
7.3.9 IAM/Secrets Management	79
7.3.10 Image Builder and PDS Integration	79
7.3.11 CI/CD Integration	79
8 Conclusion	80
References	81
Appendix	83
DSL representation	83
Exchange Model	85
Sodalite Meta-model	93
Semantic Reasoner	93



Semantic Reasoning Engine (SRE)	93
Final version of the APIs	93
Semantic Population Engine (SPE)	101
Final version of the APIs	101
RDF Triple Store	104
Domain Models	105

Executive Summary

This deliverable reports on the status of the development, at M33, of the SODALITE Modelling Layer and the integration of its components with the rest of the platform. The purpose of this deliverable is to present the final version of the Modelling Layer which has been developed during the second and the third year of the project. More specifically, the work that has been performed within the T3.1 “Application Semantic Modelling” and T3.2 “Infrastructure Semantic Modelling” tasks will be reported. The first version of the Modelling Layer is presented in D3.1[2]. This deliverable complements D4.2 [3], and D5.2 [24], and the interested reader is encouraged to read those deliverables for having a full understanding of the whole process. The goal of the Modelling Layer is the abstraction of the application deployment models. To this end, the Modelling Layer aims at:

- Describing the cloud applications and infrastructures in an abstract way
- Reducing the user effort in describing IaC by providing intelligent services
- Supporting the user through the whole lifetime of the application deployment

For achieving the above goals, the Modelling Layer components have been evolved since M12, and the progress of the following achievements will be presented in this deliverable:

- **Ontologies:** The ontologies have been extended to support new TOSCA concepts and the abstract SODALITE metamodel has been enhanced for adding new metadata. The final version of the ontologies are presented in this deliverable.
- **Semantic Reasoner - Populator:** The Semantic Populator populates the KB with AADMs (Abstract Application Deployment Models) and RMs (Resource Models). It has been enhanced significantly for mapping the new TOSCA concepts and metadata to the SODALITE ontologies, and supporting new features such as the versioning of the AADMs and the workspaces.
- **Reasoning layer:** The Reasoning layer is responsible for providing context-aware content assistance to the user and ensuring the consistency of the models that are saved to the KB. This intelligence is provided to the WP4 (T4.4) through REST APIs presented in this deliverable.
- **IDE:** The IDE has been evolved significantly for supporting the user during the whole process of the app deployment, at design, deployment and runtime. Different SODALITE roles for app deployment modeling are supported by the adoption of various SODALITE domain specific languages.
- **Integration with other components:** The Modelling Layer is central to the architecture since it is the entry point of the platform as the users use the IDE for managing their deployments, and also all the other layers (Infrastructure as a Code layer, and Runtime layer) retrieve, modify and save models from/to the Semantic Knowledge Base (KB). In particular, the IaC layer retrieves from the Modelling Layer the representation of the AADMs for producing the TOSCA blueprint and saves new discovered alternative variants and refactoring the existing AADMs accordingly. Therefore, a significant effort was placed on the integration of the Modelling Layer with the other layers.

Glossary

Acronym	Explanation
AADM	Abstract Application Deployment Model
AOE	Application Ops Experts
API	Application Program Interface
DL	Description Logic
DOLCE	Descriptive Ontology for Linguistic and Cognitive Engineering
DnS	Descriptions and Situations
DSL	Domain Specific Language
DUL	DOLCE Ultralite
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IDE	Integrated Development Environment
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
KB	Knowledge Base
OASIS	Organization for the Advancement of Structured Information Standards
ODP	Ontology Design Pattern
OWL	Web Ontology Language
PaaS	Platform as a Service
PoC	Proof of Concept
QE	Quality Expert
QoS	Quality of Service
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RDF4J	Resource Description Framework for Java (open source java framework)
RE	Resource Expert
REST	Representational State Transfer
RM	Resource Model
SaaS	Software as a Service
SPARQL	SPARQL Protocol and RDF Query Language
SPE	Semantic Population Engine



SRE	Semantic Reasoning Engine
TOSCA	Topology and Orchestration Specification for Cloud Applications
WP	Work Package

List of figures

- [Figure 1. SODALITE meta-model \(extension of DUL\)](#)
- [Figure 2. Hierarchy of new TOSCA properties](#)
- [Figure 3. Hierarchy of new TOSCA concepts and situations](#)
- [Figure 4. Hierarchy of Root with respect to the normative TOSCA policies](#)
- [Figure 5. Hierarchy of the TOSCA normative artifact types](#)
- [Figure 6. Policy template example of DSL](#)
- [Figure 7. Autoscale policy template knowledge graph](#)
- [Figure 8. Example of property definition](#)
- [Figure 9. Example of trigger definition](#)
- [Figure 10. Example policy type](#)
- [Figure 11. Example Topology Instance](#)
- [Figure 12. Example Resource Model](#)
- [Figure 13. Inputs example](#)
- [Figure 14. Inputs example knowledge graph](#)
- [Figure 15. A node template example that is using the inputs](#)
- [Figure 16. Output example](#)
- [Figure 17. Output Knowledge Graph](#)
- [Figure 18. A node template related with an optimisation DSL script](#)
- [Figure 19. Optimization Knowledge Graph](#)
- [Figure 20. Graph dataset](#)
- [Figure 21. The graphs of the KB](#)
- [Figure 22. Excerpt from a Policy Template](#)
- [Figure 23. Template example with reference to versioned node](#)
- [Figure 24. Text-based representation of AADMs](#)
- [Figure 25. Outline tree-based representation of AADMs](#)
- [Figure 26. Graphical-based representation of AADMs](#)
- [Figure 27. Form-based edition of AADM entities](#)
- [Figure 28. AADM deployment wizard](#)
- [Figure 29. Deployment Governance View](#)
- [Figure 30. Deployment Details View](#)
- [Figure 31. Deployment Resume wizard](#)
- [Figure 32. KB Browser view](#)
- [Figure 33. Content assistance for AADM syntax and structure](#)
- [Figure 34. Content assistance for data types](#)
- [Figure 35. Content assistance for node types](#)
- [Figure 36. Content assistance for requirements](#)
- [Figure 37. Content assistance for nodes satisfying requirements](#)
- [Figure 38. Semantic validation issue reported in AADM textual editor](#)
- [Figure 39. Quick fixes in AADM textual editor](#)
- [Figure 40. AADM versioning in save wizard](#)
- [Figure 41. IAM configuration](#)
- [Figure 42. HPC Secrets configuration](#)
- [Figure 43. Image Builder wizard](#)
- [Figure 44. PDS wizard](#)
- [Figure 45. Updated SODALITE layers general architecture](#)
- [Figure 46. Updated Modelling Layer Architecture](#)
- [Figure 47. Modelling Layer architecture and its interactions with the other layers](#)



- [Figure 48. Code Quality Report for Semantic Reasoner](#)
- [Figure 49. SPARQL query detecting Requirement Mismatch](#)
- [Figure 50. Capability Mismatch model example](#)
- [Figure 51. SPARQL query detecting Capability Mismatch](#)
- [Figure 52. Excerpt from the `sodalite.nodes.DockerNetwork`](#)
- [Figure 53. SPARQL query detecting required properties](#)
- [Figure 54. A node type with constraints within properties](#)
- [Figure 55. SPARQL query returning the constraints](#)
- [Figure 56. Node type hierarchy](#)
- [Figure 57. Template for which a suitable host is detected](#)
- [Figure 58. SPARQL detecting compatible requirements according to the type definition](#)
- [Figure 59. SPARQL retrieving the required capability types from a node](#)
- [Figure 60. SPARQL retrieving the valid source types of node types](#)
- [Figure 61. A SPARQL query detecting required requirements](#)
- [Figure 62. AI Training node template associated with optimization DSL](#)
- [Figure 63. SPARQL query retrieving the capabilities of the node](#)
- [Figure 64. SPARQL query retrieving the number of gpus](#)
- [Figure 65. Example model using workspaces](#)
- [Figure 66: SODALITE IDE plugins' project structure](#)
- [Figure 67: Snippet for AADM DSL grammar](#)
- [Figure 68: Sirius design of AADM visual representation](#)

List of Tables

- [Table 1. Indicative CQs and SHACL-SPARQL translations](#)
- [Table 2. Ontology pitfalls detected by OOPS!](#)
- [Table 3. Schema metrics produced by the Ontometrics tool](#)
- [Table 4. Base metrics produced by the Ontometrics tool](#)
- [Table 5. Comparison of the modelling capabilities of cloud computing ontologies](#)

1 Introduction

With the surge of the adoption of Cloud Computing, more and more IaC languages emerge leading to the vendor-lockin [1] problem. Many cloud resource management standards have been proposed for solving this interoperability issue. In such a way, many standards are proposed such as TOSCA, OCCI and CIMI are among the most known standards; however, those cloud resources described in different standards, still face interoperability problems as the semantics of the different IaC languages differ. Therefore, the representation of cloud resources in a machine-readable format is more indispensable than ever.

As described, in detail, in D3.1 [2], Semantic Web Technologies, and especially the ontologies and the reasoning mechanisms can achieve this interoperability and additionally an intelligence support system. Most of the existing cloud computing ontologies are mainly monolithic, and do not promote reusability, and interoperability. The SODALITE Conceptual model follows an Ontology Design Pattern (ODP) paradigm for building the Modelling Layer and achieving abstraction, interoperability, and flexibility. This conceptual model fosters the intelligence of the Modelling Layer, as advanced reasoning can be performed for matchmaking, validation, reuse, and context-aware content assistance.

Some of the key objectives of the Modelling Layer were described in D3.1:

- Follows a common, extensible and formal standardised model to describe cloud-related concepts.
- The information to be shared and managed as interconnected RDF knowledge graphs for capturing both structural and semantic relationships.
- Provides semantic-driven modelling assistance to the user.
- A centralised management board to be provided to the user in the IDE for having access in all the phases of the deployment.

1.1 Deliverable goal

The goal of this deliverable is to provide the final version of the Modelling Layer with respect to the points described above. In particular, this deliverable focuses on the progress of the work with respect to what was reported in Deliverable D3.1 at the end of the first project year.

Therefore, this deliverable presents the updates that have been made in the Modelling Layer components that were developed after the end of the first year of the project when the initial deliverable D2.1 [26] was produced, namely, the SODALITE ontologies, the Knowledge Base, the IDE, the Semantic Reasoner. Also, the integration of the Modelling Layer components with various components of the other layers will be presented. This deliverable has been developed in parallel and coherently to WP2, WP4, WP5 and WP6.

1.2 Overall objectives of the project

The main project goals, according to the GA, can be summarized as follows, with respect to how the current WP reflect on them:

- **O1 objective:** *The key objective is to provide code (application), resource (infrastructure) and execution semantic abstractions, injected with infrastructure performance abstractions, to ensure maximum performance of the so-abstracted application and infrastructure when concretized on specific infrastructure. We build the abstractions as extensions of standardised approaches, aiming at both machine and human readability.*

WP3 perspective: The SODALITE conceptual model is an extensible and reusable model that captures the knowledge in an abstract way. All the required abstractions by the use cases have been modelled such as type definitions, template definitions, optimization models, ansible models etc.

- **02 objective:** *To increase the performance of the deployed software on target platforms through static optimisation, using the Infrastructure performance patterns abstractions and through dynamic optimisation, using the predictive deployment refactoring approach, building on the run-time available data from the application and the platform.*

WP3 perspective: Static optimizations are supported through the MODAK, the SODALITE Application Optimizer D4.2 [3]; The Modelling layer enables the design of the optimizations in the IDE, since Optimization DSL is provided, and also suggestions are provided during the authoring process. For example, in case, an optimization flag is disabled, and according to the model capabilities, this optimization flag is recommended to be enabled. Regarding the run-time optimizations, the Knowledge Base enables the discovery of alternative resources as it is the main storage of all the application and infrastructure components. A reasoning infrastructure enables the saving of the discovered infrastructure resources by Platform Discovery Service[3] to the KB. Those resources can be reused by the Refactorer[20] by matchmaking a suitable resource in order to migrate an application component.

- **03 objective:** *To reduce the cost of software development, deployment, management and adaptation or reconfiguration in the modern infrastructures, using abstraction of the typical components (e.g. compute, storage, network) and combining them with non-functional requirements, allowing for an application to target multiple concrete infrastructures.*

WP3 perspective: The SODALITE ontologies provide an abstraction layer for representing both the application and resource models. Those ontologies enable advanced reasoning services to infer hidden knowledge, and retrieve semantic information to assist the user. In such a way, SODALITE IDE is providing intelligence to the user through an ontology-based reasoning engine for guiding the user in the design phase of the deployment models. These intelligent capabilities lessen the user effort in designing and deploying models. Additionally, the IDE supports the user through the whole lifecycle of the deployment, namely design, deployment, and runtime phases.

- **04 objective:** *To address abstractions, technologies, targeted applications, and infrastructures holistically, allowing for flexible, reusable, and long term supported software development stack for modern runtime infrastructures and professional applications.*

WP3 perspective: The Modelling Layer is the central point to the SODALITE platform as the user interacts with the IDE and also the KB contains all the models that are used by many components of the platform. All the Modelling Layer components have been integrated in the platform and contributed to a well-integrated platform. Both the IDE and Semantic Reasoner interact with components of the other SODALITE layers, namely the IaC and Runtime layers.

- **05 objective:** *To use and build on existing solutions, starting with community building or inclusion from day 1.*

WP3 perspective: All the Modelling Layer components are open-source. Also, the SODALITE ontology is based on an open-source upper-level ontology.

- **06 objective:** *To demonstrate the developed concepts using relevant professional applications and industries, covering complete software stack.*

WP3 perspective: The Modelling Layer supports all the three use-cases (that cover specific professional application or industry) of the project for modelling all the concepts needed.

1.3 Progress beyond the state of the art and potential impact

This deliverable is contributing to the progress beyond the state of the art by offering a complete environment that guides users through all the phases of the deployment. The SODALITE Modelling layer represents a novel result in the literature because it provides an abstraction layer for TOSCA-compliant descriptions of applications and resources that enable the smart-editing IDE features. Also, this abstract representation of the cloud resources allows the other layers, IaC and runtime to reuse the concepts saved in the Knowledge Base.

As already presented in D3.1, the SODALITE conceptual model promotes this abstraction layer by adopting the best practices in Ontology Engineering [4]. As such, a novel ontology-based framework is proposed that captures and interlinks all the information of the cloud-related components. This information is saved as interconnected RDF Knowledge Graphs[5] in the Knowledge Base enabling deep inference reasoning to run upon them for advanced validation, matchmaking etc. The innovative modelling capabilities of the SODALITE ontology were presented in a conference paper [7]. In the third year of the project, we focused on the following aspects: a) extending the ontologies to represent also non functional features and new TOSCA concepts, b) advancing the intelligence provided to the user, and c) enabling the interactions with other layer components for supporting the user also during the deployment time and runtime. The SODALITE smart environment for Infrastructure as a Code was presented in a workshop paper [8][9].

Also, we were paying attention to the [quality of code](#) so as to be maintainable and extensible and also to be robust with regards to security. Additionally, we were regularly testing the Modelling Layer components through the deployment of the SODALITE use cases.

1.4 Work performed from the beginning of the project

During the first year project, we developed an initial version of the components of the Modelling Layer. A first version of the ontologies were developed covering the major part of the TOSCA normative types. The saving of the AADMs were supported through the IDE, while Resource Models were not supported yet. An initial version of the Semantic Reasoner was also developed offering some basic context-assistance and validation services. Also, the first version of the IDE was developed.

During the second year, the saving of the Resource models was developed, the ontologies were extended for supporting non-functional features, and new TOSCA concepts, the IDE was enhanced with multiple features such as support to new DSL languages (e.g., the optimization DSL), multiview representation of the models, allowing the images to be built before the deployment etc. Also, the Semantic Reasoning engine was extended by providing more assistance and validation cases to the user.

During the final year, we were focusing on finalizing the ontologies by adding more TOSCA concepts, and adding more metadata such as for supporting the versioning of the AADMs. The Semantic Reasoning engine has been extended for supporting more advanced validation and abstraction cases. Also, the semantic populator was extended for enabling the saving of the discovered TOSCA models by the Platform Discovery Service to the KB, and the refactoring of the AADMs by the Refactorer. The IDE was enhanced by extending the existing DSLs languages for supporting new TOSCA concepts and by integrating with various components of all the layers to give access to the user to the whole deployment lifecycle.



1.5 Structure of the Document

This deliverable is structured as follows:

- Section 2 presents the ontology extensions for supporting new concepts, provides knowledge graphs examples with respect to the new concepts and additionally and explains how the SODALITE Ontology has been evaluated.
- Section 3 presents an example of the intermediate Exchange Model, the lightweight version of the SODALITE ODP in order to show its main updates.
- Section 4 presents briefly the advanced reasoning services from which the intelligence of the Modelling Layer derives.
- Section 5 presents the improvements and the new features of the SODALITE IDE.
- Section 6 explains the positioning of the Modelling Layer in the SODALITE architecture and its evolution during the second and third year.
- Section 7 is a manual associated with the tool implementation that shows the final version of the SODALITE Modelling Layer components. It has been built to be self-contained as it could be extracted from the main document. For this reason, the reader may find there some repetitions compared to the content of the other sections
- Section 8 presents the conclusions.
- Finally, the Appendix provides an example of topology using new concepts (policies, triggers, outputs etc.) in three different formats, in the DSL, the intermediate Exchange Model, and in the SODALITE metamodel.

2 Final SODALITE Conceptual Models

In this section, we present the updates that have been applied to the SODALITE conceptual model. In D3.1, the background for the knowledge representation and ontologies was described, and also an initial version of the ontology-based semantic abstraction layer was presented that captures the infrastructure resources and the application components. The former are specified by the Resource Experts, while the latter by the Application Ops Experts (see D2.3 [10], Section 2.1 “Identified Use Cases”).

The final version of the semantic models of SODALITE includes:

- The **SODALITE meta-model** which is the formal ontology design pattern that is used for capturing the information on different levels of abstraction.
- The **domain ontology** that captures the TOSCA normative types and generally the vocabulary of TOSCA that will be utilized in the other two modelling layers (tiers), in SODALITE, specifically the custom resources (Tier 1), and instantiations of the resources (Tier 2), the application components. More specifically, the domain ontology represents the Tier 0, the static layer of the SODALITE abstraction layers. The 3-tier approach is described in the D3.1.
- The **optimization ontology** is a lightweight ontology used for enabling the semantic reasoning that is applied on the association between the optimization model and the node template in order to provide suggestions to the user for potential optimizations.

2.1 New concepts added to the ontologies

The **TOSCA domain ontology** has been enriched so as the following constructs to be supported in the Resource Models, and the AADMs:

- **Inputs:** Input values are often used to declare input variables that will be used by interfaces and operations for passing this information to the operations (scripts) which need this data. The interfaces are defined within the nodes and are associated with lifecycle operations including scripts such as starting a database etc. The inputs are passing values to those scripts such as usernames, and passwords. A topology template in TOSCA [11][12] defines the structure of a service in the context of a Service Template.
- **Outputs:** Output values are usually provided for passing information for the state of templates that have been already deployed.
- **Policy types:** The non-functional features and Quality of Service (QoS), in TOSCA [12], are described through policies for an application and its components.

In policy type, it is specified the types of policies that may be used in a service.

All the TOSCA normative policy types have been added to the TOSCA domain ontology (tier 0). For example, some of the above-mentioned types are `tosca.policies.Root` (all the policy types derive from), `tosca.policies.Placement` (governing the placement of the nodes), `tosca.policies.Scaling` (governing the scaling of nodes).

Within a policy type, its properties, its targets, and its triggers are defined. The properties define the type of the configuration parameters that will be used by the policy. The targets represent the types of services to which the policies will apply. The triggers define the conditions that fire the policy. Henceforth, for supporting the policy types in our platform, various additional concepts have been added to the ontologies such as triggers, targets, event, condition, and other vocabulary that was needed for supporting the policies functionality.

- **Policies:** With policies[11], we can define the types of requirements that govern the use or access to the resources and can be expressed separately from the applications. In such a way, our platform is offering new benefits as we can describe how the AADM reacts to the changes in runtime. According to the TOSCA specification [12], the policies are split in three categories: (i) Access Control (ii) Placement (iii) Quality Of Service. For instance, a QoS policy could be an autoscaling policy, an example [13] is presented in this paper that uses both TOSCA policy types and templates. Policies are the instances of the policy types that are used in a service. Within a policy, the configuration parameters (*properties*) that the policy takes are defined and the target templates (*targets*) to which the policies apply are specified.
- **Capabilities/properties:** Properties can be assigned within capabilities in a template in order to assign parameters such as number of cpus, number of gpus, the architecture etc. Those properties are used for defining the capabilities for a template.
- **Optimization:** An optimization recipe is specified by an OE that designs the deployment optimizations as instances of the Optimization Model (OM) DSL. This recipe is linked with the MODAK (see D4.2) static optimizer. In particular, an application component in the AADM is associated with an optimization model. Henceforth, the ontologies have been enriched for accommodating this kind of concept.
- **Artifact types and artifacts:** An artifact represents something that can be executed, and TOSCA supports different kinds of artifacts, namely implementation and deployment artifacts. The artifact types are reusable TOSCA entities that define one or more files that are used for defining those artifacts that are referenced in the operations of the nodes or relationships. Those files are used by the orchestrator at the deployment time when deploying nodes or when calling their interfaces. By representing artifact types, users can define their own artifact types and implementation artifacts allowing the user to interact with any cloud computing service (IaaS, SaaS, PaaS). In those types, the type of the file, the extension of the file (*file_ext*), the mime_type (*mime_type*), and its properties are defined. Additionally, the artifact assignments can be present both in types and templates.

The knowledge graphs of some of the concepts in the above list will be presented in the next figures.

As it was described in D3.1, the SODALITE meta-model follows an Ontology Design Pattern(ODP) that extends the core Descriptions and Situations (DnS) ontology pattern that is part of DOLCE+DnS Ultralite (DUL) [6]. The SODALITE ODP is depicted in Figure 1. SODALITE ODP is a generic ontology pattern that defines the modelling guidelines to be followed in order to capture the domain knowledge. The domain ontology is required for providing the necessary vocabulary in order to capture the relevant knowledge in the application domain. Figure 2 shows some new key properties, relevant to the concepts described in the above list, that have been added to the domain ontology and are subproperties of `soda:specification`. Those properties can be used in `soda:SodaliteDescription` instantiations. Figure 3 depicts some new domain concepts, described in the aforementioned list, that are subclasses of the `soda:SodaliteConcept` for being used as instantiations of the SODALITE ODP. Additionally, the Input and Output structures of TOSCA, in Figure 3, are shown as a subclass of the `soda:SodaliteSituation`; thus they are represented similarly with the node types and templates.

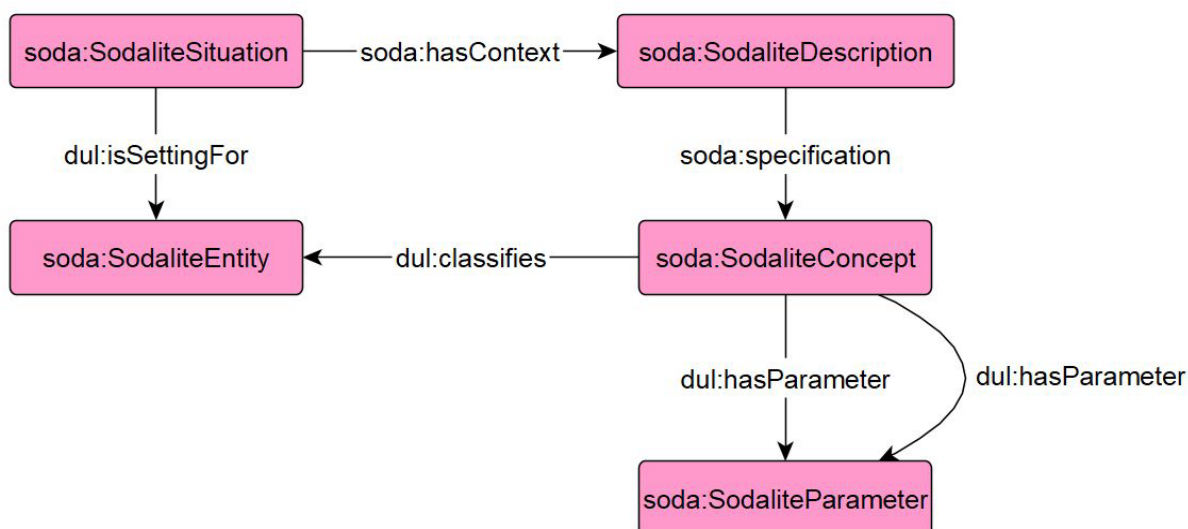


Figure 1. SODALITE meta-model (extension of DUL)

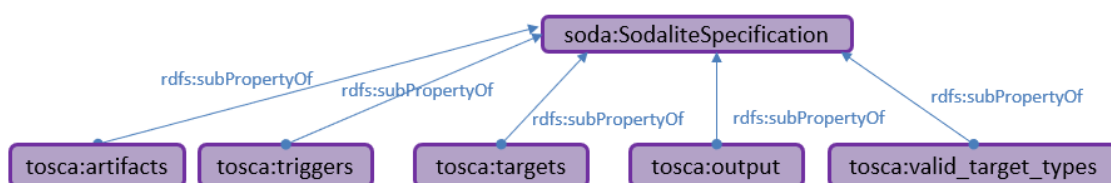


Figure 2. Hierarchy of new TOSCA properties

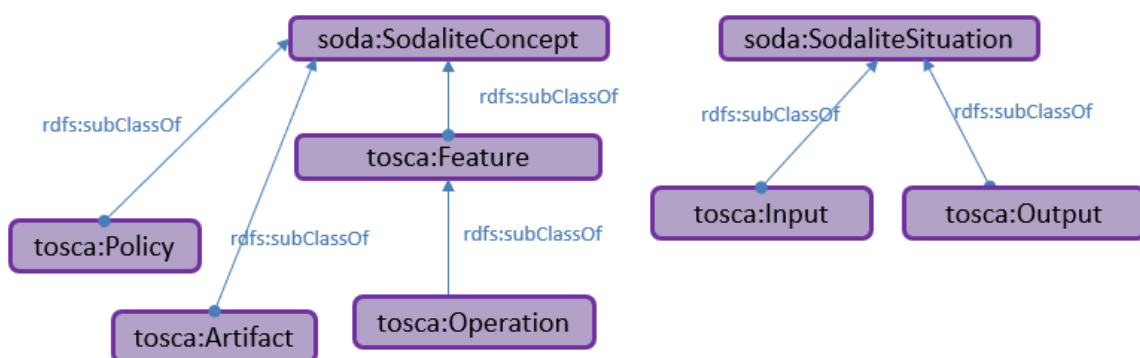


Figure 3. Hierarchy of new TOSCA concepts and situations

Also, Figure 4 depicts the TOSCA normative policy types that have been added to the TOSCA domain ontology (Tier 0). TOSCA employs policies for expressing the non-functional behavior, and are instantiated from the Policy Types. The TOSCA normative policy types are five, the root type(`tosca.Policies.Root`) that the other four inherit (`rdfs:subClassOf`). Those policy Types can represent different types of policies[12], such as access control, placement, and QoS.

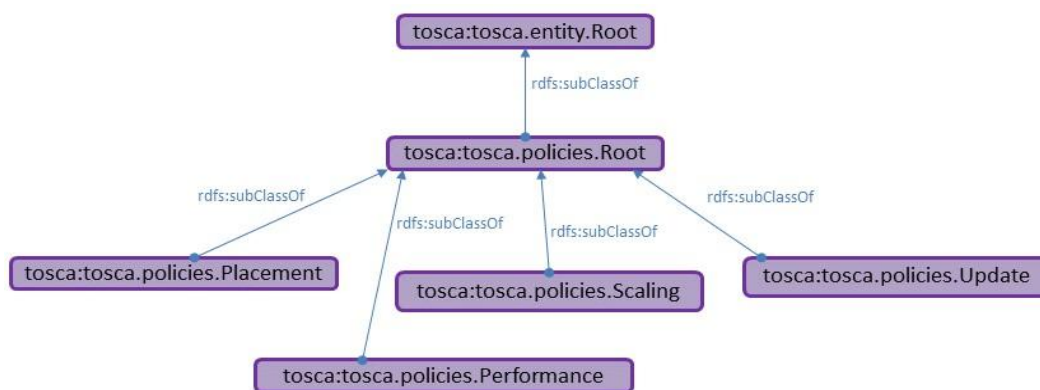


Figure 4. Hierarchy of Root with respect to the normative TOSCA policy types

Figure 5. depicts the TOSCA normative artifact types that have been added to the TOSCA domain ontology (Tier 0). The artifact types define the different kinds of files that can be supported. Those files are used during operations such as for an install or a deployment. All the normative artifact types inherit the `tosca.artifacts.Root`. Each type is a `soda:SodaliteSituation` that contains a description (`soda:SodaliteDescription`), but for brevity only the inheritance relationships are depicted.

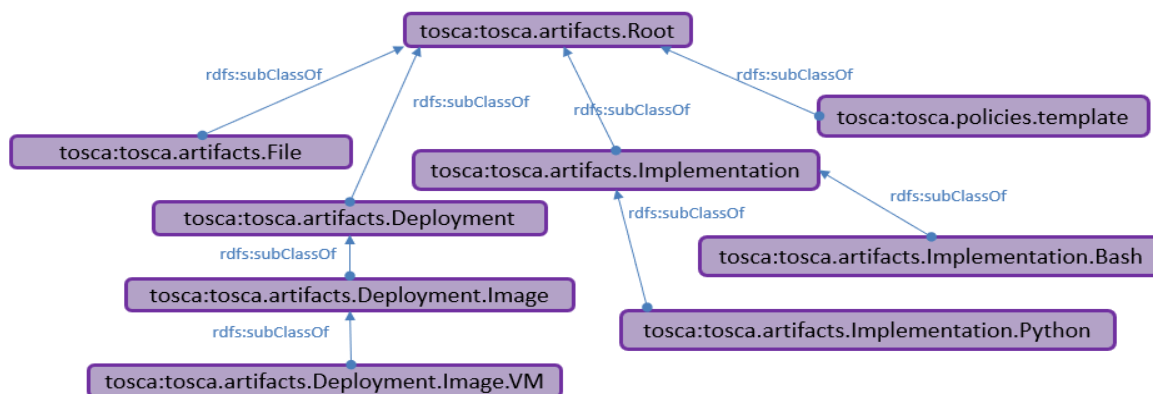


Figure 5. Hierarchy of the TOSCA normative artifact types

Based on the modelling paradigm that was described in this section, which is offering extensibility and interoperability, the ontologies were easily extended for supporting the TOSCA policies. In order to show the new concepts that have been added to the SODALITE ontologies, a Proof of Concept (PoC) example for scaling policies will be presented below showing how the policies are captured as Knowledge Graphs.

2.1.1 Scaling Policies example

An example DSL of a scaling policy template is depicted in Figure 6. This is one of the results of the collaboration with the RADON project¹. It is an autoscaling template which has various concept assignments (properties, targets, triggers). The properties define which concepts will be

¹ RADON project - <https://radon-h2020.eu/>

monitored. The triggers section defines the triggers that would initiate the scaling operation. The targets section defines to which templates the policy applies.

Example A Policy Template (Tier 2)

```
autoscale:
  type: radon/radon.policies.scaling.AutoScale
  properties:
    min_size: 3
    max_size: 7
  targets: [ radon/openstack_vm ]
  triggers:
    radon.triggers.scaling:
      description: 'A trigger for autoscaling'
      event: 'auto_scale_trigger'
      schedule:
        start_time: "2020-04-08 21:59:40"
        end_time: "2022-04-08 21:59:50"
      target_filter:
        node: radon/openstack_vm
        requirement: radon/openstack_vm.host
        capability: tosca.nodes.Compute.host
      condition:
        constraint:
          not:
            and:
              available_instances: [ greater_than: 42 ]
              available_space: [ greater_than: 1000 ]
          period: '60 sec'
          evaluations: 2
          method: 'average'
      action:
        call_operation:
          operation: radon/radon.interfaces.scaling.AutoScale.retrieve_info
        call_operation:
          operation: radon/radon.interfaces.scaling.AutoScale.autoscale
```

Figure 6. Policy template example of DSL

In Figure 7, the knowledge graph of the scaling policy template is depicted. The templates are captured according to the SODALITE ODP (Figure 1). The autoscale template has as context (soda:hasContext) a description (soda:SodaliteDescription) which describes all the concepts. The description has four concepts, two properties (tosca:properties), one trigger (tosca:triggers) and one target (tosca:targets). The knowledge graph of each concept will be analysed in Figures 7, 8, 9.

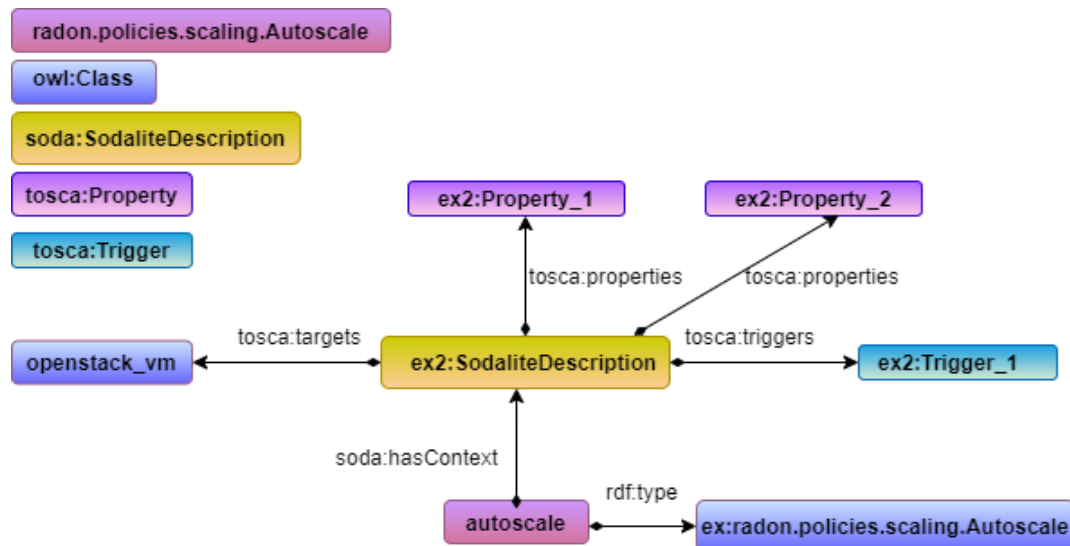


Figure 7. Autoscale policy template knowledge graph

Properties:

The knowledge graph of the properties of the autoscale template is depicted in Figure 8. The properties are instances of the `tosca:Property` concept, and each property classifies (`dul:classifies`) the property that we want to model. In our case, the max and min size of the instances are modelled in order to be monitored.

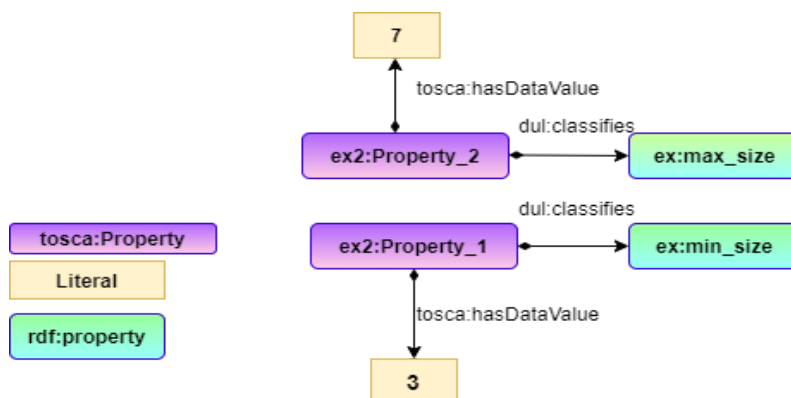


Figure 8. Example of property definition

Triggers:

The knowledge graph of the triggers of the autoscale template is depicted in Figure 9. The trigger defines when the scaling policy will be initiated. The triggers contain complex nested structures such as extra conditions about when the policy will be triggered (`tosca:condition`), constraints for defining when not firing the trigger, the period (`tosca:Period`) in which the conditions to be evaluated. Also, which interface operations will be called (`tosca:call_operation`) are referenced within the trigger, and those scripts, performing the scaling, are defined within the operations of interface types. The triggers are instances of the `tosca:Trigger` concept, and all the information that is contained within the trigger, is captured as instances of `soda:SodaliteParameter` that classify (`dul:classifies`) the corresponding concept. This complex information can be represented because the captured knowledge is following the SODALITE ODP.



Figure 9. Example of trigger definition

Example B Policy Type (Tier 1)

This is the DSL of the policy type from which the autoscale policy template is instantiated. Since the knowledge graphs of types have already been presented in D3.1, they are skipped in this deliverable. In this policy type, it is defined which concepts to be monitored, and some constraints for their values. From this abstract policy type, the autoscale policy template, described above, is instantiated. It is worth noting that, by leveraging the OWL2 punning capabilities, the types are both classes and instances (as they have property assertions for the descriptions). Thus, in such a way, subsumption hierarchies can be modelled enabling the complex representation of the TOSCA standard. An example policy type in DSL format is depicted in Figure 10.

```
radon.policies.scaling.AutoScale:
  derived_from: tosca.policies.Scaling
  properties:
    min_size:
      type: integer
      description: 'The minimum number of instances'
      required: true
      status: 'supported'
      constraints:
        greater_or_equal: 1
    max_size:
      type: integer
      description: 'The maximum number of instances'
      required: true
      status: 'supported'
      constraints:
        greater_or_equal: 10
```

Figure 10. Example policy type

2.1.2 Topology

The topology conceptual model captures all the information of the AADM (metadata and templates). A Topology example is represented as a Knowledge Graph in Figure 11. This conceptual model has been updated by containing more descriptive information about the workspace (*soda:hasNamespace*), the file name of the model (*soda:hasName*), the DSL text (*soda:hasDSL* enabling the browsing of the models in IDE) etc. The full topology policy example is presented in the [Appendix](#).

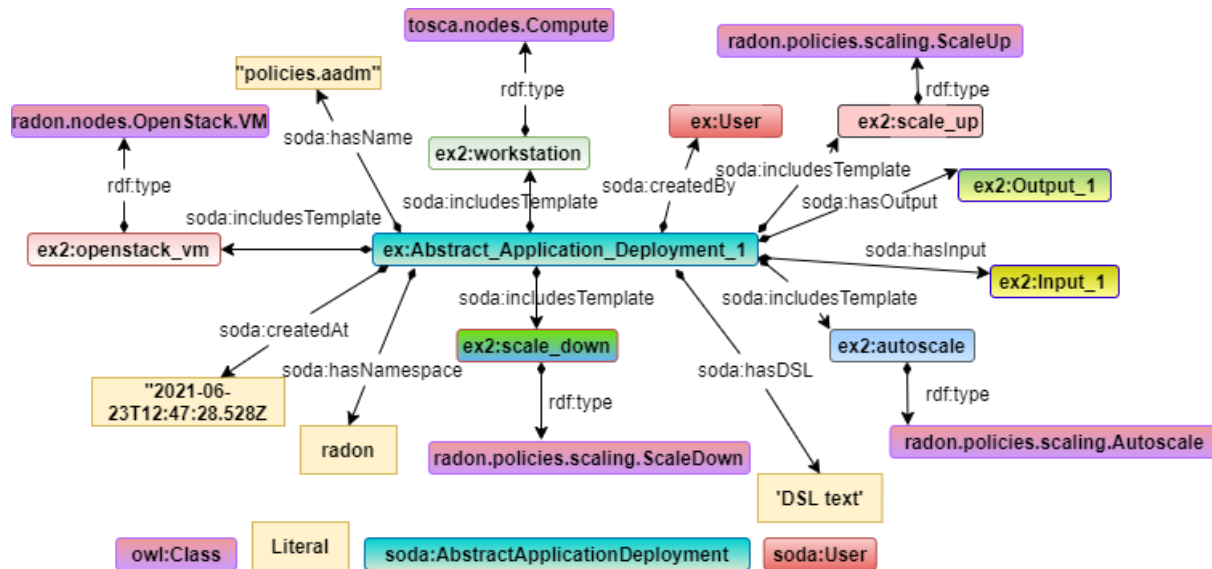


Figure 11. Example Topology Instance

2.1.3 Resource Model

After the first year of the project, the users can save resource models in the Knowledge Base by defining custom resources that extend the TOSCA normative types. The Resource Models share the same conceptual model with the Topology conceptual model. A Resource Model example is represented as a knowledge graph at Figure 12.

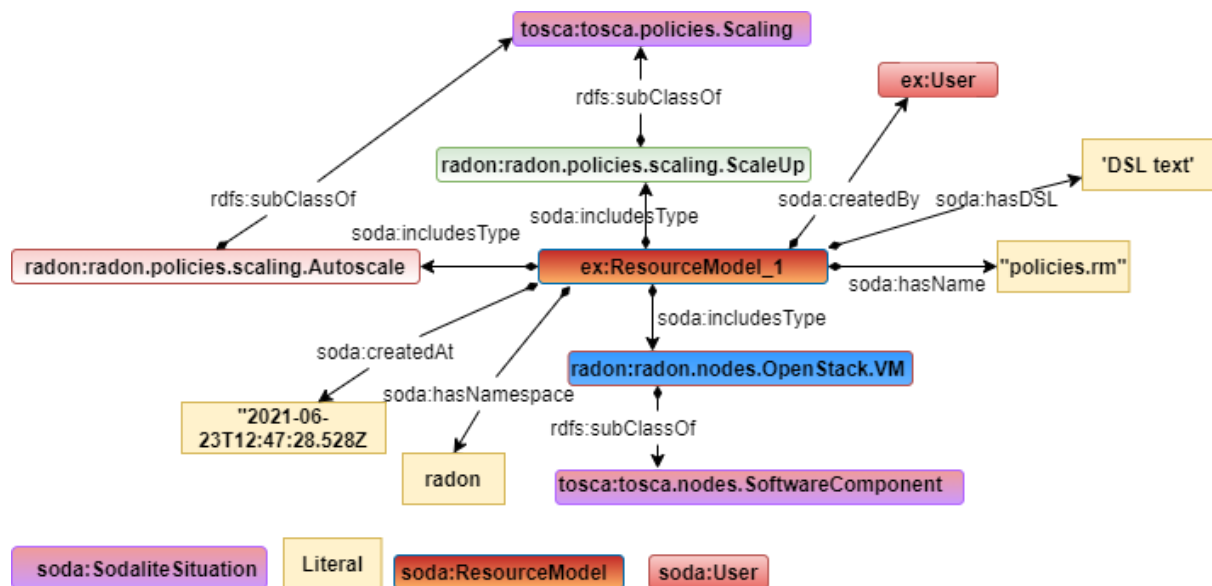


Figure 12. Example Resource Model

2.1.4 Inputs

As it is described in the introduction of the current section, the inputs allow the users to customize their inputs by not providing hardcoded values but input parameters. The inputs in TOSCA are used for passing information from the node templates through the inputs to the operation scripts that are used in the interfaces or generally to the properties of an application. In the inputs, the user can declare variables and their corresponding values. In Figure 13, an input example is

depicted that is used by a node template in Figure 15. In particular, the flavor name (*flavor-name*), the virtual machine's name (*vm-name*), and the name of the image (*image-name*) inputs are passed to the *openstack-vm* template through the properties. In Figure 14, it is depicted how the inputs are captured as knowledge graphs.

Example Input

```
inputs:
  flavor-name:
    type: string
  vm-name:
    type: string
  image-name:
    type: string
    default: "image name"
```

Figure 13. Inputs example

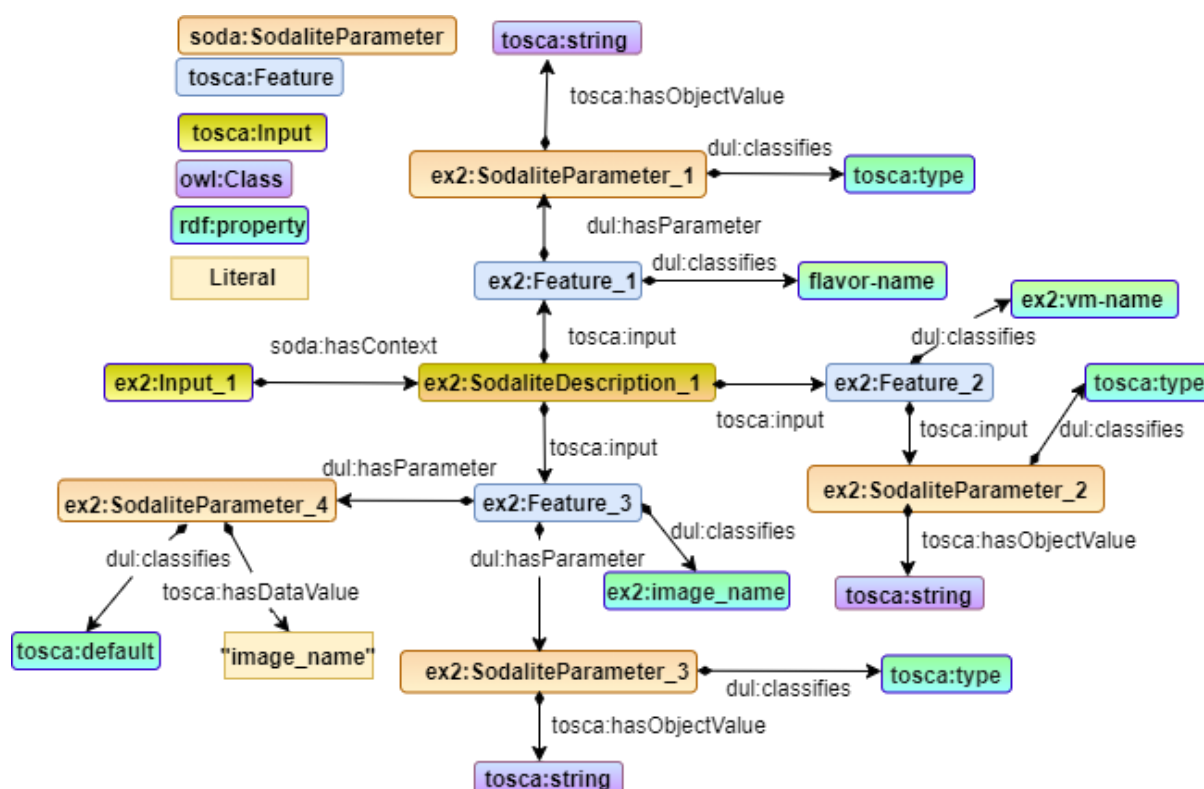


Figure 14. Inputs example knowledge graph

Node Template

```
node_templates:
  openstack_vm:
    type: radon/radon.nodes.OpenStack.VM
    properties:
      name: get_input: vm-name
      image: get_input: image-name
      flavor: get_input: flavor-name
```

Figure 15. A node template example that is using the inputs of Figure 13.

2.1.5 Outputs

Outputs are used for passing information that might describe the state of the deployed template to the user. In Figure 16, the ip address (*public_ip*), in which the compute node *workstation* has been provisioned, is provided in the outputs. The knowledge graph of the outputs of Figure 16 is depicted in Figure 17.

Example Output

```
outputs:
  public_ip:
    type: string
    description: 'The public IP of the provisioned VM'
    value:
      get_attribute:
        entity: radon/workstation
        attribute: radon/workstation.public_address
```

Figure 16. Output example

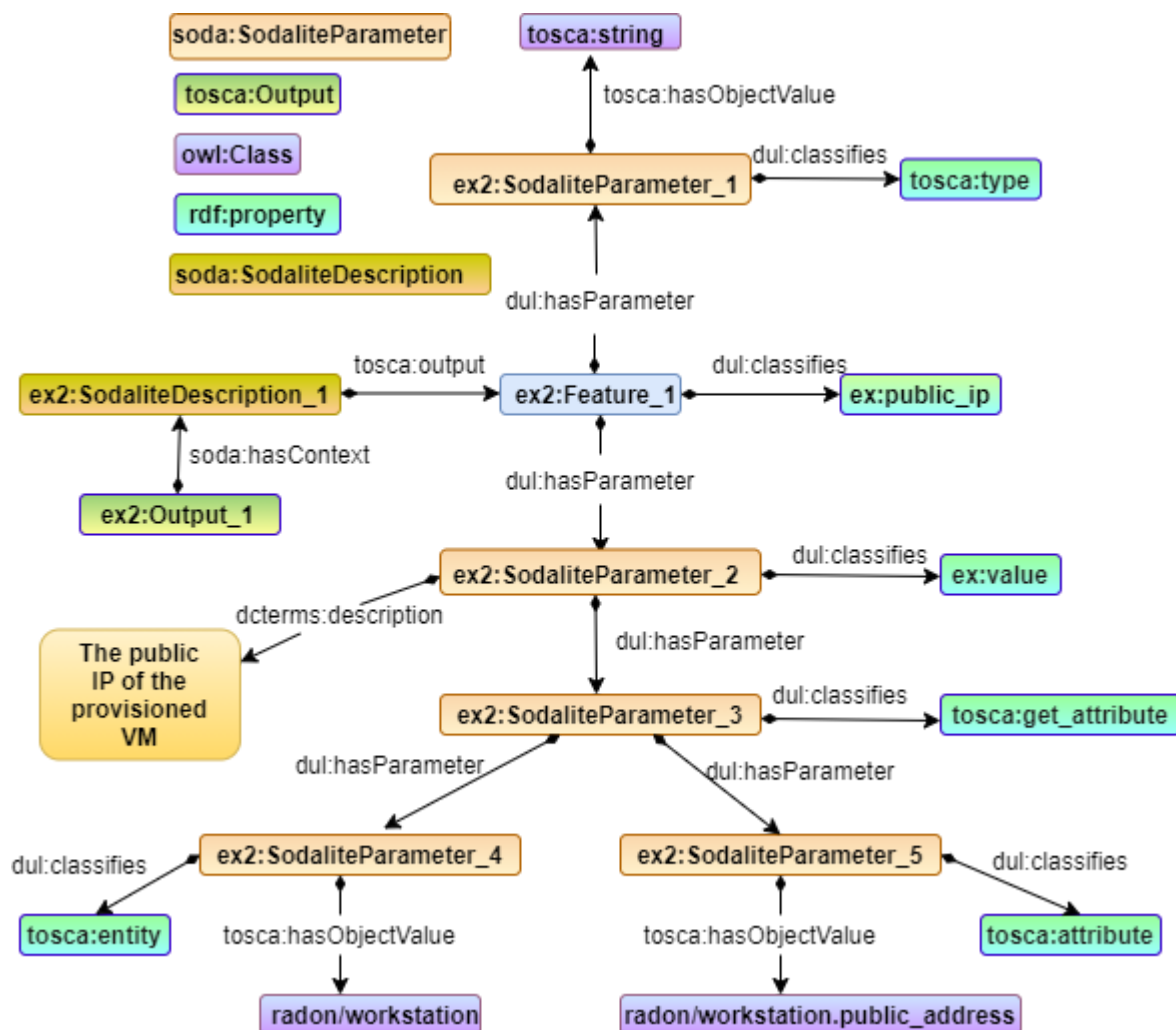


Figure 17. Output Knowledge Graph

2.1.5 Optimization

The optimization recipe that will be used by MODAK optimizer (more details in D4.2) is associated with a template. The DSL of a node template containing optimization is depicted in Figure 18, and its corresponding knowledge graph in Figure 19.

```
node_templates:
  optimization-skyline-extractor:
    type: docker/sodalite.nodes.DockerizedComponent
    optimization: ai_training.tensor_flow
    properties:
      ...
```

Figure 18. A node template related with an optimisation DSL script

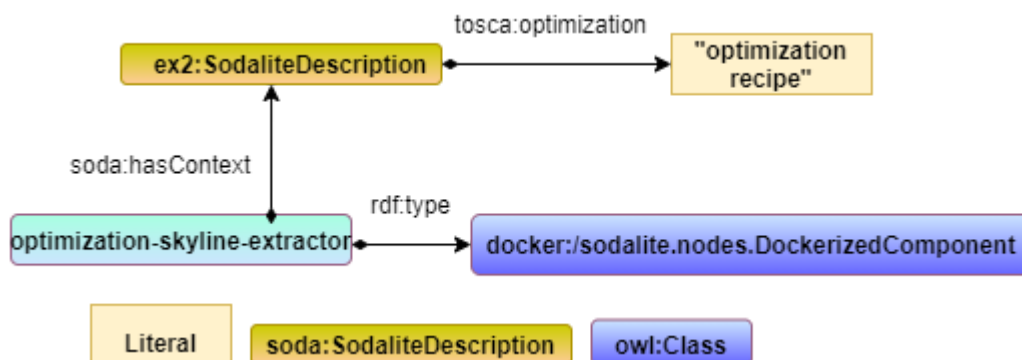


Figure 19. Optimization Knowledge Graph

2.1.5 KB architecture

The Knowledge Base is divided into a public workspace and private workspaces. Many users can share the same private workspace and in such a way a user can reuse resources and application components of other users. The users have access to a private workspace only when they have the corresponding permission. The public workspace accommodates all the TOSCA normative types and all the users have access to the public workspace.

From the implementation view, for supporting the workspaces, we leverage how the graph data set can be split. Namely, a graph dataset consists of a set of *named* graphs and the *default* graph.

A group of statements with a unique name (a URI) is called a 'named graph'. Instead of using one monolithic graph, it is desirable to have several multiple graphs so as to assign one workspace to each named graph. Each private workspace is saved in a different named graph. The public workspace is contained in the *default* graph. The end user can extend the specification by saving resource models and AADMs by creating a new graph or extending an existing graph. As it is depicted in Figure 20, the KB contains the default graph and multiple named graphs identified by a URI. For example, the workspace associated with the Snow use case belongs to the <https://www.sodalite.eu/ontologies/workspace/1/snow/> URI. The descriptive information of the topologies and the resource models are saved in the default graph, while the actual resources and application components are saved in the named graphs. In Figure 21, it is depicted how the parts of the models are distributed in the RDF graph dataset through the radon policy AADM example. The appendix includes the full Radon policy AADM. This AADM is composed of five templates that belong to the module *radon*. In Figure 21, we can observe that the templates per se are saved in the *radon* named graph, while the TOSCA normative types, the descriptive context of the AADMs/RMs (user, timestamp, version etc.) are saved in the default graph.

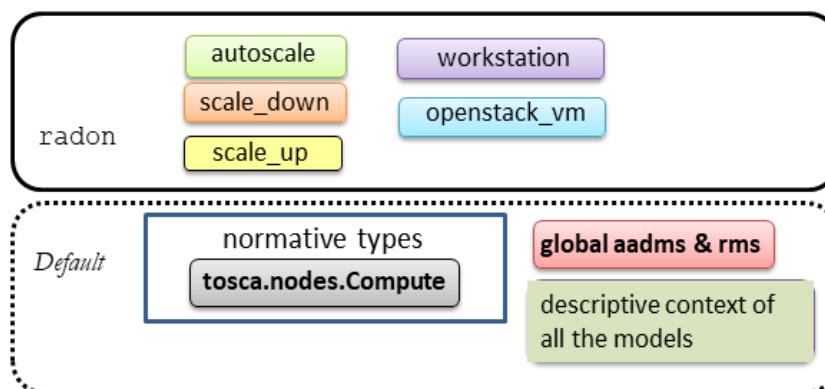


Figure 20. Graph dataset with one named graph and a default graph

The screenshot shows the 'Graphs overview' page in the GraphDB interface. On the left is a sidebar with navigation options: Import, Explore, Graphs overview (highlighted), Class hierarchy, Class relationships, Visual graph, Similarity, SPARQL, Monitor, Setup, and Help. The main area has a search bar and buttons for 'Export repository' and 'Clear repository'. Below these is a table of graphs:

<input type="checkbox"/>			Graphs
<input type="checkbox"/>			The default graph
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/docker/
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/openstack/
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/snow/
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/radon/
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/test/
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/batch/
<input type="checkbox"/>			https://www.sodalite.eu/ontologies/workspace/1/hpc/

Figure 21. The graphs of the KB

2.2 Evaluation of SODALITE Ontologies

The users have only direct interaction with the IDE for designing their models. The ontologies are the basis on which the Intelligence of the Modelling layer about designing IaC relies. In this section, we present how the ontologies have been evaluated from different perspectives. Namely, the consistency and the overall quality of the ontology were assessed by two well-known online tools that will be introduced in the next subsections, while the structure of the ontology is validated against some competency questions that are based on the use-cases.

2.1.1 Assessment

Indicative Competency Questions

We have defined a number of Competency Questions (CQs) for validating the ontologies by ensuring that they capture the semantics according to the SODALITE use-cases.

Table 1 reports on the competency questions that drive the development of the SODALITE ontological framework and their corresponding validation rule expressed in the SHACL[14] constraint language.

Competency Question	SHACL constraints
Does each AADM contain at least one template?	<pre>ex:AADMShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:AbstractApplicationDeployment ; sh:sparql [sh:message "AADM contains no template" ; sh:select """ SELECT distinct \$this { \$this a soda:AbstractApplicationDeployment . FILTER NOT EXISTS { \$this DUL:isSettingFor ?template. } } """ ;] .</pre>

<p>Does each RM contain at least one type?</p>	<pre> ex:RMShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:ResourceModel ; sh:sparql [sh:message "RM contains no type" ; sh:select "" SELECT distinct \$this { \$this a soda:ResourceModel . FILTER NOT EXISTS { \$this DUL:isSettingFor ?node. } } "" ;] . </pre>
<p>Does each AADM contain the required metadata?</p>	<pre> ex:AADMMetadataShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:AbstractApplicationDeployment ; sh:sparql [sh:message "AADM does not contain metadata"; sh:select "" SELECT distinct \$this ?time ?user ?name { \$this a soda:AbstractApplicationDeployment. FILTER NOT EXISTS { \$this soda:createdAt ?time . \$this soda:createdBy ?user . \$this soda:hasName ?name . } } group by \$this ?time ?user ?name"" ;] . </pre>
<p>Is each type associated with at least one concept?</p>	<pre> ex:TypeFullShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:SodaliteSituation ; sh:sparql [sh:message "Type contains no description" ; sh:select "" SELECT distinct \$this { ?RM a soda:ResourceModel . ?RM soda:includesType \$this. FILTER NOT EXISTS { </pre>

	<pre> \$this soda:hasContext ?context . } } "";] . </pre>
Is each template associated with at least one concept?	<pre> ex:TemplateFullShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:SodaliteSituation ; sh:sparql [sh:message "Template contains no description" ; sh:select "" SELECT distinct \$this { ?AADM a soda:AbstractApplicationDeployment . ?AADM soda:includesTemplate \$this. FILTER NOT EXISTS { \$this soda:hasContext ?context . } ""; }] . </pre>
Does each requirement include a node, and/or a capability and/or a relationship?	<pre> ex:RequirementAssignmentShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:SodaliteSituation ; sh:sparql [sh:message "Each requirement should include assignments only for node, capability, relationship and occurrences " ; sh:select ""select distinct ?v ?r_a ?r_i where { \$this soda:hasContext/tosca:requirements ?r . ?r DUL:classifies ?r_a. ?r DUL:hasParameter [DUL:classifies ?r_i; DUL:hasRegion ?v] . FILTER NOT EXISTS { FILTER(strends(str(?r_i), "node") strends(str(?r_i), "capability") strends(str(?r_i), "relationship") strends(str(?r_i), "occurrences")) } } """;] . </pre>

Is each SoftwareComponent template hosted in a node?	<pre> ex:SoftwareComponentShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass soda:SodaliteSituation ; sh:sparql [sh:message "A software component should always be hosted on a compute node" ; sh:select """select distinct ?host { FILTER NOT EXISTS { \$this a soda:SodaliteSituation ; a toska:tosca.nodes.SoftwareComponent; (soda:hasContext/tosca:requirements /tosca:hasObjectValue)* ?host . ?host a toska:tosca.nodes.Compute. } }""";]. </pre>
Does for each property that takes its value from an input, the corresponding input is defined?	<pre> ex:PropertyInputShape a sh:NodeShape ; sh:severity sh:fatalError ; sh:targetClass toska:Property ; sh:sparql [sh:message "If a property gets value from an input, that input should be present in the topology" ; sh:select """SELECT ?inputClass ?inputValue { #check properties getting value from input ?topology soda:includesTemplate ?template. ?template soda:hasContext/tosca:properties \$this . \$this DUL:hasParameter ?param . ?param DUL:classifies ?propClass . FILTER(strends(str(?propClass), "get_input")). ?param toska:hasDataValue ?inputValue . #Check inputs FILTER NOT EXISTS { ?topology soda:includesInput/soda:hasContext/ toska:input/DUL:classifies ?inputClass. FILTER(strends(str(?inputClass), ?inputValue)) } }""";]. </pre>

Table 1. Indicative CQs and SHACL-SPARQL translations

Quality Checking

We used the online tool OOPS!² (Ontology Pitfall Scanner) [15] for detecting the most common pitfalls in ontologies.

The pitfalls detected by OOPS are split into three categories:

- A. *Critical* : It is crucial to correct the pitfall. Otherwise, it could affect the consistency and the reasoning of the ontology.
- B. *Important* : Though not critical for ontology function, it is important to correct this type of pitfall.
- C. *Minor* : It is not a problem, but by correcting it we will improve the ontology.

Table 2 shows the pitfalls detected for the SODALITE ontology. Some pitfalls were also detected for the DOLCE imported ontology, but since this is an ontology that the SODALITE imports, we include only the SODALITE ontology pitfalls. No *critical* pitfall was detected for the SODALITE ontologies.

No	Pitfall	Importance	Results
1	Missing annotations: This pitfall consists in creating an ontology element and failing to provide human readable annotations attached to it. Consequently, ontology elements lack annotation properties that label them	Minor	107 cases
2	Missing domain or range in properties: The domain or range (or both) of a property (relationships and attributes) is defined by stating more than one <code>rdfs:domain</code> or <code>rdfs:range</code> statements.	Important	25 cases
3	Creating a property chain with just one property: The OWL 2 construct <code>owl:propertyChainAxiom</code> allows a property to be defined as the composition of several properties.	Minor	1 case
4	Inverse relationships not explicitly declared: This pitfall appears when any relationship (except for those that are defined as symmetric properties using <code>owl:SymmetricProperty</code>) does not have an inverse relationship (<code>owl:inverseOf</code>) defined within the ontology.	Minor	24 cases
5	Equivalent Classes not explicitly declared: This pitfall consists in missing the definition of equivalent classes (<code>owl:equivalentClass</code>) in case of duplicated concepts.	Important	3 cases
6	Untyped class: An ontology element	Important	2 cases

² <http://oops.linkeddata.es/>

	is used as a class without having been explicitly declared as such using the primitives owl:Class or rdfs:Class.		
--	--	--	--

Table 2. Ontology pitfalls detected by OOPS!.

Evaluation of the structure

Ontometrics³ [16] provides an online platform for ontology metrics calculation. It provides a web interface for uploading the ontologies in RDF/XML format. An extension of this tool is presented in [17], where the ontometrics is presented as Ontology Metrics as a Service, and most other tools are not usable anymore, and are mainly outdated.

We submitted the SODALITE ontology to the Ontometrics online platform for getting some advanced ontology metrics. In Table 3, and Table 4, a subset of the metrics calculated by Ontometrics is presented.

The results are divided into two sections, the *Base Metrics*, and the *Schema Metrics*. The Base Metrics are mainly about the count of properties, classes and other concepts so as the quantity of ontology elements to be measured. The DL expressivity, included in this section, indicates which variant of Description Logics is used. *SRIN(D)*⁴ indicates that it is an ontology with transitive, inversive properties, cardinality restrictions and limited complex role axioms. The Schema Metrics are about addressing design issues in the ontology such as the richness, inheritance, and the ratio. From those metrics that measure the richness of the ontology, it is concluded that the SODALITE ontology is an ontology that provides high domain coverage. It is worth to note the inheritance richness has a high value showing that the ontology can capture subsumption hierarchies that govern in the TOSCA metamodel.

Schema Metrics	Attribute richness	0.14
	Inheritance richness	1.45
	Relationship richness	0.37
	Axiom/class ratio	61.756
	Class/relation ratio	0.4397

Table 3. Schema metrics produced by the Ontometrics tool

³ <https://ontometrics.informatik.uni-rostock.de/>

⁴ https://handwiki.org/wiki/Description_logic

Base Metrics	Axioms	12166
	Class count	197
	Object property count	129
	Data property count	24
	Properties count	153
	DL expressivity	SRIN(D)
	SubClassOf axioms count	283
	Equivalent classes axioms count	10
	Disjoint classes axioms count	26
	SubObjectPropertyOf axioms count	70
	Inverse object properties axioms count	55
	SubPropertyChainOf axioms count	1

Table 4. Base metrics produced by the Ontometrics tool

2.2.2 Comparison with other ontologies

Our SODALITE ontology is based on the TOSCA standard since it offers interoperability. Also, TOSCA permits you to create your own custom types allowing, in such a way, to model any cloud model definition, namely XaaS. Additionally, the ontologies capture both the functional and non-functional requirements. The functional requirements are captured through properties, requirements and other concepts, and the non-functional requirements are captured through the policies. Notwithstanding, there is a surge in modelling cloud environments in ontologies, little focus has been given on building an expandable and modular ontology that can represent complex concepts such as the TOSCA standard. As analyzed in D3.1, our SODALITE ontology leverages the best practices in Ontology Engineering, namely the Ontology Design Patterns, the expressiveness of OWL2. In this subsection, we will compare the modelling capabilities of the SODALITE ontology with other ontologies in the cloud computing domain.

A description of a cloud service is composed of both functional and non-functional features. The functional features represent tasks that make the cloud service functional as storage, network, host etc. The non-functional features represent the quality of the aforementioned tasks such as scaling, cost and QoS properties. Also, there are three types of cloud computing: IaaS, SaaS and PaaS. The current cloud computing ontologies cover some types/one type of the cloud computing, and partially the functional and non-functional features. Also, it is essential, an ontology not to be

monolithic, but expandable, being able to represent many layers in a modular and reusable manner. A comparison of the most important cloud computing ontologies, based on the aforementioned aspects, has been performed, and its results are depicted in Table 5.

The CoCoon [18] is an OWL2 ontology that describes IaaS services by focusing on non-functional features. Only the prices and QoS characteristics are modelled, thus the non-functional features are partially captured.

The Cloud FNF [19] is a unified ontology that represents both functional and non-functional features in all types of cloud services; despite this ontology covers many concepts, it defines all the concepts in classes explicitly without following any design pattern for offering flexibility.

The TOSCA ontology [20] is designed to fill the gap between the structural aspect of the TOSCA and the domain of applications, however, the structural relationships are captured rather than an abstracted model. As this ontology represents the TOSCA standard, it models all the types of cloud services and captures the non-functional features through the TOSCA policies.

The mOSAIC [21] project defines a cloud ontology capturing all types of cloud services (IaaS, SaaS, PaaS) and both functional and non-functional features; However, it does not represent an abstract model. The non functional features include various concepts such as scalability, availability, QoS, Performance.

The PaaSPort [22] ontology defines an ontology that captures PaaS cloud services. This ontology is modular and expandable by using the same upper level ontology as the SODALITE ontology, namely the DnS ontology. Also, some of the OASIS standards were used for designing the ontology. In [23], a three standard ontology is proposed used within a semantic framework for promoting interoperability across IaaS cloud resources. The framework aims at representing a unified model for representing three different well-known standards (TOSCA, CIMI, and OCCI) and the transformation between them. The representation focuses only on the functional features. This ontology enables interoperability by supporting three different standards; however, it is not based on an ODP for promoting the expandability.

Ontology	IaaS	SaaS	PaaS	Functional features	Non-functional features	Expandability
Mosaic	Y	Y	Y	Y	Y	N
CloudFNF	Y	Y	Y	Y	Y	N
TOSCA	Y	Y	Y	Y	Y	N
PaaSPort	N	N	Y	N	N	Y
CoCoon	Y	N	N	Y	N	N
Three standards ontology	Y	N	N	Y	P	N
SODALITE	Y	Y	Y	Y	Y	Y

Table 5. Comparison of the modelling capabilities of cloud computing ontologies

Y= Yes, N = No, P = Partially

3 Updated Ontology Population and Checking

As it was described in D3.1, the intermediate Exchange Model is a lightweight version of the SODALITE ODP and was created for hiding the complexity of the KB conceptual model. WP3 is responsible for mapping this exchange model to the SODALITE ontologies, and checking the consistency of the KB in terms of the native OWL2 semantics. The user authors their models in the SODALITE DSL language, and the IDE sends them in the exchange format to the SPE module. The SPE module that offers all the mapping services for mapping the DSL models to the ontologies. The mapping services have been significantly enhanced for mapping all the new concepts and kinds of models. In this section, we will provide two examples containing various new concepts that have been introduced in the exchange model.

Policy template Example

In Figure 22, an excerpt from a policy template in DSL is shown with two new concepts that were introduced, the targets and the triggers.

```
autoscale:
  type: radon.policies.scaling.AutoScale
  targets: [openstack_vm]
  triggers:
    radon.triggers.scaling:
      description: 'A trigger for autoscaling'
      event: 'auto_scale_trigger'
      target_filter:
        node: openstack_vm
```

Figure 22. Excerpt from a Policy Template

The representation of the above template in the exchange format (Turtle syntax⁵) is as follows:

```
:PolicyTemplate_3
  rdf:type exchange:Template ;
  exchange:name "autoscale" ;
  exchange:type 'radon/radon.policies.scaling.AutoScale' ;
  exchange:targets :Parameter_26 ;
  exchange:triggers :Trigger_1 ;
.
:Parameter_26
  rdf:type exchange:Parameter ;
  exchange:listValue 'radon/openstack_vm' ;
.
:Trigger_1
  rdf:type exchange:Trigger ;
  exchange:name "radon.triggers.scaling" ;
  exchange:description 'A trigger for autoscaling' ;
  exchange:hasParameter :Parameter_1 ;
```

⁵ [https://en.wikipedia.org/wiki/Turtle_\(syntax\)](https://en.wikipedia.org/wiki/Turtle_(syntax))

```
exchange:hasParameter:Parameter_5 ;
.
:Parameter_1
  rdf:type exchange:Parameter ;
  exchange:name "event" ;
  exchange:value 'auto_scale_trigger' ;
.
:Parameter_2
  rdf:type exchange:Parameter ;
  exchange:name "node" ;
  exchange:value 'radon/openstack_vm' ;
.
:Parameter_5
  rdf:type exchange:Parameter ;
  exchange:name "target_filter" ;
  exchange:hasParameter:Parameter_2 ;
.
```

Versioning

As it is mentioned in subsection 7.1.2.1, versioning is supported in the SODALITE platform so that the users are able to define different versions per each AADM. As such, each AADM can refer to versioned templates saved in other AADMs. For example, in Figure 23, the *snow-mysql* template is hosted in a *snow-docker-host* component of version 1.0 that is saved in an AADM with version 1.0 in the *snow* module.

```
snow-mysql:
  type: docker/sodalite.nodes.DockerizedComponent
  requirements:
    host:
      node: snow/snow-docker-host@v1.0
```

Figure 23. Template example with reference to versioned node

The representation of the above template in the exchange format (Turtle syntax) is as follows:

```
:Template_1
  rdf:type exchange:Template ;
  exchange:name "snow-mysql" ;
  exchange:type 'docker/sodalite.nodes.DockerizedComponent' ;
  exchange:requirements:Requirement_1 ;
.
:Parameter_1
  rdf:type exchange:Parameter ;
  exchange:name "node" ;
  exchange:value 'snow/snow-docker-host@v1.0' ;
```



.

```
:Requirement_1
  rdf:type exchange:Requirement ;
  exchange:name "host" ;
  exchange:hasParameter :Parameter_1 ;
```

.

4 Advanced Reasoning services

This section analyzes the intelligent services provided to the IDE user based on the strong inference capabilities offered by the KB for uncovering information out of the existing relations. The models are saved in the KB as RDF interconnected graphs capturing all the structures and the relationships in a formal structure that enable the retrieval and the reuse of the knowledge in an unambiguous manner. Those services include context-aware content search, reuse, matchmaking, validation for detecting inconsistencies, and abstraction of the model.

1. Context-aware content search

The modellers can get suggestions for auto-completing their models with knowledge that is already saved in KB, for instance:

- Property, capability, interface, requirement, attribute, operation and trigger names can be proposed in template assignments according to its corresponding type definition.
- All the workspaces that are available
- Optimizations

2. Matchmaking and reuse

All the resources and application nodes are saved in KB fostering the reuse as there is a central point with all the resources and nodes that can be referenced in other models, significantly reducing the amount of work needed by the user.

- Matching and reuse nodes that can satisfy requirement assignments of a template. For instance, virtual machines that can serve as a host of an application, dependent components such as a database, or network to which the application will be connected.
- Getting templates that are saved in specific workspaces.

3. Validation

Validation is a crucial part of the smart reasoning processes by minimizing the IaC errors during the design process and obviating the detection of those errors during the deployment time. Henceforth, the modellers save time and effort. Validation is based on:

- Constraints of properties
- Property definitions
- Requirement definitions

4. Abstraction of DSL

One of the most fundamental kinds of smartness in the SODALITE platform is the abstracted DSL. Information can be omitted in the model, and the intelligent reasoning services will autofill the model. Precisely, the user can omit, for instance, where an application can be hosted, which database will be used, or to which network be connected. During the design time, the reasoner can detect which required or optional requirements are missing, and inform the user which templates can satisfy those requirements by a suggestion (for optional requirement) or an error (for mandatory requirement). During the deployment time, the model is concretized by the reasoner when a required/optional requirement is missing.

5. Optimization suggestions

Application performance is statically optimized by MODAK (D4.2). MODAK is enabled when an Optimization DSL (D3.4) is provided and associated with an AADM. Within the optimization DSL, the user can choose the configuration of the application before the deployment so as to achieve better performance. Cloud and High Performance computing is executed on an architecture consisting of diverse execution platforms that make it complex for the user to harness the power of the architecture by selecting the optimal settings of the application. With ontological reasoning, based on the capabilities that are already declared on the AADM, the optimal settings of the application can be suggested to the user.

5 SODALITE IDE

In the reporting period, a number of IDE features have been either extended or newly implemented. In the following, a functional description of these features is provided.

5.1 Domain Specific Languages

The IDE enables different SODALITE roles in the specification of infrastructure resources and the topology of the deployment of their applications, by adopting SODALITE domain specific languages:

- Resource DSL: adopted by Resource Experts (REs) for the specification of types of resources in heterogeneous infrastructures, such as Cloud, HPC or Edge.
- Abstract Application Deployment DSL: adopted by Application Ops Experts (AoEs) for the specification of application deployment topologies.
- Optimization DSL: adopted by the Quality Experts (QEs) for the specification of strategies for the optimization application components deployed as Artificial Intelligence (AI) - kind, Cloud-kind or HPC-kind components.
- Ansible DSL: adopted by either REs or AoEs for the specification of the implementation of operations associated to the interface life-cycle of both infrastructure resources and application components.
- Alerting Rule DSL: adopted by QEs for the specification of rules that govern the triggering of alerts upon the detection of conditions on monitored deployed applications.

Model instances of these metamodels are created by SODALITE roles through the IDE:

- Resource models (RMs) are instance models of the Resource DSL that contain types of infrastructure resources,
- Abstract Application deployment models (AADMs) are instance models of the Abstract Application Deployment DSL, containing the deployment topology of an application,
- Optimization models (OMs) are instances of the Optimization DSL, containing optimizations for application components deployed for target execution environments,
- Ansible models (AMs) are instances of the Ansible DSL, containing the implementation of operations for the life-cycle of application components,
- Alerting rule models (ARMs) are instances of the Alerting Rule DSL, containing the rules that govern the triggering of alerts upon the detection of monitoring conditions in deployed application components.

Resource DSL, Abstract Application Deployment DSL and Optimization DSL were initially implemented in the previous reporting period and reported in D3.1. In this reporting period, Resource DSL and Abstract Application Deployment DSL have been largely improved. Optimization

DSL, Ansible DSL and Alerting Rule DSL have been designed and implemented during this reporting period. In the following we describe the main new characteristics of these DSLs

5.1.1 AADMs

The Abstract Application Deployment DSL has been extended since last M18 specification with a number of features:

- Modules: applications can be defined within a module (i.e. namespace) to avoid name collision between components defined for multiple applications stored in the KB. Similarly, AADMs can import modules existing in the KB, so that the resources defined within those imported modules get available for reuse therein.
- Policies: TOSCA based policies can be also defined and associated to application components
- Inputs/outputs: the inputs required by the application and the outputs it produces can also be declared

AADMs reuse define infrastructure types declared within RMs as well as infrastructure resource instances available in the KB. The Abstract Application Deployment DSL enables AoEs to focus on the specification of an application's components and its deployment topology, while relying on infrastructure's resource types defined elsewhere by REs (and stored in the KB for reuse).

5.1.2 RMs

The Resource DSL has been extended since last M18 specification with a number of features:

- Modules: resources can be defined within a module to avoid name collision between resources defined for multiple infrastructures stored in the KB. Similarly, RMs can import modules existing in the KB, so that the resources defined within those imported modules get available for reuse therein.
- Types: specification of other types are now supported, including data, artifacts, capabilities, interfaces, relationships and policies.

The Resource DSL enables REs to focus on the specification of infrastructure resource types in RMs. They can also specialize or reference other infrastructure resource types available in the KB. This separation of modeling concerns between application components (in AADMs) and reusable infrastructure resources (in RMs) largely simplifies the modeling process and enables modelling the specialization of AoE and RE roles.

5.1.3 Optimization Models

The Optimization DSL was created based on the optimization specification provided in D4.2. An OM specifies the:

- kind of application to optimize (e.g. AI_Training, HPC, etc),
- optimization build configuration, declaring the type of CPU or accelerator,
- autotuning configuration, and the
- optimization configuration for the kind of application to optimize, including (application kind specific):
 - application kind configuration,
 - associated data
 - runtime frameworks for application kind (e.g. TensorFlow or Keras for AI_Training kind)

Optimization models can be bound to AADM for the optimization of concrete application components.

5.1.4 Ansible Models

The IDE also supports the specification of the operation associated with resources in an RM/AADM. This specification is supported by a simple DSL from which Ansible blueprints are automatically generated. An initial version of this DSL, which we call Ansible DSL, has been reported in D4.2 released at M24. A new version is being consolidated and will be reported in D4.3. This work, in fact, is at the intersection between WP3 and WP4.

5.1.5 Alerting Rule Models

The Alerting Rule DSL was created based on Prometheus PromQL specification⁶. This language enables QEs to define conditions around monitoring metrics that, when held for some given duration⁷, an alert with some given associated data⁸ should be triggered, and dispatched to registered observers (e.g. refactoring). Each alerting rule model consists of one or more rules, organised in groups. Each rule consists of an expression, formalized in PromQL, which describes the condition, expressed as a boolean-evaluated expression, that has to be held during a given time duration (expressed in the for attribute) to trigger the alert. The expression consists of a combination of monitoring metrics, processed by functions, aggregation functions and filters. The severity label attribute specifies the severity associated with the triggered alert, and it is interpreted by the observer that captures it. Additional data to be shipped within the alert can be encoded in one or more annotations.

Alerting rule models can be registered in the monitoring system from the IDE.

5.2 New and extended features (M18 - M33)

During this reporting period, a number of features have been improved and other new ones implemented. In the following, a functional description of these features is provided.

5.2.1 Multiview representation of the AADM

Primary intention of SODALITE Abstract Application Deployment DSL is to support AoEs in the specification of their complex application deployment topologies, maximizing their modeling productivity. To address this high modelling productivity, a textual based AASM DSL edition was conceived (see [Figure 24](#)). This approach offers high flexibility and productivity on one side, but on the other side, textual based edition requires high specialization on the underlying DSL notation and semantics, and lacks adequate expressivity for communication.

⁶ <https://prometheus.io/docs/prometheus/latest/querying/basics/>

⁷ This duration is specified by the rule designer as part of the rule specification

⁸ This data is also specified by the rule designer, and may include concrete monitoring data to ship within the alert.

```
radon.aadm ⌵
node_templates:
  workstation:
    type: tosa.nodes.Compute
    description: "workstation description"
    attributes:
      private_address: "localhost"
      public_address: "localhost"

  openstack_vm:
    type: radon/radon.nodes.OpenStack.VM
    properties:
      name: "HostVM"
      image: "centos7"
      flavor: 'ml.xsmall'
      network: 'provider_64_net'
      key_name: 'my_key'
    requirements:
      host:
        node: radon/workstation

policies:
  scale_down:
    type: radon/radon.policies.scaling.ScaleDown
    description: "scale down policy description"
    properties:
      cpu_upper_bound: 90
      adjustment: 1

  scale_up:
    type: radon/radon.policies.scaling.ScaleUp
    properties:
      cpu_upper_bound: 90
      adjustment: 1
```

Figure 24. Text-based representation of AADMs

To address these limitations in textual edition, SODALITE IDE automatically generates two visual representations of textual representation, while the user is editing the textual model:

- Tree-based outline representation (see [Figure 25](#)). This read-only representation displays the main AADM elements in a tree-based widget, offering an in-a-glimpse representation of the AADM. This outline is synchronized with the textual editor, so changes in the editor are immediately reflected in the outline. Moreover, by selecting one element in the outline, the corresponding element is selected in the textual editor, and vice-versa. This representation is particularly useful for looking for concrete components of a large and complex application.

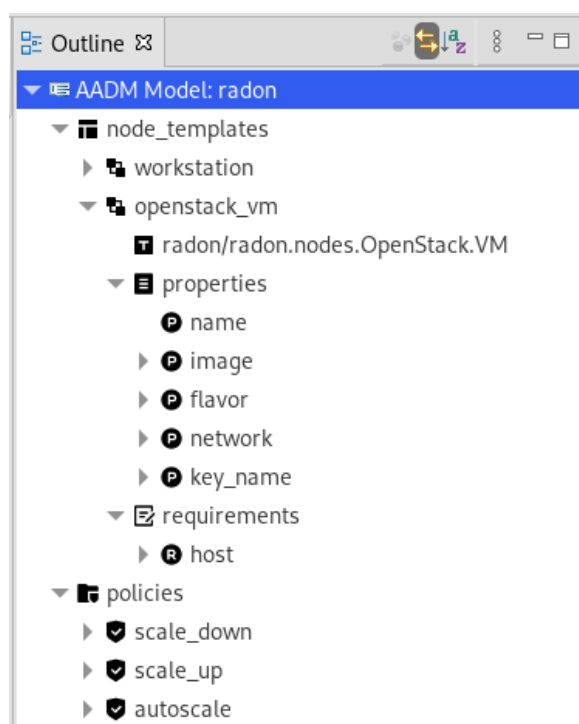


Figure 25: Outline tree-based representation of AADMs

- Canvas-based visual representation of AADMs (see [Figure 26](#)). This representation shows AADM entities represented as blocks (with different shape and color depending on their kind) connected to each other according to the associations defined in the DSL. This representation is editable, so pre-existing entities can be edited through their associated form in properties view (see [Figure 27](#)). Moreover, new entities, such as components (i.e. node types) can be inserted in the model by selecting them from the editor's palette (see right-side panel in [Figure 26](#)) and dropping them into the canvas. Textual and graphical representations are synchronized upon saving, so modifications in one representation are reflected in the another. When this visual representation is selected, the outline view offers a thumbnail visualization of the entire AADM visual representation, which can be used to frame the visual canvas in the part of the AADM the user is interested in.

The combination of outline tree-based representation and canvas-based visual representation of AADMs largely improve the expressivity for communication of the IDE. This is particularly helpful for large and complex app deployment topologies. The visual editor is also offering a content assistance in the associated form-based editors that is similar to the one supported by the textual editor (see section *Improved Content Assistance* below) .

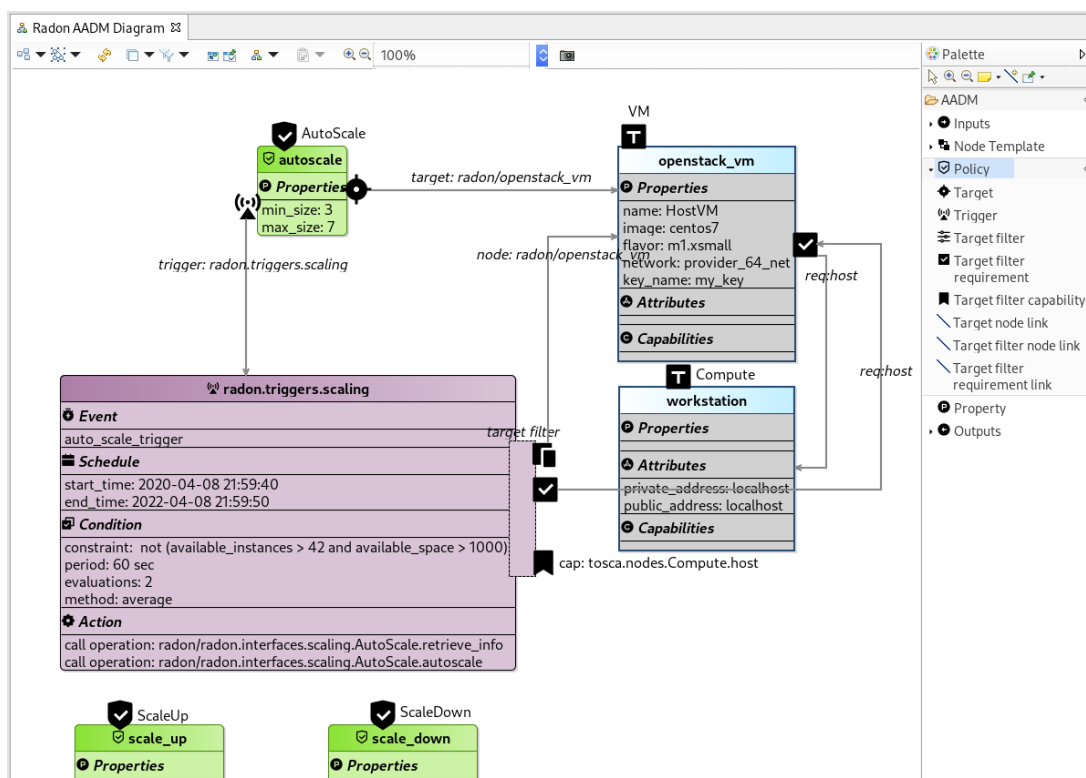


Figure 26: Graphical-based representation of AADMs

Figure 27: Form-based edition of AADM entities

5.2.2 Extended AADM Deployment Support

The deployment of AADMs has been largely improved since the last reporting period, both in terms of the IDE wizard-based assistance and reporting, as well as on the backend end-to-end deployment process. AADM deployment wizard (see [Figure 28](#)) can be triggered from the contextual menu of the selected AADM in the *Project Explorer* view.

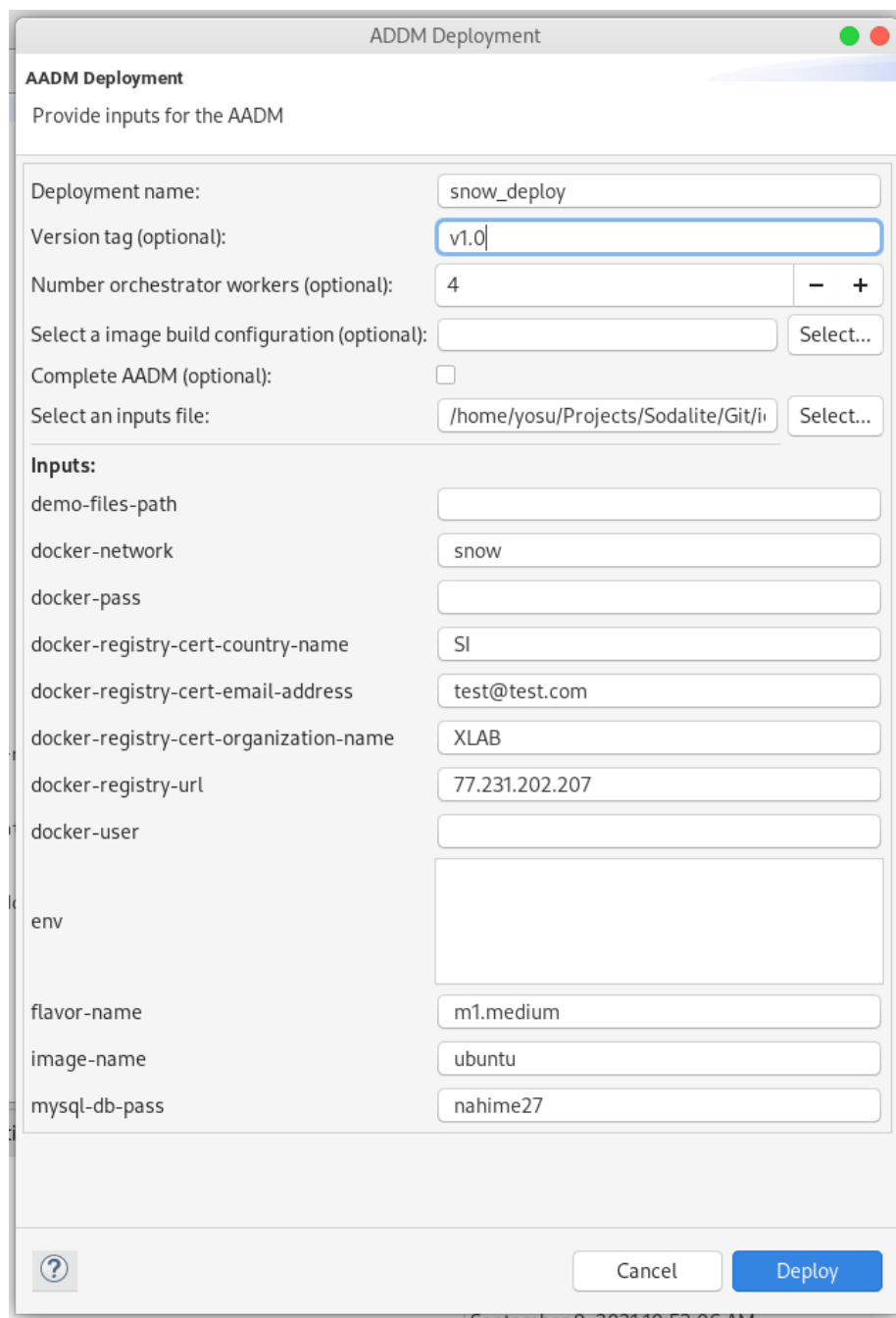


Figure 28: AADM deployment wizard

This wizard prompts AoEs to provide mandatory fields such as a deployment name and values for inputs declared within the AADM. Optionally, AoEs can specify the number of workers to use for parallel deployment (see D5.2) and check whether or not the AADM should be completed by the KB Reasoner, in case the AADM is abstract as it does not resolve all component requirements. If

marked, KB tries to resolve missing requirements with resources registered in the KB. If the complete resolution is not possible, the user is notified and the deployment process is cancelled. Another optional feature is the possibility to associate an *ImageBuilder* descriptor to build the images required by the AADM deployment (see D5.2) before this one takes place.

In terms of the completion of the backend deployment process, this can be extended to include all these steps:

- The AADM is saved into the KB. During this step semantic validation is conducted. If errors are detected, they are reported back to the user and the deployment is cancelled.
- A JSON representation of the AADM is retrieved from the KB and sent to the IaCBuilder for blueprint generation.
- The *Orchestrator* is requested to deploy the AADM from the generated blueprint. This process concludes when the orchestrator reports to the IDE the final status. If deployment fails the user is notified and the deployment aborted. If it succeeds, the deployment process continues.
- The *Monitoring* is notified to register the dashboards associated with this deployment.
- The *Refactoring* is also notified to register this deployment for supervision.

After a successful deployment the user is notified, and deployment details are available after refreshing the *Deployment Governance* view.

5.2.3 Deployment Governance View

This view (see [Figure 29](#)) enables AoEs to browse and manage their deployed applications. This view shows the complete list of active deployments grouped by blueprint (i.e. the application deployment snapshot). AoEs can refresh the view after triggering new deployments by using the view top level left toolbar, or by using a similar contextual popup menu for any row selected in the view table. For each selected deployment, AoEs can either resume them (in case of failed or undeploy applications) or undeploy them (in case of successfully deployed ones).

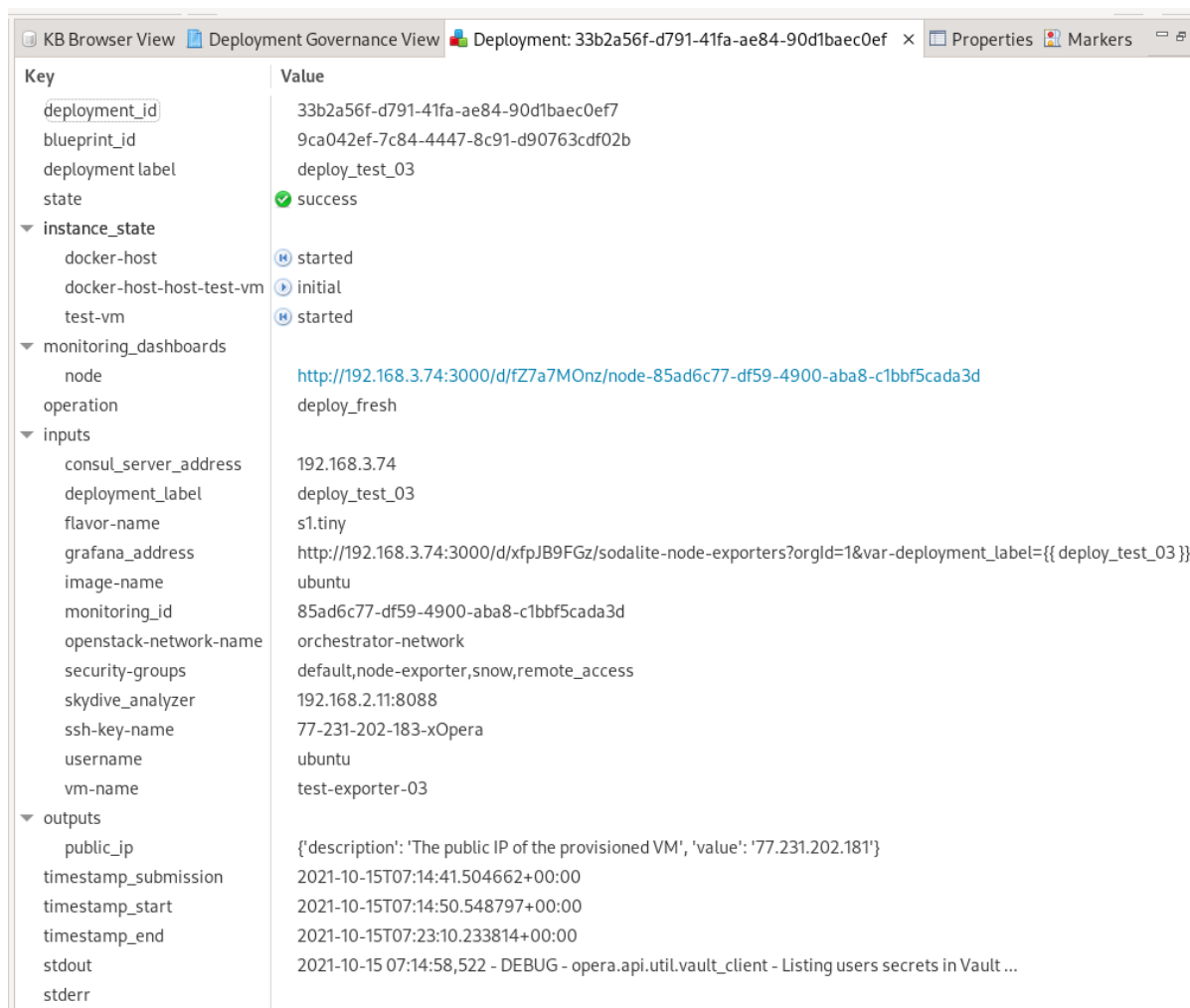
Id	Name	AADM id	Module	URL	Timestamp	State	Version id
a73275e7-470d-46af-a30f-fe20	test.aadm	AADM_fgcn9flmo0sfi6fc6b5	test	https://gitlab.com/sodalite.xopera/gitDB	2021-09-14 07:08:25	-	v1.0
e2c1e34c-1016-46ba-9855-cc74	test.aadm	AADM_fgcn9flmo0sfi6fc6b5	test	https://gitlab.com/sodalite.xopera/gitDB	2021-09-14 09:22:51	-	v1.0
de174390-6d05-4c43-8796-ect	test.aadm	AADM_fgcn9flmo0sfi6fc6b5	test	https://gitlab.com/sodalite.xopera/gitDB	2021-09-14 12:01:20	-	v1.0
a56aafbe-950e-49cd-98f8-6	deploy_test12	-	-	-	2021-09-14 12:01:21	success	-
33fdceb9-c148-4506-b767-027	test.aadm	AADM_fgcn9flmo0sfi6fc6b5	test	https://gitlab.com/sodalite.xopera/gitDB	2021-09-14 12:01:28	-	v1.0
d11922da-0c87-4534-884e-3	deploy_test13	-	-	-	2021-09-14 12:08:18	success	-
809e5f51-b8e9-4089-a106-347	test.aadm	AADM_fgcn9flmo0sfi6fc6b5	test	https://gitlab.com/sodalite.xopera/gitDB	2021-09-14 13:57:14	-	v1.0
df29a828-b97a-429c-b24e-2	deploy_test14	-	-	-	2021-09-15 07:21:11	failed	-

Figure 29: Deployment Governance View

AoEs can also inspect the details of one deployment by double clicking on them. This opens a detailed representation of the deployment in a new (see [Figure 30](#)). This view offers details of the deployment, such as:

- Unique identifiers for the deployment and associated blueprint
- State of deployment and allocated instances
- The endpoints for monitoring dashboards associated with this deployment. By clicking on these endpoints, they are opened into the default system browser, where the AoEs can inspect the application's component runtime behavior
- Input values provided to the application deployment
- Output values provided by the *Orchestrator*

- Log output provided by the Orchestrator during the deployment. By double clicking on it, a popup dialog shows the last log lines⁹



Key	Value
deployment_id	33b2a56f-d791-41fa-ae84-90d1baec0ef7
blueprint_id	9ca042ef-7c84-4447-8c91-d90763cdf02b
deployment_label	deploy_test_03
state	success
instance_state	
docker-host	started
docker-host-host-test-vm	initial
test-vm	started
monitoring_dashboards	
node	http://192.168.3.74:3000/d/fZ7a7MOnz/node-85ad6c77-df59-4900-aba8-c1bbf5cada3d
operation	deploy_fresh
inputs	
consul_server_address	192.168.3.74
deployment_label	deploy_test_03
flavor-name	s1.tiny
grafana_address	http://192.168.3.74:3000/d/xfpJB9FGz/sodalite-node-exporters?orgId=1&var-deployment_label={{ deploy_test_03 }}
image-name	ubuntu
monitoring_id	85ad6c77-df59-4900-aba8-c1bbf5cada3d
openstack-network-name	orchestrator-network
security-groups	default,node-exporter,snow,remote_access
skydive_analyzer	192.168.2.11:8088
ssh-key-name	77-231-202-183-xOpera
username	ubuntu
vm-name	test-exporter-03
outputs	
public_ip	{'description': 'The public IP of the provisioned VM', 'value': '77.231.202.181'}
timestamp_submission	2021-10-15T07:14:41.504662+00:00
timestamp_start	2021-10-15T07:14:50.548797+00:00
timestamp_end	2021-10-15T07:23:10.233814+00:00
stdout	2021-10-15 07:14:58,522 - DEBUG - opera.api.util.vault_client - Listing users secrets in Vault ...
stderr	

Figure 30: Deployment Details View

AoEs can also select a failed or undeployed deployment to resume it (from last failing node or from scratch, respectively), by selecting the corresponding menu entry in the top left view toolbar or contextual popup menu. Then, a wizard (see [Figure 31](#)) prompts the AoEs for input values for resuming the deployment and the number of workers for parallel deployment). Similarly, AoEs can request as well to undeploy a successfully deployed application.

⁹ Up to the last 1500 lines are displayed and not all them due to limitations in the IDE graphical subsystem

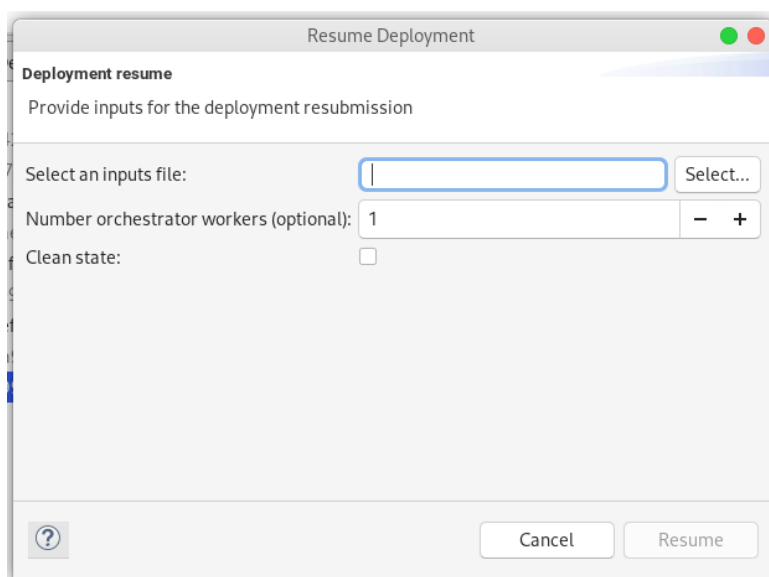


Figure 31: Deployment Resume wizard

5.2.4 KB Browser view

Different REs and AOEes can contribute to the common, shareable KB with RMs and AADMS describing infrastructure resource types and instances that could be reused by others. These models can be authored either by individuals or by collaborative teams. Therefore, the models stored in the KB should be accessible for all those who have read permissions for the modules those models were declared in. To browse, retrieve or delete models stored in the KB, the IDE offers the KB Browser view (see [Figure 32](#)).

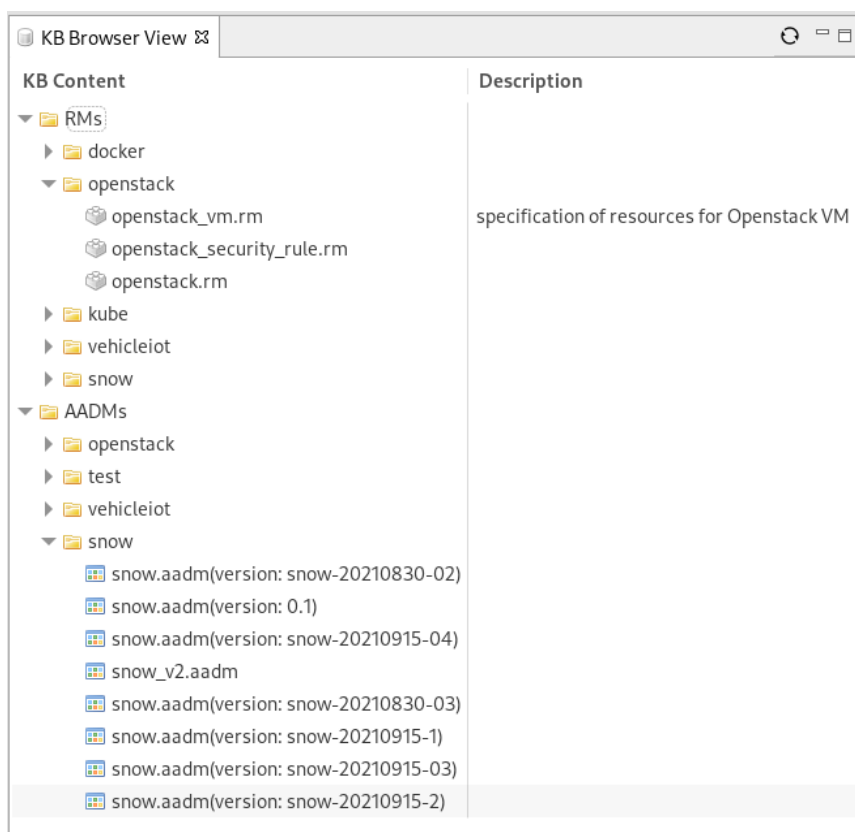


Figure 32: KB Browser view

This view shows a tree-based representation of the KB content, organized by model kind (RMs, AADMs) and the nested modules (e.g. docker, openstack, etc). Only models in modules for which the user has read permissions in IAM are included in the view. This KB content can be refreshed from the *refresh KB* button in the top-left toolbar, or from the *refresh KB* contextual popup menu associated with any entry in the tree. For each selected model, this popup menu permits the user to retrieve the model into her workspace or delete it from the KB. Similar functionality is available for complete modules. In this latter case, all models in the module are retrieved into the selected target folder within the workspace, or deleted, respectively. Users can only delete models for which they have write permissions associated with the module in her IAM configuration. A retrieved model can be opened, from the Project Explorer view, into the associated SODALITE editor for inspection and further edition.

5.2.5 Improved Content Assistance

Context-aware intelligent context assistance has been largely improved for AADM edition and supported as well for RM edition. As the authoring of both RMs and AADMs could be complex and prone to error for REs and AoEs not quite familiarized with the TOSCA specification, edition assistance will facilitate the creation of those models. Content assistance must be context-aware so that it is only advised on the syntax or content that fits into a concrete edition insertion point. Depending on this point, the syntactic structure of the model is suggested to the user, showing the elements permitted by the DSL grammar at this point. See [Figure 33](#) for an example of syntactic structure content suggested to the user in AADMs.

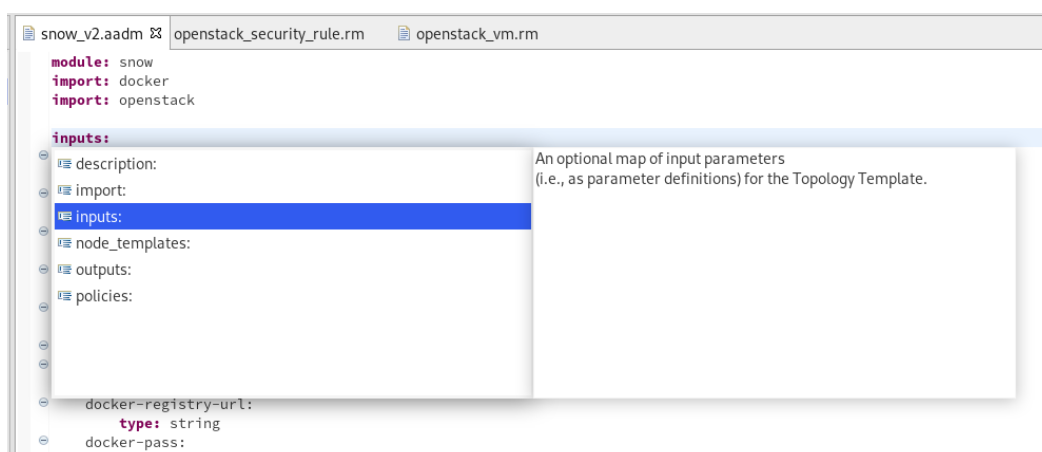


Figure 33: Content assistance for AADM syntax and structure

At another edition insertion point, content assistance can suggest content that can be either retrieved by searching the model, and/or by requiring the KB. The model itself is searched for entities defined therein that fit into the insertion point. Similarly other entities stored in the KB that fit into that point can be retrieved by requesting the KB reasoner, which provides a number of APIs for complex semantic queries. These queries can search for:

- modules stored in the KB for importing,
- inputs defined in the model,
- data types, node types, relationship types, interface types, (see [Figure 34](#) and [Figure 35](#))
- properties, attributes, requirements, capabilities, etc. of a given node type (see [Figure 36](#)),
- node templates that can satisfy a given requirement (see [Figure 37](#)),
- file system implementations and dependencies of operations of interfaces, etc.

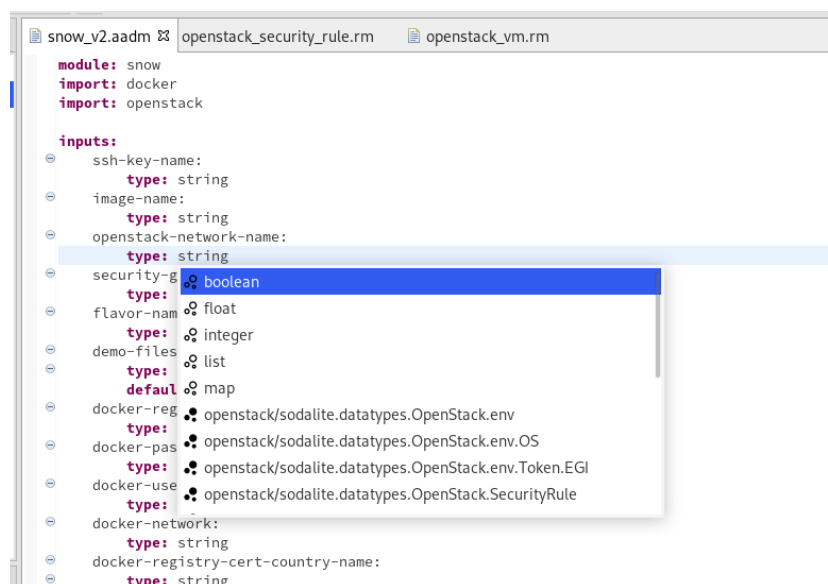


Figure 34: Content assistance for data types

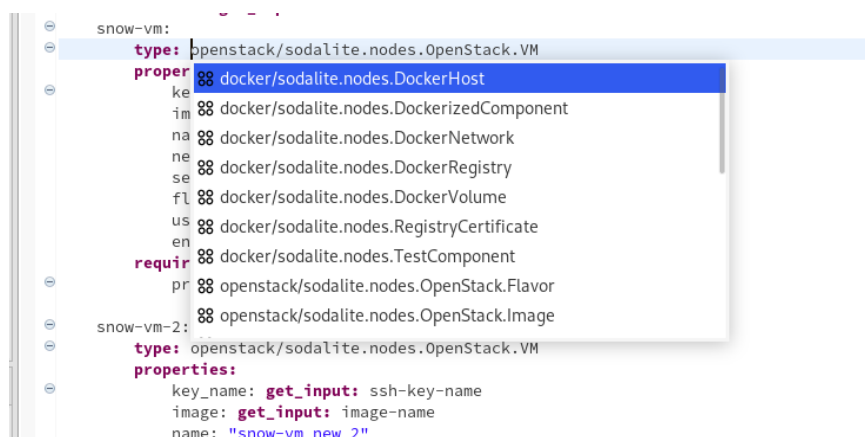


Figure 35: Content assistance for node types

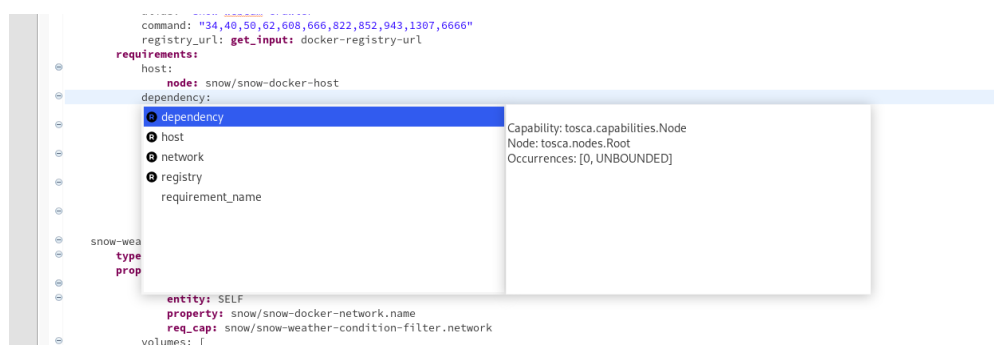


Figure 36: Content assistance for requirements

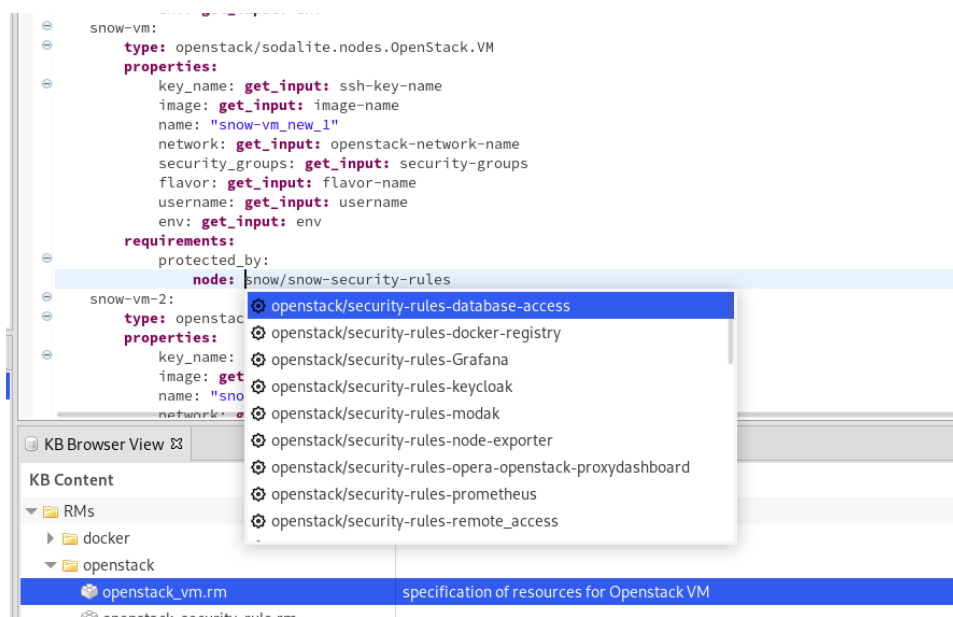


Figure 37: Content assistance for nodes satisfying requirements

When content assistance is requested by the user, a drop-down combo box is unrolled, showing all suggestions, ordered alphabetically. The user can navigate through the list using the vertical scroll bar or the cursor arrows. The list can also be filtered by typing, so that only suggestions matching the typed string are displayed. If the user accepts one selected suggestion, it is inserted at the cursor. In case the suggestion contains additional information, this is displayed in another window next to the one showing the suggestions.

5.2.6 Semantic Validation (Reporting and Quick fixes)

Semantic validation has been largely improved since M18 release. When the user saves a model (either an AADM or a RM) into the KB, the KB Reasoner parses the model and conducts a semantic validation process that checks for the semantic consistency of the model. In case validation errors are detected, the saving process is aborted and the user gets reported with the list of found errors. If no errors are detected, the KB Reasoner conducts a number of additional validation checks searching for recommendations. KB Reasoner also relies on the Defects Predictor (D4.2) for additional suggestions. Both KB-based recommendations and Defects-Predictor-based recommendations are reported (as previous errors) to the user.

As a result of the process that saves a AADM or RM into the KB, the IDE may obtain a list of semantica validation issues (e.g. errors, recommendations and suggestions), all of them associated with a concrete path within the model. If so, the IDE displays them all to the user in the model editor, at the point described by the issue's path (see [Figure 38](#)).

In some cases, the KB Reasoner can propose suggestions to fix detected errors or recommendations. In so, the IDE offers the user to apply those suggestions as *quick fixes*. The user can click on one of those quick fixes to apply the associated suggestion into the model. For instance, the user can select a quick fix to create a requirement for an application component (e.g. node template) (see [Figure 39](#))

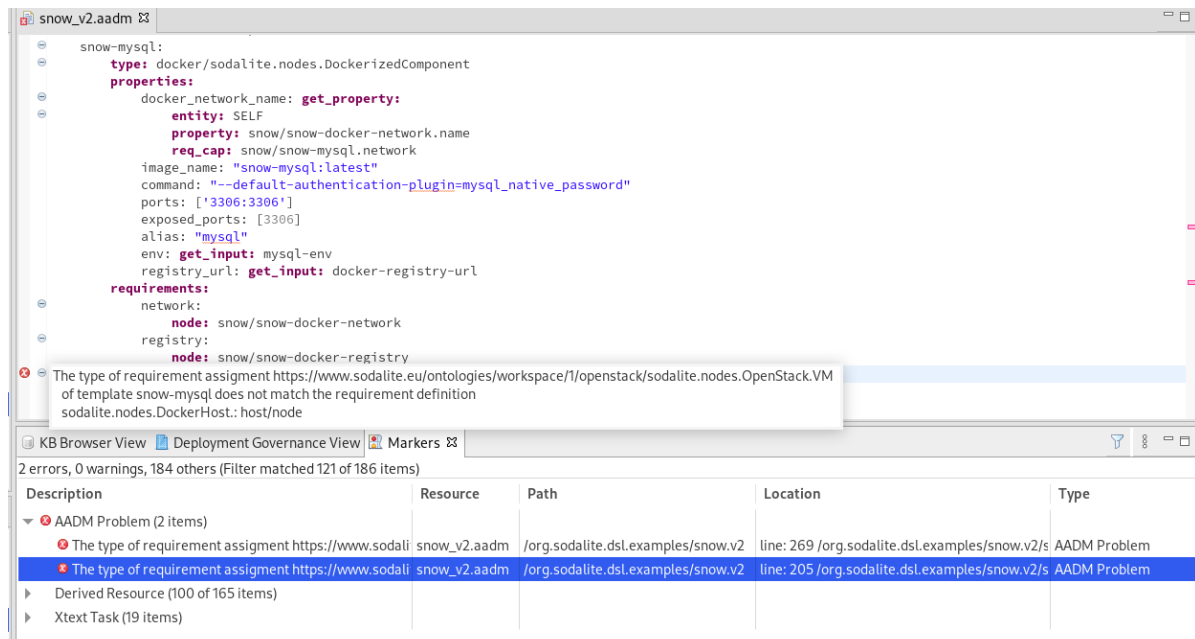


Figure 38: Semantic validation issue reported in AADM textual editor

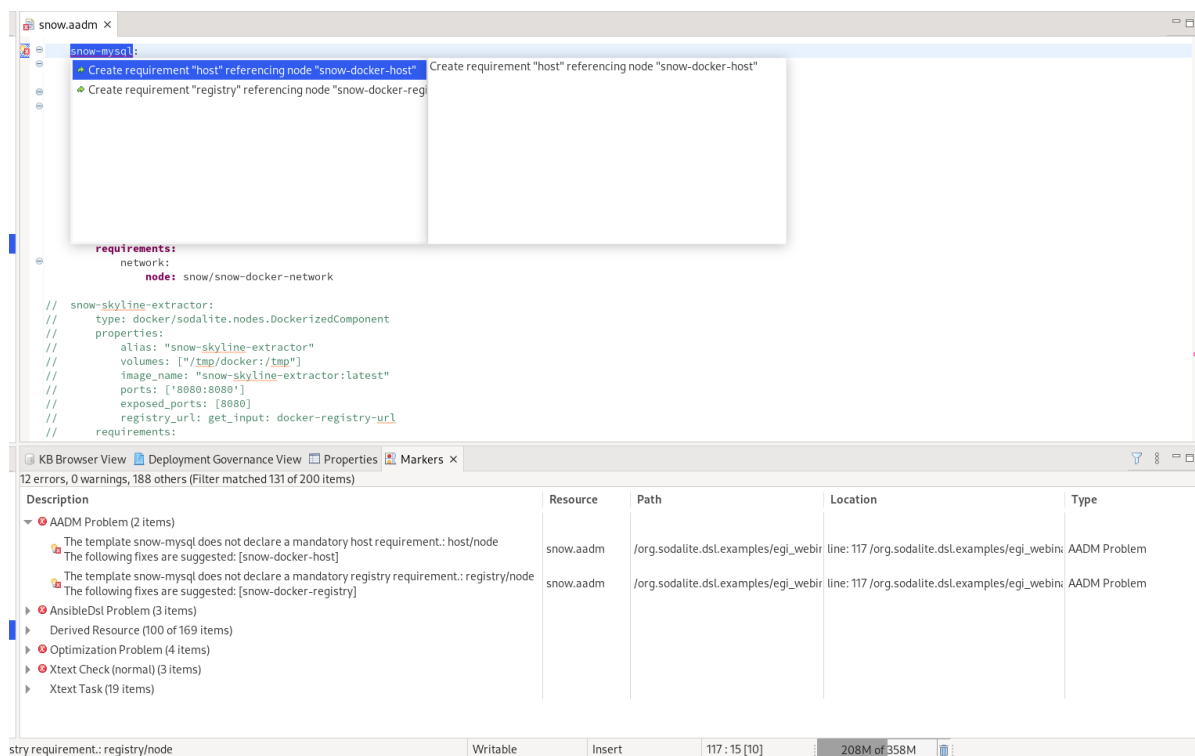


Figure 39: Quick fixes in AADM textual editor

5.2.7 AADM Versioning

IDE and KB support the versioning of AADMs, as AoEs may need to maintain multiple evolving versions of their applications and components. For this reason, when AADMs are saved into the KB, the IDE wizard (see [Figure 40](#)) prompts AoEs to set an optional version label or to retrieve an existing one.

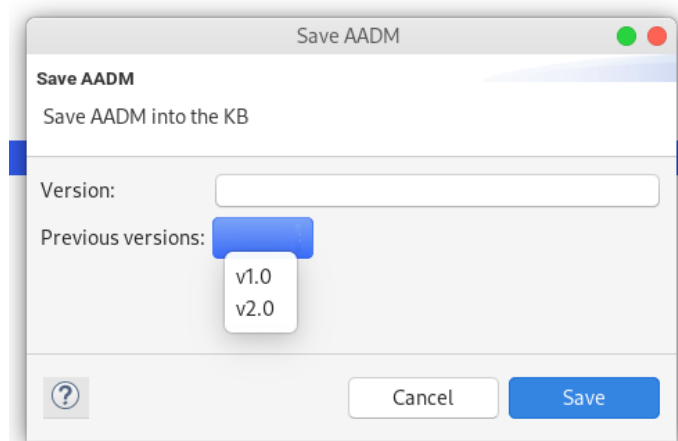


Figure 40: AADM versioning in save wizard

Similarly, the KB Browser view shows the different versions stored in the KB for an existing AADM (see [Figure 32](#)). Moreover, In content assistance, multiple versions of available resource instances (declared in different versions of AADMs) are suggested to the user, if available.

5.2.8 IAM/Secrets Management

The communication between the IDE and the SODALITE backend services (e.g. KB, Orchestrator, etc) is restricted to registered authorized users, by adopting the SODALITE IAM framework [D2.3]. Therefore, the IDE user must configure her IAM account in the SODALITE User Account preference page (see [Figure 41](#)).

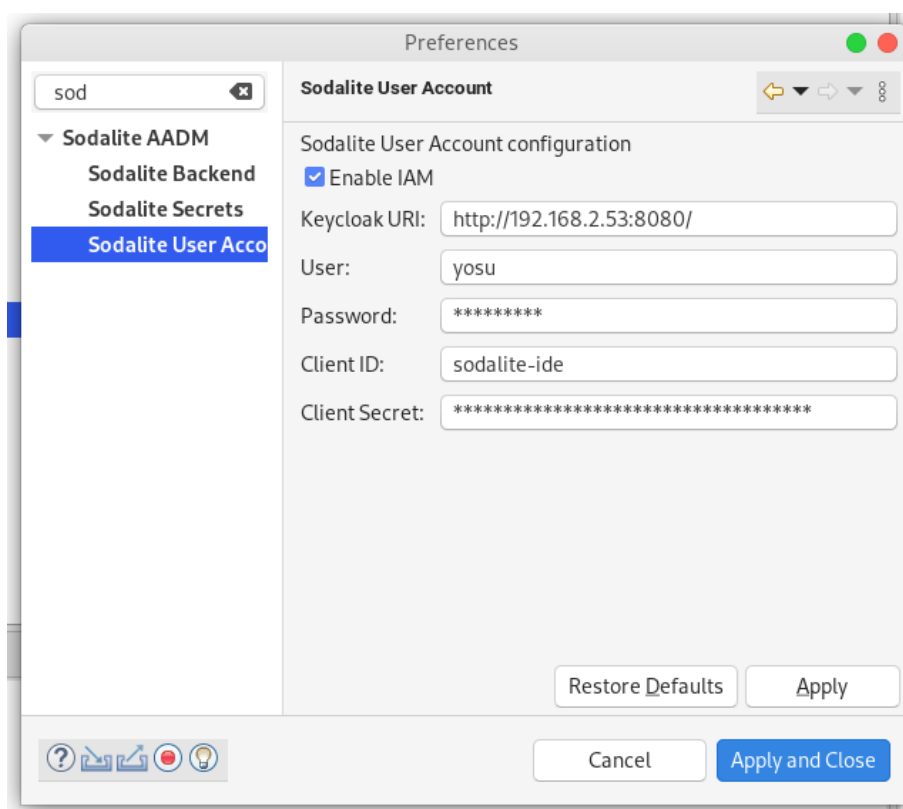


Figure 41: IAM configuration

Similarly, the secrets required to access HPC infrastructures must be stored in the SODALITE Vault by using the Secrets preference page (see [Figure 42](#)). Using this page, AoEs can add their secrets for each HPC infrastructure they need to deploy applications into. Secrets are stored in Vault.

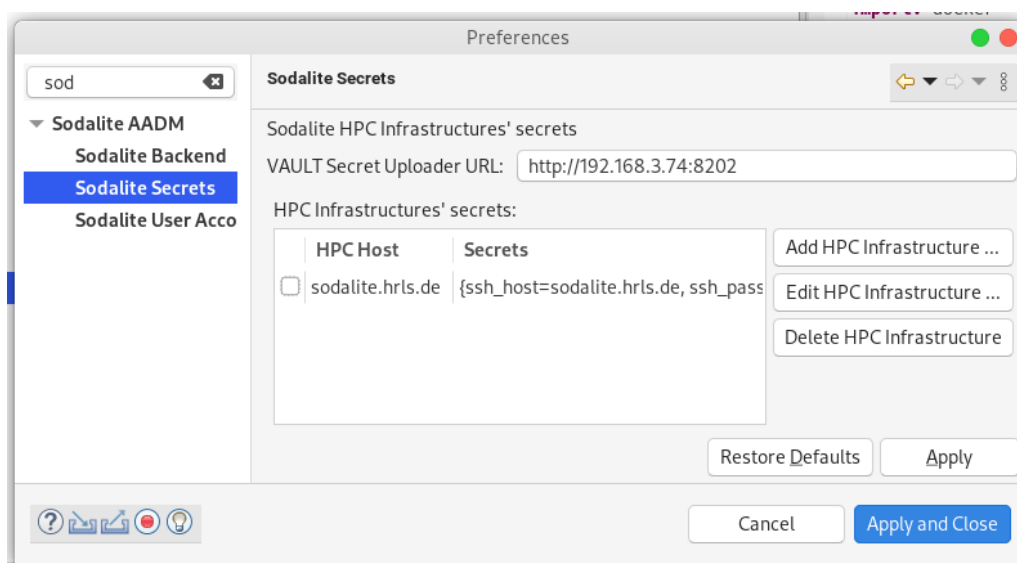


Figure 42: HPC Secrets configuration

5.2.9 Image Builder Integration

Before deploying applications that require specific containers, their images must be available in the image registry. In case of AoE specific images, they need to be built and registered before any application that allocates container instances of such images is deployed. The IDE permits AoEs or REs to request Image Builder (D4.2) to build images by providing an image descriptor (see [Figure 43](#))

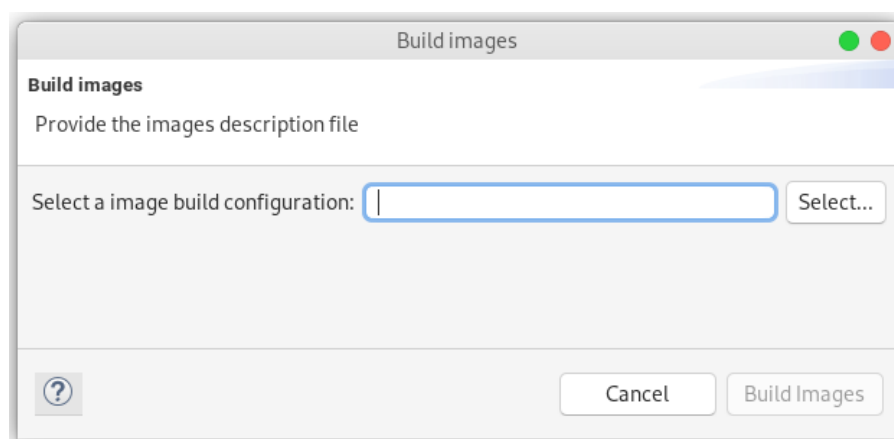


Figure 43: Image Builder wizard

5.2.10 PDS Integration

Infrastructure resources can be discovered by the Platform Discovery Service (PDS) (D4.2) for specific kinds. The IDE also permits REs to request, on demand, to PDS to discover the resources of an infrastructure kind (see [Figure 44](#)). The wizard prompts the user to provide a PDS descriptor file that configures the discovery process, a namespace (e.g. module in IDE terminology) where to associate discovered resources with, and the platform type (from those supported by PDS).

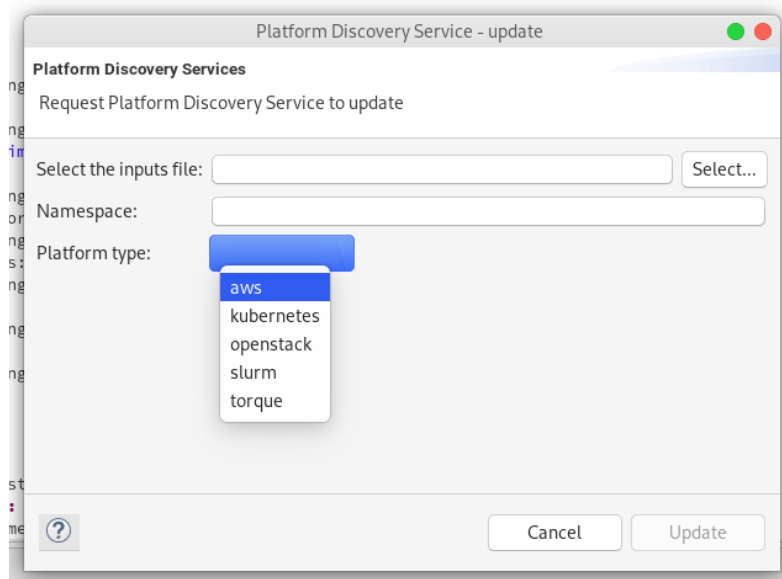


Figure 44: PDS wizard

6 Updated Modelling Layer architecture

The initial architecture of the Modelling Layer was initially defined in D2.1 and further detailed in D3.1, D2.2 [27] and D2.3. This section describes the updates that have been made since then. Figure 45 shows the updated high-level SODALITE General Architecture by layers where there were limited changes. Figure 46 shows the Modelling Layer architecture where significant changes were done after the First Year of the project. All those changes have already been reported in D2.3.

During the development phase, new and updated interactions with various components have been added for solving integration issues and adding new functionality to the SODALITE platform. Regarding the Modelling Layer, the following changes have been added:

- An interaction with the Defect Predictor for presenting to the users detected anti-patterns and bugs in the topology (more details in D4.2).
- Additional DSL editors and languages have been added to the IDE for targeting different modeling roles and supporting separation of modeling concerns. In particular, new DSL editors have been developed for authoring Ansible Models, Resource Models, and Optimization Models.
- IDE interacts with new APIs for giving visibility and access to the user in the deployment and runtime phase. Additionally, the user can call the Platform Discovery Service, described in D4.2, for discovering resource models and saving them in the KB.

Also, the Modelling Layer components have been enhanced with Security for making the framework more safe and appealing to the users. Specifically, the Modelling Layer has been integrated with the Security APIs that have been introduced in the second year of the project. The IDE and the Semantic Reasoner use the Identity and Access API (IAM) for authentication and authorization of the requests, and only the IDE uses the Secret Management API (Vault Service) for securely handling secret storage.

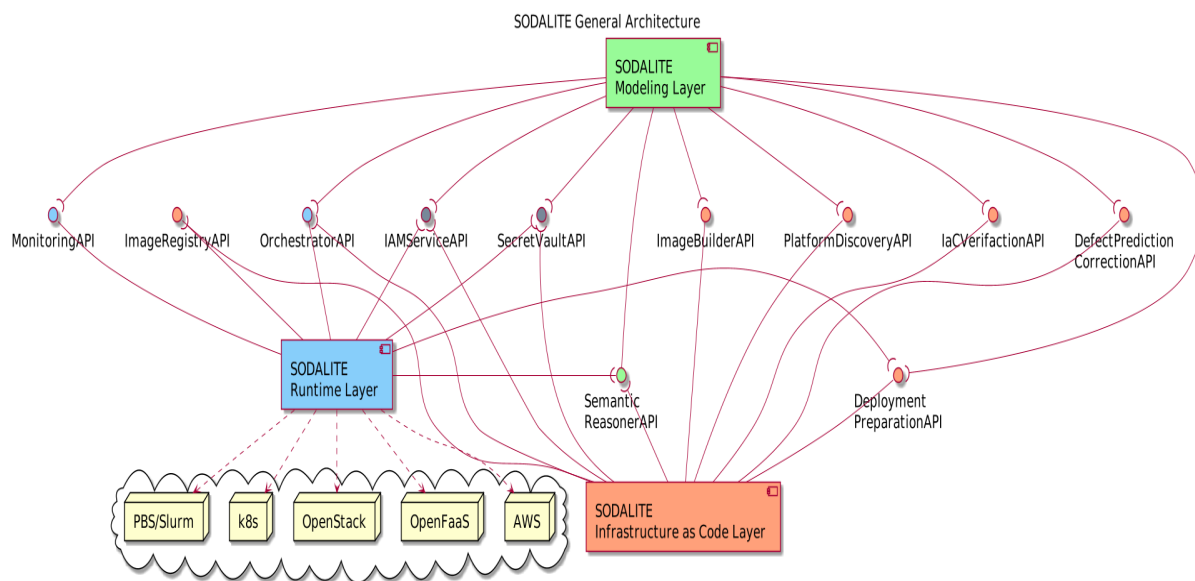


Figure 45. Updated SODALITE layers general architecture

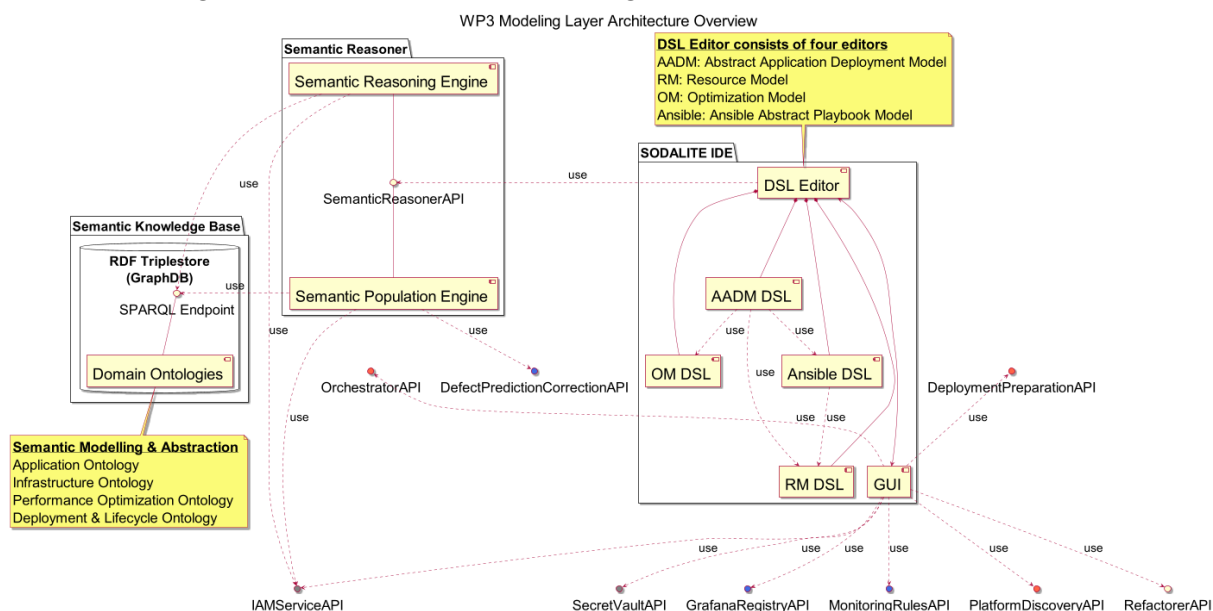


Figure 46. Updated Modelling Layer Architecture

7 Final version of the components

In this section, the final version of the Modelling Layer components will be shown.

In Figure 47, the Modelling Layer components are depicted with their internal interactions and their external interactions with the other two SODALITE layers, the IaC layer and the Runtime Layer. It is a different figure than Figure 46, so as to group and highlight the interactions with the other two layers. The IDE interacts with the Semantic Reasoner either for saving a model (*exchange TTL- input*) or for getting other information such as context assistance (*url parameters - input*). As it is shown from the interactions, the IDE interacts with various components of all the layers in order to enable the user to manage the process of application deployment. Also, the Semantic Reasoner interacts with new components, namely the Platform Discovery Service, the Defect Predictor and the Refactorer. Also, all the interactions have been secured by getting and validating secrets through the Secret Vault and the IAM services.

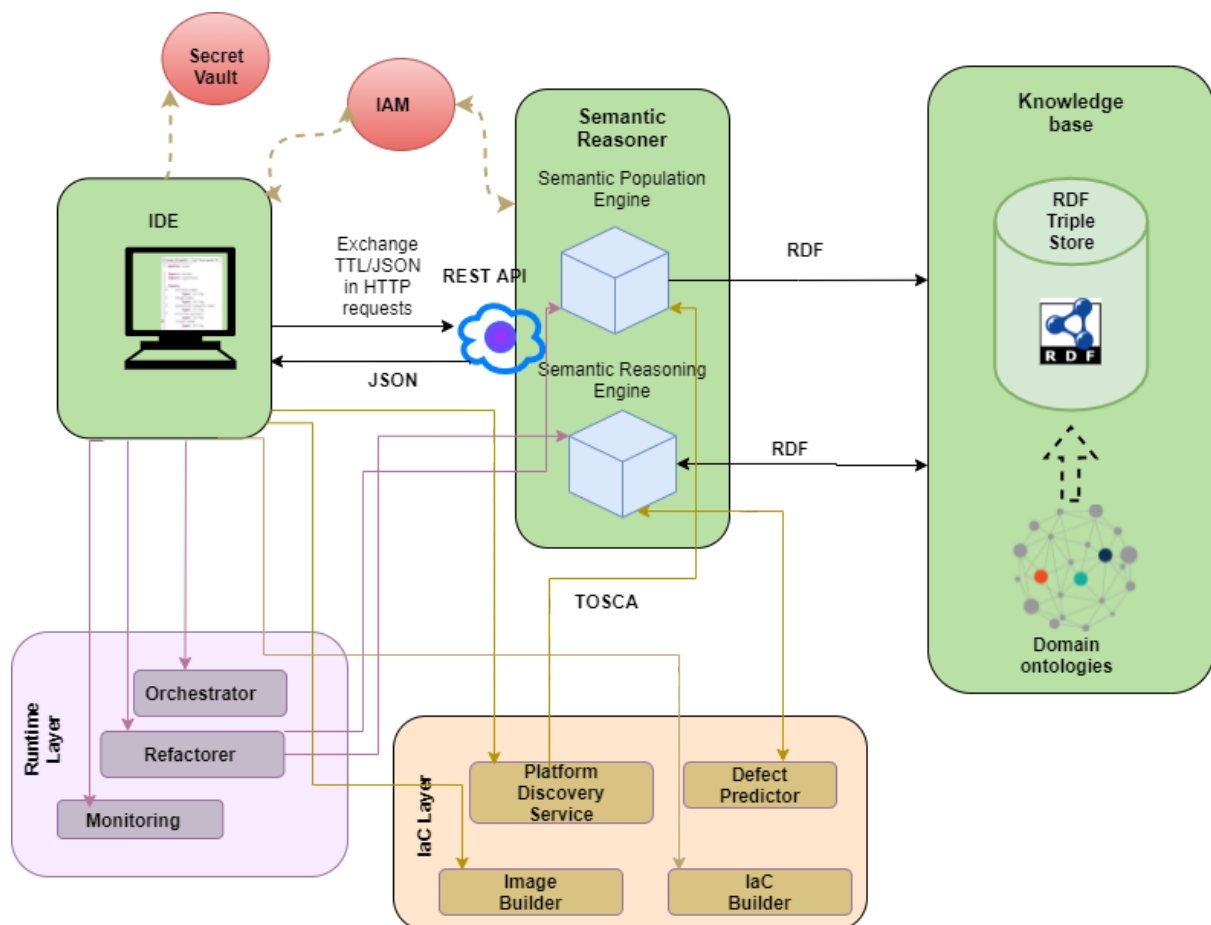


Figure 47. Modelling Layer architecture and its interactions with the other layer

7.1 Semantic Reasoner

The Semantic Reasoner is responsible for uncovering hidden knowledge through the strong inference capabilities of the Knowledge Base. It is a service that serves as an interface between the KB and the rest of the components. It is a central point on the platform, as it is used by components of all the layers of the platform, the Modelling layer, the IaC layer and the Runtime Layer. Namely:

- IDE has an intensive interaction with the Semantic Reasoner for getting access to the Intelligent services

- The Defect predictor communicates with the Semantic Reasoner for getting access to the models in the KB.
- The IaC builder
- The Platform Discovery Service that discovers TOSCA Resource Models, sends those models to the Semantic Reasoner for saving them to the KB and making them available to the modellers.
- The Refactorer communicates with the Semantic Reasoner and the KB for discovering new deployment variants, and then saving the modified model to the KB.

Semantic Reasoner uses external services for IAM [Keycloak] to perform user authentication and authorization getting a *token* as an input.

- **Semantic Reasoning Engine (SRE)**

The Reasoning Engine guarantees the consistency of the RDF Knowledge Graphs in terms of native OWL2 semantics in the OWL2 RL dialect. It provides most services that provide intelligence to the IDE user. Those services implementing custom reasoning logic are implemented in T4.4 - Analytics and Semantic Decision Support. An initial version was presented in D3.1, and a significantly updated version is presented in this deliverable.

- **Semantic Population Engine (SPE)**

The population engine implements the custom population logic of the KB, for example, by mapping the DSL concepts sent by IDE in exchange TTL format to the SODALITE abstraction model. The mapping services were significantly extended for:

1. mapping Resource Models, and additional new concepts such as policy types, triggers, data types etc.
2. mapping new concepts in the AADMs such as outputs, inputs, policies etc.
3. mapping TOSCA to ontologies. This part is used both by the Platform Discovery Service and the Refactorer. This mapping service enables PDS to save TOSCA resources in the KB. Also, the Refactorer uses those mapping back-end services for saving the refactored models to the KB.

Code Quality

This module is written in Java, and it is integrated in SonarCloud for quality assessment and obtained the following quality score : 0 bugs, 0 vulnerabilities, 0 Security Hotspots, 84.5% code coverage, 758 code smells and 2.1% code duplications. The Sonarcloud results for the Semantic Reasoner are displayed in SonarCloud [Dashboard](#). In Figure 48, the quality results for the Semantic Reasoner are depicted.

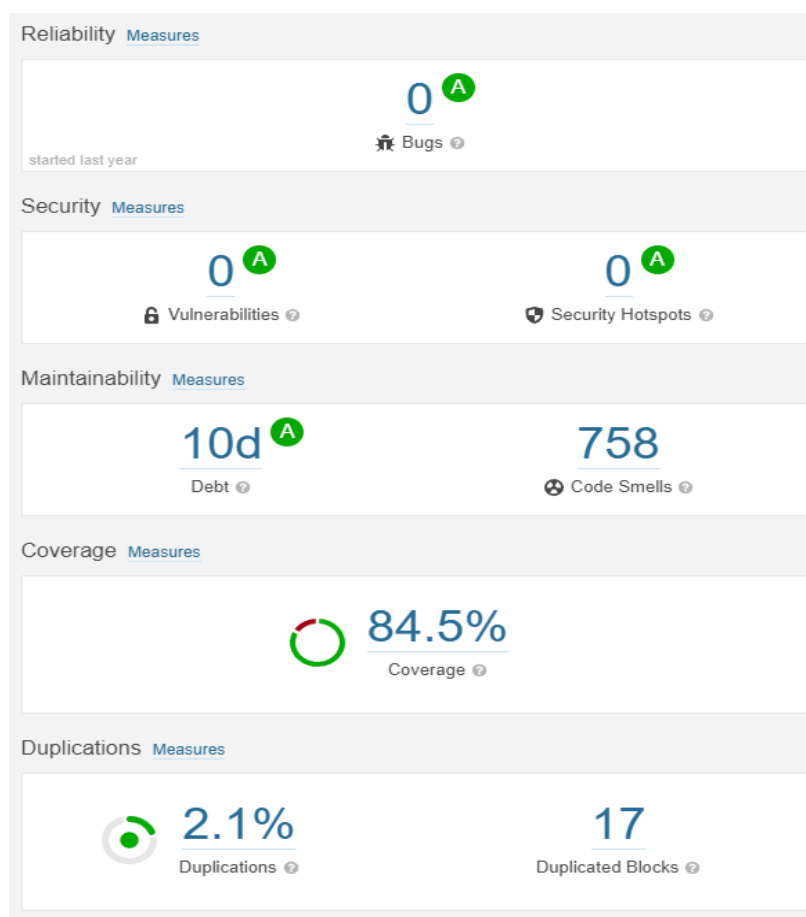


Figure 48. Code Quality Report for Semantic Reasoner

7.1.1 Semantic Reasoning Engine (SRE)

During the second and third year of the project, the existing REST API endpoints, that assist the modellers, were enhanced, and additional APIs were developed. This functionality has been developed under T4.4 for providing semantic decision support to the user. Some of the main functionalities that were developed:

- Enable the KB browser view in the IDE by providing services for getting and deleting models.
- Context assistance for getting the available workspaces in the KB, the available types(data, capability, node etc.), templates, operations, nodes that can satisfy requirements.
- Return the Resource Models in json format for enabling the IDE to display the DSL of the models discovered by the Platform Discovery Service.

The final version of this component along with its APIs are documented in the [Appendix](#).

7.1.1.1 Advanced Reasoning services - examples

Validation

1. Topology validation

There are errors in the topology of an application that are difficult to be manually detected since the application components depend on inter-node relationships. Thus, it is crucial, at the design time, to ensure that the TOSCA topology does not contain errors, and all the relationships between the nodes are valid. We followed the validity conditions defined in the Sommelier tool [25], an open-source validator for TOSCA application topologies. All the TOSCA elements that are forming a relationship are checked, namely the source (Requirements of a node), the relationship itself, and its target (a node or a capability of a node).

We will show two examples, one for the requirement validation, and one for the capability validation. Regarding the requirement validation, one simple validation is depicted in Figure 49 through a SPARQL query. The requirement assignment names in a template should be present in the node type hierarchy of the template. One more complex validation is about additional checks for capabilities. More precisely, when a requirement assignment is instead only indicating the target node template (without indicating any of its capabilities), it must be checked whether such node template is offering at least one type-compatible capability. In Figure 50, we can see an example of the *snow-demo* template that has as a network the *snow/snow-docker-network* whose type (*snow.nodes.DockerNetwork*) offers a capability of a type - *tosca.nodes.Root*, that is not subclass of the *requirements/network/capability=tosca.capabilities.Network* denoted in the *sodalite.nodes.DockerizedComponent*. In Figure 51, a SPARQL query is depicted that detects the capability mismatch.

```
1  select distinct ?nodeType ?ctx
2  where {
3    ?nodeType soda:hasInferredContext ?ctx;
4    rdfs:subClassOf tosca:tosca.entity.Root.
5    ?ctx tosca:requirements [DUL:classifies ?r_a] .
  }
```

Figure 49. SPARQL query detecting Requirement Mismatch

```

snow-demo:
  type: docker/sodalite.nodes.DockerizedComponent
  requirements:
    network:
      node: snow/snow-docker-network ←invalid

snow-docker-network:
  type: docker/sodalite.nodes.DockerNetwork

sodalite.nodes.DockerNetwork:
  derived_from: tosca.nodes.SoftwareComponent
  capabilities:
    network:
      type: tosca.nodes.Root

sodalite.nodes.DockerizedComponent:
  derived_from: tosca.nodes.SoftwareComponent
  requirements:
    network:
      capability: tosca.capabilities.Network

```

Figure 50. Capability Mismatch model example

```

1  select *
2  where {
3    ?template soda:hasContext [tosca:requirements [DUL:classifies ?r_a; DUL:hasRegion ?r_a_node]] .
4    ?r_a_node sesame:directType ?type_r_a_node.
5    ?type_r_a_node rdfs:subClassOf tosca:tosca.entity.Root .
6
7    ?type_r_a_node soda:hasInferredContext ?ctx2;
8    rdfs:subClassOf tosca:tosca.entity.Root.
9    ?ctx2 tosca:capabilities [DUL:classifies ?r_a; DUL:hasParameter
10     [DUL:classifies tosca:type; DUL:hasRegion ?templateCapabilityType]] .
11
12    ?ctx tosca:requirements [DUL:classifies ?r_a; DUL:hasParameter
13     [DUL:classifies tosca:capability; DUL:hasRegion ?r_d_capability]] .
14    FILTER NOT EXISTS {?templateCapabilityType rdfs:subClassOf ?r_d_capability .}
15  }

```

Figure 51. SPARQL query detecting Capability Mismatch

2. Property/Attribute Validation

In the type schema, it is optional to be defined if a property is required to be assigned to a template by the *required* key. If there is a type with a property that has *required* = true, and there is no default value, then all instances of this type, namely templates, should have assignment for this property. If not, the Semantic Reasoner throws the corresponding error to the IDE user. In Figure 52, a node type that has a required property is depicted. In Figure 53, a SPARQL detecting the required properties with no default values and not assigned to a template is depicted.

```
node_types:
  sodalite.nodes.DockerNetwork:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
      name:
        type: string
        description: "The name of the network"
        required: true
```

Figure 52. Excerpt from the sodalite.nodes.DockerNetwork

```
1  select distinct ?property
2  where {
3    ?resource soda:hasInferredContext ?context .
4    ?context tosca:properties ?concept .
5    ?concept DUL:classifies ?property .
6    {
7      ?concept DUL:hasParameter [DUL:classifies tosca:required; tosca:hasDataValue true].
8    } UNION {
9      FILTER NOT EXISTS {?concept DUL:hasParameter
10                          [DUL:classifies tosca:required; tosca:hasDataValue []]}.
11    }
12 }
```

Figure 53. SPARQL query detecting required properties

3. Constraint property validation

A constraint clause might be optionally present in the property definition of the type defining the allowed values that can be assigned in the corresponding template property. The constraints can be as simple as a list of valid values or as complex as the length of custom types. For example, in Figure 54, the constraints for the *ports* property is the minimum number of assigned *openstack/sodalite.datatypes.OpenStack.SecurityRule* objects. In Figure 55, the SPARQL query returns the constraint for the properties that have *entry_schema*. If an instance of the *openstack/sodalite.nodes.OpenStack.SecurityRules* has no *openstack/sodalite.datatypes.OpenStack.SecurityRule* objects assigned to the *ports* property, an error is thrown as the *min_length* is 2.

```
node_types:
  sodalite.nodes.OpenStack.SecurityRules:
    derived_from: tosca.nodes.Root
    properties:
      ...
      ports:
        required: false
        constraints:
          min_length: "1"
          type: map
          entry_schema: openstack/sodalite.datatypes.OpenStack.SecurityRule
```

Figure 54. A node type with constraints within properties


```
1  select distinct ?property ?constraint ?constr_type ?value ?entry_schema
2  where {
3    ?var soda:hasInferredContext ?context .
4    ?context tosca:properties ?concept .
5    ?concept DUL:classifies ?property .
6    ?concept DUL:hasParameter
7      [DUL:classifies tosca:constraints; DUL:hasParameter
8      [DUL:classifies ?constraint; tosca:hasDataValue ?value]].
9    ?concept DUL:hasParameter
10     [DUL:classifies tosca:type; tosca:hasValue ?constr_type].
11    ?concept DUL:hasParameter
12     [DUL:classifies tosca:entry_schema; DUL:hasParameter
13     [DUL:classifies tosca:type; tosca:hasObjectValue ?entry_schema]].
14  VALUES ?constraint {tosca:min_length tosca:length tosca:max_length}
15 }
```

Figure 55. SPARQL query returning the constraints

Matchmaking and reuse

Nodes can be detected that satisfy a specific requirement. The check for the requirement definition does not suffice, but more complex checks are needed for the capabilities offered. This will be presented through an example. In Figure 57, the node template for which a suitable requirement host is detected. In Figure 56, some node types are depicted. The *sodalite.nodes.DockerizedComponent* type is the type from which the *snow-my-sql* is instantiated. This type inherits the *tosca.nodes.SoftwareComponent* which has as a requirement/host/node the *tosca.nodes.Compute* type. So all the node templates that are instances of *tosca.nodes.Compute* can host the *snow-my-sql* (Step 1). In addition to *tosca.nodes.Compute*, other types of templates can also serve as a host for *snow-my-sql*. The *tosca.nodes.SoftwareComponent* has requirements/host/capability=*tosca.capabilities.Compute*. The *sodalite.nodes.DockerHost* has capabilities/host/type=*tosca.capabilities.Compute* which is the same type denoted within the requirements of the *tosca.nodes.SoftwareComponent* (Step 2), and has *valid_source_types* = *docker/sodalite.nodes.DockerizedComponent* (Step 3). So both instances of *tosca.nodes.Compute* and *docker/sodalite.nodes.DockerHost* types can serve as a host for the *snow-my-sql* template.

In Figure 57, the *snow-mysql* template is depicted, for which we are searching a valid requirement host node, and in this figure the *snow/docker-host@v1* is added as a suitable example host of type *sodalite.nodes.DockerHost*. In Figure 58, the Step 1 is depicted, the SPARQL query retrieves the type of templates that can serve as a host from the type definition (requirements/host/node) of the template. In Figure 59, the Step 2 is depicted, the types that offer a capability with the same type (e.g. *tosca.capabilities.Compute*) as in the definition of the type of the template are detected through a SPARQL query. Finally, in Figure 60, the Step 3 is depicted, the *valid_source_types* of the matching types of Step 2 are retrieved, and if the type of our template (*snow-mysql*) is subclass of the *valid_source_types*, then those types from Step 2 are suitable hosts for the *snow-mysql*.

```

sodalite.nodes.DockerizedComponent:
    derived_from: toska.nodes.SoftwareComponent

sodalite.nodes.DockerHost:
    derived_from: toska.nodes.SoftwareComponent
    capabilities:
        host:
            type: toska.capabilities.Compute
            valid_source_types:[docker/sodalite.nodes.DockerizedComponent]
)
toska.nodes.SoftwareComponent:
    derived_from: toska.nodes.Root
    requirements:
        host:
            capability: toska.capabilities.Compute
            node: toska.nodes.Compute
            relationship: toska.relationships.HostedOn

```

Figure 56. Node type hierarchy

```

snow-mysql:
    type: docker/sodalite.nodes.DockerizedComponent
    requirements:
        host:
            node: snow/docker-host@v1.0

```

Figure 57. Template for which a suitable host is detected

1	select ?r ?v
2	where {
3	?node soda:hasInferredContext ?ctx.
4	?node rdfs:subClassOf toska:tosca.entity.Root.
5	?ctx toska:requirements ?r .
6	?r DUL:classifies ?p.
7	?r DUL:hasParameter [DUL:classifies toska:node; DUL:hasRegion ?v] .
8	FILTER (STRENDS (str(?p), ?requirementName)) .
9	}

Figure 58. SPARQL detecting compatible requirements according to the type definition

```
1  select ?v
2  where {
3      ?node soda:hasInferredContext ?ctx .
4      ?node rdfs:subClassOf tosca:tosca.entity.Root .
5      ?ctx tosca:requirements ?r .
6      ?r DUL:classifies ?p .
7      ?r DUL:hasParameter [DUL:classifies tosca:capability;
8                          DUL:hasRegion ?v] .
9      FILTER (STRENDS (str(?p), ?requirementName)) .
10 }
```

Figure 59. SPARQL retrieving the required capability types from a node definition

```
1  select ?node ?v_s_type
2  FROM <http://www.ontotext.com/explicit>
3  FROM NAMED <https://www.sodalite.eu/ontologies/workspace/1/docker>
4  where {
5      ?node soda:hasContext ?ctx.
6      ?ctx tosca:capabilities ?c .
7      ?c DUL:classifies ?p.
8      ?c DUL:hasParameter [DUL:classifies tosca:type;
9                          tosca:hasObjectValue ?cap_type] .
10     ?c DUL:hasParameter [DUL:classifies tosca:valid_source_types;
11                          tosca:hasObjectValue/tosca:hasObjectValue ?v_s_type]
12     FILTER (STRENDS (str(?p), ?requirementName)) .
13 }
```

Figure 60. SPARQL retrieving the `valid_source_types` of node types that offer specific capability type (`cap_type`)

Abstraction DSL

Requirements can totally be omitted in the AADM, and the Semantic Reasoner can handle the current situation. Required requirements are those which have no occurrences¹⁰ in the type definition or a minimum value of 1. If a required requirement is missing at the design time, the Semantic Reasoner throws an error and suggests suitable nodes. If a required requirement is missing at the deployment time, the Semantic Reasoner autofills the missing requirements in the model. In Figure 61, a SPARQL is depicted that detects the required requirements that are missing in a template according to the type definition. If an optional requirement is missing (occurrences with `min = 0`), a suggestion is sent during the design time, and the requirement is concretized by the reasoner during the deployment time.

¹⁰

https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html#_Toc26969448

```
1 select distinct ?v ?r_a ?r_i
2 where {
3   ?aadm soda:includesTemplate ?template .
4
5   ?template a soda:SodaliteSituation ;
6   sesame:directType ?templateType .
7   ?templateType soda:hasInferredContext ?ctx.
8   ?templateType rdfs:subClassOf tosca:tosca.entity.Root.
9   ?ctx tosca:requirements ?r .
10  ?r DUL:classifies ?r_a.
11  ?r DUL:hasParameter [DUL:classifies ?r_i; DUL:hasRegion ?v] .
12  {
13    FILTER NOT EXISTS {
14      ?r DUL:hasParameter [DUL:classifies tosca:occurrences; DUL:hasParameter
15        [DUL:classifies tosca:min; tosca:hasDataValue ?occMin]] .
16    }
17  } UNION {
18    ?r DUL:hasParameter [DUL:classifies tosca:occurrences; DUL:hasParameter
19      DUL:classifies tosca:min; tosca:hasDataValue ?occMin]] .
20    FILTER (xsd:integer(?occMin) > 0) .
21  }
22  FILTER NOT EXISTS {
23    ?template soda:hasContext [tosca:requirements [DUL:classifies ?r_a]] .
24  }
25 }
```

Figure 61. A SPARQL query detecting required requirements

Optimization suggestions

MODAK is enabled when an Optimization DSL (D3.4) is provided and associated with an AADM. In Figure 62, the optimization-skyline-extractor, which is an ai training application, is depicted as associated with an optimization DSL, the *ai_training.tensorflow*. The *optimization-skyline-extractor* is hosted in the *optimization-vm* which has as capabilities `num_gpus = 1`. Since the number of GPUs is greater than zero, if the optimization DSL does not have the XLA compiler enabled for the Tensorflow framework, the Semantic Reasoner returns a suggestion for enabling the XLA compiler. Figure 63 shows a SPARQL query for retrieving the capabilities of the host of a template. Since the host can be nested in many levels, thus the UNION in the query. Then, after the capabilities are retrieved, the SPARQL query, in the Figure 64, is executed for retrieving the exact number of gpus.

```

optimization-skyline-extractor:
    type: docker/sodalite.nodes.DockerizedComponent
    optimization: ai_training.tensor_flow
    ...
    requirements:
        host:
            node: optimization/optimization-vm

optimization-vm:
    type: openstack/sodalite.nodes.OpenStack.VM
    capabilities:
        host:
            properties:
                disk_size: "10 GB"
                num_cpus: 4
                num_gpus: 1 ←
                mem_size: "4 GB"
        os:
            properties:
                architecture: "x86_64"

```

Figure 62. AI Training node template associated with optimization DSL

```

1  select ?resource ?capability ?optimizations ?templateType
2  where {
3      {
4          #capabilities of host nodes
5          ?resource (soda:hasContext/tosca:requirements/tosca:hasObjectValue)+/
6          soda:hasContext/tosca:capabilities ?capability .
7      } UNION {
8          #direct capabilities
9          ?resource soda:hasContext/tosca:capabilities ?capability .
10     }
11     ?resource soda:hasContext/tosca:optimization ?optimizations .
12     ?resource sesame:directType ?templateType .
13     ?templateType rdfs:subClassOf tosca:tosca.entity.Root .
14     ?var_aadm soda:includesTemplate ?resource .

```

Figure 63. SPARQL query retrieving the capabilities of the node where a template is hosted

```
1 select ?ngpu
2 where {
3
4     #Retrieve number of gpus
5     ?capability DUL:classifies tosca:host .
6     ?capability tosca:properties ?property .
7     ?property DUL:classifies ?property_gpus .
8     FILTER (strends(str(?property_gpus), "num_gpus")) .
9     ?property tosca:hasDataValue ?ngpu .
10 }
```

Figure 64. SPARQL query retrieving the number of gpus of a capability

7.1.2 Semantic Population Engine (SPE)

As already mentioned in the introduction of this section, the Semantic Population Engine has been significantly updated since the first year. Saving Resource Models, mapping from TOSCA to ontologies that is used by the Platform Discovery Service, mapping services that are used by the Refactorer for saving refactored AADMs to the KB, and mapping of additional concepts (policies, output etc.) to ontologies are some of the most significant extensions.

The mapping services were updated in order to:

- support versioning of the AADMs.
- support workspaces.
- provide authentication of the user. Only users having read/write permission to specific workspace can access a resource. For example, a user with no write permission to the *docker* workspace cannot save a Resource Model in this specific workspace.
- save the DSL to the KB for enabling the KB Browser view IDE feature.
- save metadata in the KB such as the model's file name.
- convert new structures such as policies, inputs, outputs, optimizations, tosca models into the SODALITE knowledge graphs.

The final version of this component along with its APIs are documented in the [Appendix](#).

7.1.2.1 Workspaces and Versioning

The user can save the models in a workspace that can be private or shared with other users.

This workspace support enables the users to work as teams and reuse models and resources among them. As it is depicted in Figure 65, a model denotes the workspace to which belongs e.g. snow, and the workspaces to be reused are imported. Also, we can notice the reference to private workspaces with the slash such as *docker/sodalite.nodes.RegistryCertificate*. If there is no prefix in a reference, then the global space is searched. Description about how the KB represents the workspaces is described in [2.1.5 section](#).

```
module: snow
import: docker
import: openstack
node_templates:
    snow-docker-registry-certificate:
        type: docker/sodalite.nodes.RegistryCertificate
        ...
    requirements:
        host:
            node: snow/snow-vm
```

Figure 65. Example model using workspaces

Also, the users can save different versions for the same aadm. The versioning mechanism can enable the Refactorer's work, developed in WP5, for saving the modified AADM in a different version from the original model.

7.1.3 CI/CD Integration

Like for the other SODALITE components, the process of deploying the Semantic Reasoner has been automated with the main CI/CD integration in SODALITE Jenkins¹¹.

7.2 Semantic Knowledge Base

7.2.1 RDF Triple Store

Knowledge base has been tuned for better performance. More specifically, the owl:sameas (*property to create an equivalence class between nodes of an RDF graph*) has been disabled since significant improvement was shown in the READ and DELETE operations. Also, we configured the Java heap memory allocated to the graphdb to be approximately 2/3 of the heap memory (flag -Xmx).

More technical details are described in the [Appendix](#).

7.2.2 Domain Models

This module contains all the SODALITE ontologies. The technical details are described in the [Appendix](#).

7.2.3 CI/CD Integration

Like for the other SODALITE components, the process of deploying the Knowledge Base with its updated domain ontologies has been automated with the main CI/CD integration in SODALITE Jenkins¹².

¹¹ <https://jenkins.sodalite.eu>

¹² <https://jenkins.sodalite.eu>



7.3 SODALITE IDE

The SODALITE IDE is implemented as a set of plugins for the Eclipse IDE. The code of these plugins is available in the GitHub repository: <https://github.com/SODALITE-EU/ide> under the path:

dsl/org.sodalite.IDE.parent (see [Figure 66](#)).

For each supported DSL (e.g. AADM, RM, optimization model, Ansible model, Alerting Rule model) there are four associated plugin projects:

- *org.sodalite.dsl.<DSL_NAME>*: this project defines the DSL grammar, and provides implementation for textual serialization (e.g. to the Turtle serialization required by the KB), textual formatting (how the DSL is visualized in the textual editor),
- *org.sodalite.dsl.<DSL_NAME>.ide*: this helper plugin supports the configuration of the DSL editors into the Eclipse IDE.
- *org.sodalite.dsl.<DSL_NAME>.ui*: this plugin supports different Eclipse UI features of the DSL editor, including support for content-assistance, notification of semantic errors and support for quick fixes in editor, implementation of the outline view. This plugin has been also used for other interactions with the Eclipse UI, hosting the implementation of DSL related menus and associated wizards, implementation for the integration of the backend processes (e.g. create AADM, save AADM or RM, deploy AADM, build images, invoke PDS).
- *org.sodalite.dsl.<DSL_NAME>.feature*: this plugin groups all DSL associated plugins into a common Eclipse IDE installable feature.



Figure 66: SODALITE IDE plugins' project structure

SODALITE DSL and associated plugin implementation is based on XText¹³ framework (see D5.1 for an SoTA analysis of this framework and the rationale for using it).

Other projects provides implementation for other features of the SODALITE IDE:

- `org.sodalite.dsl.AADM.design`: this plugin provides the implementation for the AADM graphical representation and edition, based on canvas and forms. The implementation is based on Sirius¹⁴ framework (see D5.1 for an SoTA analysis of this framework and the rationale for using it).

¹³ <https://www.eclipse.org/Xtext/>

¹⁴ <https://www.eclipse.org/sirius/>

- `org.sodalite.dsl.kb_reasoner_client`: this project centralizes the communications of the IDE with the SODALITE backend components (e.g. KB Reasoner, Orchestrator, etc.) by sending requests through their exposed REST APIs, and processing their JSON responses. This plugin relies on Spring Framework Boot¹⁵ APIs for REST, and Jackson Databind¹⁶ and Google GSON¹⁷ for JSON processing.
- `org.sodalite.dsl.preferences`: this plugin implements Eclipse preference pages for the configuration of the AIM security and secrets, and the configuration of the endpoints of the SODALITE backend services.
- `org.sodalite.ide.feature`: this plugin configures the entire bundle of features that constitute the SODALITE IDE.
- `org.sodalite.IDE.repository`: this plugin configures the IDE update site content, that facilitates the installation of the SODALITE IDE.
- `org.sodalite.ide.ui`: this plugin provides additional Eclipse UI features for the SODALITE IDE, including the KB Browser view and the Deployment Governance view.

The following sections provide additional technical implementation details for the main SODALITE IDE features.

7.3.1 Domain Specific Languages

As aforementioned, SODALITE DSLs are implemented as XText grammars. The grammar (see [Figure 67](#)) describes the syntax and grammatical rules of the DSL entities. SODALITE DSLs exploit the inheritance relationship of XText grammars, so that AADM DSL extends the RM DSL and imports (i.e. uses) the Optimization DSL.

```
8  * Contributors:
9  *   Jesús Gorroño - Design and implementation
10 *****/
11 //grammar org.sodalite.dsl.AADM with org.eclipse.xtext.common.Terminals
12 grammar org.sodalite.dsl.AADM with org.sodalite.dsl.RM
13
14 generate aADM "http://www.sodalite.org/dsl/AADM"
15 //import "http://www.sodalite.org/dsl/RM" as rm
16 import "http://www.sodalite.org/dsl/optimization/Optimization" as opt
17 //import "http://www.eclipse.org/emf/2002/Ecore" as ecore
18
19 AADM_Model:
20   ('description:' description=STRING)? &
21   ('module:' module=ID)? &
22   ('import:' imports+=ID)* &
23   ('inputs:'
24     BEGIN
25       inputs = EInputs
26     END)? &
27   ('node_templates:'
28     BEGIN
29       nodeTemplates=ENodeTemplates
30     END)? &
31   ('policies:'
32     BEGIN
33       policies=EPolicies
34     END)? &
35   ('outputs:'
36     BEGIN
37       outputs = EOutputs
38     END)?
39 ;
```

Figure 67: Snippet for AADM DSL grammar

From the DSL grammar, XText tools are used to generate default textual editors for the SODALITE IDE. Some of the textual edition features (e.g. context assistance, textual formatting, code

¹⁵ <https://spring.io/projects/spring-boot>

¹⁶ <https://github.com/FasterXML/jackson>

¹⁷ <https://github.com/google/gson>



serialization, etc.) are further extended by the SODALITE IDE development team and reported in following sections.

7.3.2 New and extended features (M18 - M33)

The following describes the technical implementation of the main SODALITE IDE features extended or developed during the reporting period.

7.3.3 Multiview representation of the AADM

Textual representation of AADM DSL (and other DSLs) is automatically created by XText out of the grammar definition. Specialized textual formatting is implemented by the SODALITE team in class *AADMFormatter* of project *org.sodalite.dsl.AADM* (similar formatting has been implemented for RMs). Outline representation from textual one is generated by class *AADMLabelProvider* of project *org.sodalite.dsl.AADM.ui*. Turtle serialization for sending models to the KB are supported by the class *AADMGenerator* of project *org.sodalite.dsl.AADM* (similar classes for other DSLs). Most of those classes are implemented using XTend¹⁸ language (i.e. a dialect of Java).

The visual representation, which offers a canvas based editable representation of the AADM, combined with editable forms for entity configuration, has been implemented using Sirius framework. Almost every aspect of the visual representation (including canvas notation, forms, and palette with edition tools) is declared in the *AADM.odesign* Sirius descriptor (see [Figure 68](#)). Sirius offers a visual tool for creating this descriptor. It declares the visual elements in the canvas, their associated toolbars (in palette) for element creation, and their associated forms in the properties view for element configuration. The entire code for this visual representation is automatically created by Sirius out of this descriptor.

Specialized Java code for content assistance associated to the field of the forms associated to entities of the graphical representation or for entity creation (from the palette) has been encoded in classes *Services* and *KBReasonerProxy* of the *org.sodalite.dsl.AADM.design* project. These helper classes are associated with visual forms in the *odesign* descriptor, and invoked when the user selects particular entities in the canvas or create new ones.

The synchronization between textual and graphical representations is automatically managed by both XText and Sirius frameworks, after some configuration. The correct formatting of AADM in the textual editor after changes propagated from the visual editor is managed by the aforementioned *AADMFormatter* class.

¹⁸ <https://www.eclipse.org/xtend/>

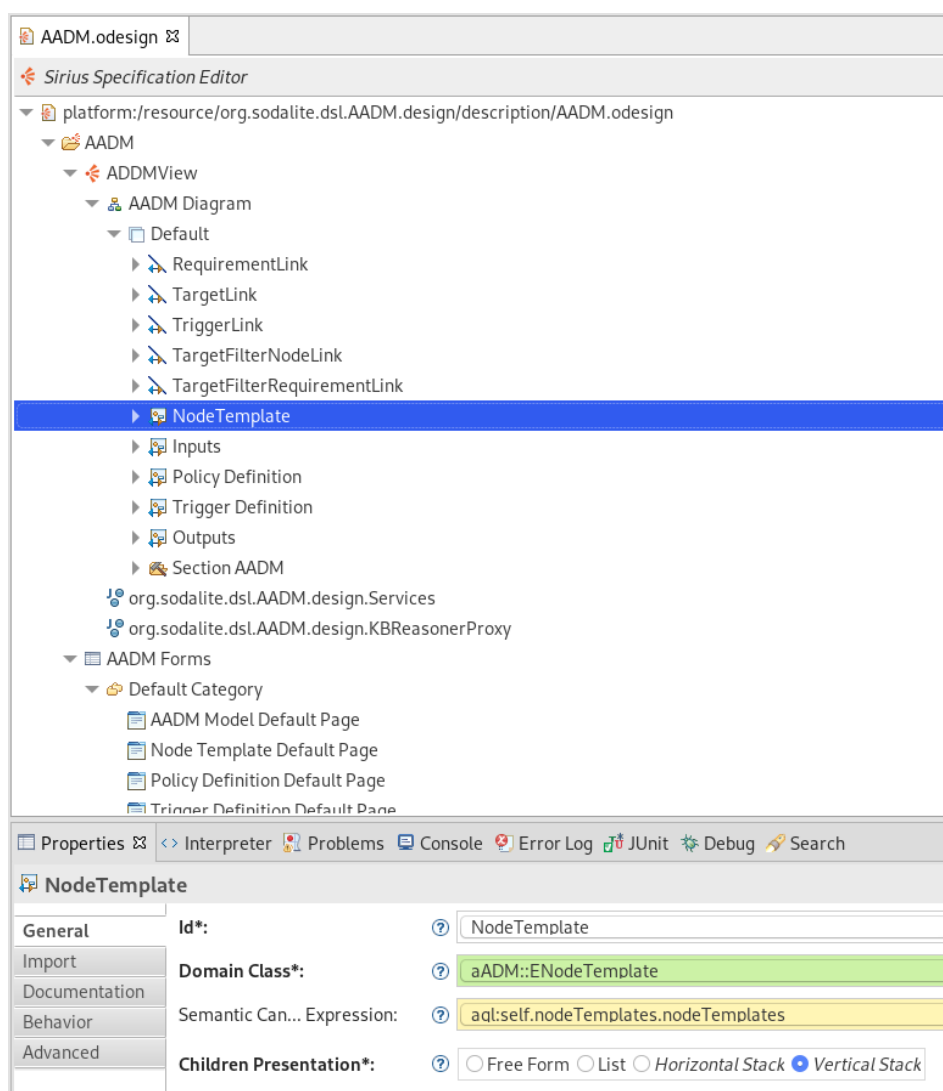


Figure 68: Sirius design of AADM visual representation

7.3.4 Extended AADM Deployment Support.

The extended AADM deployment feature consists of the deployment wizard, implemented in the package *org.sodalite.dsl.ui.wizards.deployment* of the project *org.sodalite.dsl.AADM.ui*, and of the deployment process itself, implemented in the class *AADMBackendProxy*. The wizard implementation uses Eclipse JFace¹⁹ and SWT²⁰ frameworks. The *AADMBackendProxy* relies on *org.sodalite.dsl.kb_reasoner_client* plugin for communication with the backend services that are supporting the deployment process.

7.3.5 Deployment Governance and KB Browser Views

These views, as for the SODALITE IDE wizards, have been implemented using JFace and SWT frameworks. The Deployment Governance view implementation is located in the *org.sodalite.ide.ui.views.deployment* package and the KB Browser view in the *org.sodalite.ide.ui.views.deployment* package, of the *org.sodalite.ide.ui* project. These views interact with the backend *Orchestrator* and *KB Reasoner*, respectively, through the proxy framework implemented in project *org.sodalite.dsl.kb_reasoner_client*.

¹⁹ <https://wiki.eclipse.org/JFace>

²⁰ <https://www.eclipse.org/swt/>



7.3.6 Improved Content Assistance

Context-aware content assistance for AADM and RM textual editors is implemented by customizing the default XText content assistance in Xtend classes *AADMPProposalProvider* and *RMProposalProvider*, of *org.sodalite.dsl.AADM.ui* and *org.sodalite.dsl.RM.ui* projects, respectively. Similar content assistance for Alerting Rule models is provided in the *AlertingProposalProvider* Xtend class of *org.sodalite.dsl.alerting.ui* project. These content providers interact with the backend *KB Reasoner*, through the proxy framework implemented in *org.sodalite.dsl.kb_reasoner_client* project.

7.3.7 Semantic Validation (Reporting and Quick fixes)

Semantic validation is implemented as part of the process that saves AADMs and RMs into the KB. It is implemented in the *AADMBackendProxy* and *RMBackendProxy* classes of the *org.sodalite.dsl.AADM.ui* and *org.sodalite.dsl.RM.ui* projects, respectively. Quick fixes for AADM are implemented by extending the default XText implementation in the *AADMQuickfixProvider* Xtend class of *org.sodalite.dsl.AADM.ui* project.

7.3.8 AADM Versioning

AADM Versioning is implemented as part of the process that saves AADMs into the KB. This is managed by the AADM saving wizard and associated backend process, but also by the KB Browser view (see below section). The wizard for saving AADMs is implemented in the *org.sodalite.dsl.ui.wizards.saveaadm* package of project *org.sodalite.dsl.AADM.ui* project. The process for saving AADMs is implemented in the *AADMBackendProxy* class of the *org.sodalite.dsl.AADM.ui* project.

7.3.9 IAM/Secrets Management

IAM-based communications from the IDE to the backend is managed by the proxy framework implemented in *org.sodalite.dsl.kb_reasoner_client* project. It gets from Keycloak a token for the user registered within the IDE. This token is used in any IDE-backend communication. The configuration of the IAM credentials and the management of the user's secrets stored in the Vault are supported through Eclipse preferences pages, which are implemented in the *org.sodalite.dsl.ui.preferences* and *org.sodalite.dsl.ui.preferences.hpc_secrets* packages of the *org.sodalite.dsl.preferences* project. The *org.sodalite.dsl.kb_reasoner_client* proxy cares for getting the KeyCloak token and manages the IAM-secured interactions with the backend services.

7.3.10 Image Builder and PDS Integration

These two features provide wizards that are implemented, by adopting JFace and SWT frameworks, in the *org.sodalite.dsl.ui.wizards.buildimages* and *org.sodalite.dsl.ui.wizards.pds* packages of the project *org.sodalite.dsl.AADM.ui* project, respectively. Their respective backend processes are implemented in the *AADMBackendProxy* class.

7.3.11 CI/CD Integration

Like for the other SODALITE components, the process for building the IDE and providing its Eclipse update site from where users can install a local standalone IDE instance, have been automated with the main CI/CD integration in SODALITE Jenkins²¹. The IDE update site is automatically updated after any commit to the master branch of the Sodalite repository²².

²¹ <https://jenkins.sodalite.eu>

²² <https://github.com/SODALITE-EU/ide>



8 Conclusion

This deliverable has reported on the final release (M33) of the SODALITE Modelling Layer, by emphasizing on the main new features that have been added since the initial release reported in D3.1, and also reporting the progress on the existing features. The work reported is relevant to T3.1 “Application Semantic Modelling” and T3.2 “Infrastructure Semantic Modelling”. The final version of the conceptual model was presented with all the new concepts that have been added. Also, the updated Semantic Reasoner has been presented that now provides new advanced reasoning services that foster the simplification of the model, the validation, the matchmaking, the reuse, and the context-aware content assistance. This module also provides the Semantic Populator that has been updated significantly for supporting new TOSCA concepts, the workspaces, the versioning of the AADMs, and for enabling components from other layers to save and matchmaking models and resources in the Knowledge Base. The IDE has significantly improved since the M18 release, by offering new DSLs and editors, wizards and views for supporting different modeling roles through the whole lifecycle of the deployment process.

Next steps include a PoC implementation of the IDE as a SaaS, and some subtle extensions in the reasoning services as part of WP4/T4.4 - Analytics and Semantic Decision Support task.

References

- [1] J. Opara-Martins, R. Sahandi and F.Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective", *Journal of Cloud Computing*, vol. 5, 2016.
- [2] D3.1, First version of ontologies and the semantic repository, SODALITE Technical Deliverable 2020.
- [3] D4.2, IaC Management - Intermediate version, SODALITE Technical Deliverable 2020.
- [4] T. Tudorache, "Ontology engineering: Current state, challenges, and future directions", *Semantic Web – Interoperability, Usability, Applicability an IOS Press Journal*, vol. 11, no. 1, pp. 125-138, 2019.
- [5] D. Fensel, U. Şimşek, K. Angele, E. Huaman, E. Kärle, O. Panasiuk, I. Toma, J. Umbrich, A. Wahler. (2020). *Knowledge Graphs: Methodology, Tools and Selected Use Cases*.
- [6] A. Gangemi and P. Mika, "Understanding the Semantic Web through Descriptions and Situations," in *Proceedings of ODBASE03 Conference*, Springer, Berlin, Heidelberg, pp. 689–706, 2003.
- [7] G.Meditkos, Z.Vasileiou, A.Karakostas, S.Vrochidis, I.Kompatsiaris, "A Pattern-based Semantic Lifting of Cloud and HPC Applications using OWL 2 Meta-modelling", 4th Special Session on High Performance Services Computing and Internet Technologies, 2020.
- [8] Jesús Gorroñogoitia, Zoe Vasileiou, Emilio Imperiali, Indika Kumara, Dragan Radolović and Georgios Meditskos, "A Smart Development Environment for Infrastructure as Code", *CEUR Workshop Proceedings*, 2021
- [9] E. Di Nitto et al., "An Approach to Support Automated Deployment of Applications on Heterogeneous Cloud-HPC Infrastructures," 2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNAS), pp. 133-140 2020.
- [10] D2.3, Requirements, KPIs, evaluation plan and architecture - final version, SODALITE Technical Deliverable 2021.
- [11] J.Soldani and P.Wang, "TOSCA in a nutshell: Promises and perspectives", *Service-Oriented and Cloud Computing*, vol. 8745, pp. 171-186, 2014.
- [12] TOSCA Simple Yaml profile
https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html#_Toc26969496
- [13] M. Cankar, A.Luzar, and D.Tamburri, "Auto-scaling Using TOSCA Infrastructure as Code", In book *Software Architecture, 14th European Conference, ECSA 2020 Tracks and Workshops*, pp. 260-268, 2020.
- [14] Knublauch, H., Kontokostas, D.: *Shapes Constraint Language (SHACL)*. W3C Proposed Recommendation, June 2017
- [15] M. Poveda-Villalón, A. Gómez-Pérez, & M.C. Suárez-Figueroa. OOPS! (Ontology Pitfall Scanner!): An on-line tool for ontology evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 10(2), pp. 7-34, 2014.
- [16] Lantow, B., "OntoMetrics: Putting Metrics into Use for Ontology Evaluation", pp. 186-191, 2016
- [17] Reiz, A.; Dibowski, H.; Sandkuhl, K. & Lantow, B., "Ontology Metrics as a Service (OMaaS)", 12th International Conference on Knowledge Engineering and Ontology Development, pp. 250-257, 2020.
- [18] Zhang, Qian and Haller, Armin and Wang, Qing, CoCoOn: Cloud Computing Ontology for IaaS Price and Performance Comparison, in book "The Semantic Web – ISWC 2019", pp. 325-341, 2019.
- [19] al-sayed, Mustafa and Omara, Fatma, "CloudFNF: An ontology structure for functional and non-functional features of cloud services", *Journal of Parallel and Distributed Computing*, 2020
- [20] B.Martino, A.Esposito, S. Nacchia, S. Maisto, U. Breitenbücher, "An Ontology for OASIS



TOSCA”, *Advances in Intelligent Systems and Computing* , vol. 1150, pp. 709-719, Springer 2020.

[21] F.Moscato, R. Aversa, Di Martino, B., Fortiş, T. and Munteanu, V., “An Analysis of mOSAIC ontology for Cloud Resources annotation.”, *Federated Conference on Computer Science and Information Systems*, pp. 973-980, 2011

[22] N. Bassiliades, M. Symeonidis, G. Meditskos, E. Kontopoulos, P. Gouvas, and I. Vlahavas, “PaaSport Semantic Model: An Ontology for a Platform-as-a-Service Semantically Interoperable Marketplace”, in *Data and Knowledge Engineering* , vol. 113, pp. 81-115, Elsevier, 2018.

[23] k. Yongsiriwit, Sellami M. and Gaaloul W., “A Semantic Framework Supporting Cloud Resource Descriptions Interoperability”, *IEEE 9th International Conference on Cloud Computing*, 2016, pp. 585-592

[24] D5.2, Application deployment and dynamic runtime - intermediate version. SODALITE Technical Deliverable 2021.

[25] A. Brogi, A. Di Tommaso, J. Soldani, “Sommelier: A Tool for Validating TOSCA Application Topologies”, In book: *Model-Driven Engineering and Software Development*, July 2018

[26] D2.1 Requirements, KPIs, evaluation plan and architecture - first version, SODALITE Technical Deliverable 2020.

[27] D2.2 Requirements, KPIs, evaluation plan and architecture - intermediate version, SODALITE Technical Deliverable 2020.



Appendix

In this section, we present the radon AADM example that was presented in [Chapter 2](#).

In particular, the DSL syntax, the intermediate exchange format and the final SODALITE meta-model definition are presented.

DSL representation

```
module: radon

import: docker
import: hpc

inputs:
  flavor-name:
    type: string
  vm-name:
    type: string
  image-name:
    type: string
    default: "image_name"

node_templates:
  workstation:
    type: toska.nodes.Compute
    description: "workstation description"
    attributes:
      private_address: "localhost"
      public_address: "localhost"

  openstack_vm:
    type: radon/radon.nodes.OpenStack.VM
    properties:
      name: get_input: vm-name
      image: get_input: image-name
      flavor: get_input: flavor-name
      network: 'provider_64_net'
      key_name: 'my_key'
    requirements:
      host:
        node: radon/workstation

policies:
```

```
scale_down:
  type: radon/radon.policies.scaling.ScaleDown
  description: "scale down policy description"
  properties:
    cpu_upper_bound: 90
    adjustment: 1

scale_up:
  type: radon/radon.policies.scaling.ScaleUp
  properties:
    cpu_upper_bound: 90
    adjustment: 1

autoscale:
  type: radon/radon.policies.scaling.AutoScale
  properties:
    min_size: 3
    max_size: 7
  targets: [ radon/openstack_vm]
  triggers:
    radon.triggers.scaling:
      description: 'A trigger for autoscaling'
      event: 'auto_scale_trigger'
      schedule:
        start_time: "2020-04-08 21:59:40"
        end_time: "2022-04-08 21:59:50"
      target_filter:
        node: radon/openstack_vm
        requirement: radon/openstack_vm.host
        capability: tosa.nodes.Compute.host
      condition:
        constraint:
          not:
            and:
              available_instances:[greater_than:
42]

available_space:[greater_than:1000]
          period: '60 sec'
          evaluations: 2
          method: 'average'
        action:
          call_operation:
            operation:
radon/radon.interfaces.scaling.AutoScale.retrieve_info
          call_operation:
```



operation:

radon/radon.interfaces.scaling.AutoScale.autoscale

outputs:

```
    public_ip:
      type: string
      description: 'The public IP of the provisioned VM'
      value: get_attribute:
        entity: radon/workstation
        attribute: radon/workstation.public_address
```

Exchange Model

```
# baseURI: https://www.sodalite.eu/ontologies/exchange/radon/
# imports: https://www.sodalite.eu/ontologies/exchange/

@prefix : <https://www.sodalite.eu/ontologies/exchange/radon/> .
@prefix exchange: <https://www.sodalite.eu/ontologies/exchange/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:
  rdf:type owl:Ontology ;
  owl:imports exchange: ;
  owl:versionInfo "Created by the SODALITE IDE" ;
.

:AADM_1
  rdf:type exchange:AADM ;
  exchange:userId "27827d44-0f6c-11ea-8d71-362b9e155667" ;
.

:Parameter_29
  rdf:type exchange:Parameter ;
  exchange:name "attribute" ;
  exchange:value 'radon/workstation.public_address' ;
.

:Parameter_30
  rdf:type exchange:Parameter ;
  exchange:name "entity" ;
  exchange:value 'radon/workstation' ;
.

:Parameter_28
```



```
    rdf:type exchange:Parameter ;
    exchange:name "get_attribute" ;
    exchange:hasParameter :Parameter_29 ;
    exchange:hasParameter :Parameter_30 ;
.
:Parameter_26
    rdf:type exchange:Parameter ;
    exchange:name "type" ;
    exchange:value 'string' ;
.
:Parameter_27
    rdf:type exchange:Parameter ;
    exchange:name "value" ;
    exchange:hasParameter :Parameter_28 ;
.
:Output_1
    rdf:type exchange:Output ;
    exchange:name "public_ip" ;
    exchange:hasParameter :Parameter_26 ;
    exchange:description 'The public IP of the provisioned VM' ;
    exchange:hasParameter :Parameter_27 ;
.
:Parameter_28
    rdf:type exchange:Parameter ;
    exchange:name "type" ;
    exchange:value 'string' ;
.
:Input_1
    rdf:type exchange:Input ;
    exchange:name "flavor-name" ;
    exchange:hasParameter :Parameter_28 ;
.
:Parameter_29
    rdf:type exchange:Parameter ;
    exchange:name "type" ;
    exchange:value 'string' ;
.
:Input_2
    rdf:type exchange:Input ;
    exchange:name "vm-name" ;
    exchange:hasParameter :Parameter_29 ;
.
:Parameter_30
    rdf:type exchange:Parameter ;
```



```
    exchange:name "type" ;
    exchange:value 'string' ;
.
:Parameter_31
    rdf:type exchange:Parameter ;
    exchange:name "default" ;
    exchange:value 'image-name' ;
.
:Input_3
    rdf:type exchange:Input ;
    exchange:name "image-name" ;
    exchange:hasParameter :Parameter_30 ;
    exchange:hasParameter :Parameter_31 ;
.
:Property_1
    rdf:type exchange:Property ;
    exchange:name "name" ;
    exchange:value "HostVM" ;
.
:Property_2
    rdf:type exchange:Property ;
    exchange:name "image" ;
    exchange:value "centos7" ;
.
:Property_3
    rdf:type exchange:Property ;
    exchange:name "flavor" ;
    exchange:value "m1.xsmall" ;
.
:Property_4
    rdf:type exchange:Property ;
    exchange:name "network" ;
    exchange:value "provider_64_net" ;
.
:Property_5
    rdf:type exchange:Property ;
    exchange:name "key_name" ;
    exchange:value "my_key" ;
.
:Property_6
    rdf:type exchange:Property ;
    exchange:name "cpu_upper_bound" ;
    exchange:value "10" ;
.
:Property_7
    rdf:type exchange:Property ;
```

```
    exchange:name "adjustment" ;
    exchange:value "1" ;
.
:Property_8
    rdf:type exchange:Property ;
    exchange:name "cpu_upper_bound" ;
    exchange:value "90" ;
.
:Property_9
    rdf:type exchange:Property ;
    exchange:name "adjustment" ;
    exchange:value "-1" ;
.
:Property_10
    rdf:type exchange:Property ;
    exchange:name "min_size" ;
    exchange:value "3" ;
.
:Property_11
    rdf:type exchange:Property ;
    exchange:name "max_size" ;
    exchange:value "10" ;
.
:Parameter_1
    rdf:type exchange:Parameter ;
    exchange:name "event" ;
    exchange:value 'auto_scale_trigger' ;
.
:Parameter_2
    rdf:type exchange:Parameter ;
    exchange:name "node" ;
    exchange:value 'radon/openstack_vm' ;
.
:Parameter_3
    rdf:type exchange:Parameter ;
    exchange:name "requirement" ;
    exchange:value 'radon/openstack_vm.host' ;
.
:Parameter_4
    rdf:type exchange:Parameter ;
    exchange:name "capability" ;
    exchange:value 'radon/openstack_vm.host' ;
.
:Parameter_5
    rdf:type exchange:Parameter ;
    exchange:name "target_filter" ;
```



```
exchange:hasParameter :Parameter_2 ;
exchange:hasParameter :Parameter_3 ;
exchange:hasParameter :Parameter_4 ;
.
:Parameter_6
  rdf:type exchange:Parameter ;
  exchange:name "greater_than" ;
  exchange:value "42" ;
.
:Parameter_7
  rdf:type exchange:Parameter ;
  exchange:name "available_instances" ;
  exchange:hasParameter :Parameter_6 ;
.
:Parameter_8
  rdf:type exchange:Parameter ;
  exchange:name "greater_than" ;
  exchange:value "1000" ;
.
:Parameter_9
  rdf:type exchange:Parameter ;
  exchange:name "available_space" ;
  exchange:hasParameter :Parameter_8 ;
.
:Parameter_10
  rdf:type exchange:Parameter ;
  exchange:name "and" ;
  exchange:hasParameter :Parameter_7 ;
  exchange:hasParameter :Parameter_9 ;
.
:Parameter_11
  rdf:type exchange:Parameter ;
  exchange:name "not" ;
  exchange:hasParameter :Parameter_10 ;
.
:Parameter_12
  rdf:type exchange:Parameter ;
  exchange:name "constraint" ;
  exchange:hasParameter :Parameter_11 ;
.
:Parameter_13
  rdf:type exchange:Parameter ;
  exchange:name "period" ;
  exchange:value '60 sec' ;
.
:Parameter_14
```



```
    rdf:type exchange:Parameter ;
    exchange:name "evaluations" ;
    exchange:value 2 ;
.
:Parameter_15
    rdf:type exchange:Parameter ;
    exchange:name "method" ;
    exchange:value 'average' ;
.
:Parameter_16
    rdf:type exchange:Parameter ;
    exchange:name "condition" ;
    exchange:hasParameter :Parameter_12 ;
    exchange:hasParameter :Parameter_13 ;
    exchange:hasParameter :Parameter_14 ;
    exchange:hasParameter :Parameter_15 ;
.
:Parameter_17
    rdf:type exchange:Parameter ;
    exchange:name "operation";
    exchange:value
    'radon/radon.interfaces.scaling.AutoScale.retrieve_info' ;
.
:Parameter_18
    rdf:type exchange:Parameter ;
    exchange:name "inputs" ;
.
:Parameter_19
    rdf:type exchange:Parameter ;
    exchange:name "call_operation" ;
    exchange:hasParameter :Parameter_17 ;
    exchange:hasParameter :Parameter_18 ;
.
:Parameter_20
    rdf:type exchange:Parameter ;
    exchange:name "action" ;
    exchange:hasParameter :Parameter_19 ;
.
:Parameter_21
    rdf:type exchange:Parameter ;
    exchange:name "operation";
                                exchange:value
'radon/radon.interfaces.scaling.AutoScale.autoscale' ;
.
:Parameter_22
    rdf:type exchange:Parameter ;
```




```
    exchange:name "inputs" ;
.
:Parameter_23
    rdf:type exchange:Parameter ;
    exchange:name "call_operation" ;
    exchange:hasParameter :Parameter_21 ;
    exchange:hasParameter :Parameter_22 ;
.
:Parameter_24
    rdf:type exchange:Parameter ;
    exchange:name "action" ;
    exchange:hasParameter :Parameter_23 ;
.

:Trigger_1
    rdf:type exchange:Trigger ;
    exchange:name "radon.triggers.scaling" ;
    exchange:description 'A trigger for autoscaling' ;
    exchange:hasParameter :Parameter_1 ;
    exchange:hasParameter :Parameter_5 ;
    exchange:hasParameter :Parameter_16 ;
    exchange:hasParameter :Parameter_20 ;
    exchange:hasParameter :Parameter_24 ;
.
:Attribute_1
    rdf:type exchange:Attribute ;
    exchange:name "private_address" ;
    exchange:value "localhost" ;
.
:Attribute_2
    rdf:type exchange:Attribute ;
    exchange:name "public_address" ;
    exchange:value "localhost" ;
.
:Parameter_25
    rdf:type exchange:Parameter ;
    exchange:name "node" ;
    exchange:value 'radon/workstation' ;
.
:Requirement_1
    rdf:type exchange:Requirement ;
    exchange:name "host" ;
    exchange:hasParameter :Parameter_25 ;
.
:Template_1
    rdf:type exchange:Template ;
```



```
exchange:name "workstation" ;
exchange:type 'tosca.nodes.Compute' ;
exchange:attributes :Attribute_1 ;
exchange:attributes :Attribute_2 ;
.
:Template_2
  rdf:type exchange:Template ;
  exchange:name "openstack_vm" ;
  exchange:type 'radon/radon.nodes.OpenStack.VM' ;
  exchange:properties :Property_1 ;
  exchange:properties :Property_2 ;
  exchange:properties :Property_3 ;
  exchange:properties :Property_4 ;
  exchange:properties :Property_5 ;
  exchange:requirements :Requirement_1 ;
.
:PolicyTemplate_1
  rdf:type exchange:Template ;
  exchange:name "scale_down" ;
  exchange:type 'radon/radon.policies.scaling.ScaleDown' ;
  exchange:properties :Property_6 ;
  exchange:properties :Property_7 ;
.
:PolicyTemplate_2
  rdf:type exchange:Template ;
  exchange:name "scale_up" ;
  exchange:type 'radon/radon.policies.scaling.ScaleUp' ;
  exchange:properties :Property_8 ;
  exchange:properties :Property_9 ;
.
:Parameter_26
  rdf:type exchange:Parameter ;
  exchange:listValue 'radon/openstack_vm' ;
.
:PolicyTemplate_3
  rdf:type exchange:Template ;
  exchange:name "autoscale" ;
  exchange:type 'radon/radon.policies.scaling.AutoScale' ;
  exchange:properties :Property_10 ;
  exchange:properties :Property_11 ;
  exchange:targets :Parameter_26 ;
  exchange:triggers :Trigger_1 ;
.
```



Sodalite Meta-model

The AADM example in the turtle format can be found [here](#).

Semantic Reasoner

Semantic Reasoning Engine (SRE)

During the second and third year of the project, the existing REST API endpoints, that assist the modellers, were enhanced, and additional APIs were developed for improving the recommendation and validation services.

Software Dependencies

- Java 11 or newer
- Jersey web services 2.32
- Graph DB 9.7.0 or newer
- Docker engine 19.03 or newer
- RDF4J v3.0.0
- Tomcat 9

Requirements

- OWL2 reasoning to be supported
- The interface to access the semantic repository to be provided
- The reasoning infrastructure for custom rule-based logic to be provided.

Composed Of

The REST API and the reasoning infrastructure by interfacing with the GraphDB reasoning engine.

Roles that interact with the component (i.e. AOE, RE, QE)

No direct interaction with the user.

Depends on

The Semantic Reasoning Engine depends on the Semantic Knowledge Base.

Docker image

https://hub.docker.com/r/sodaliteh2020/semantic_web

Repositories

<https://github.com/SODALITE-EU/semantic-reasoner>

Final version of the APIs

Many APIs are using the *template* parameter. The *template = false* is set by the IDE since the concepts from the type definitions are retrieved for assisting the user when authoring the templates. The *template=true* is sent when the reasoner calls the APIs for building the aadm json that will be used by the IaC builder for creating the blueprint.

Security has been added to all the APIs for making the framework more solid and safe for the user, thus all the APIs have been enhanced with the *token* parameter. Also, the parameters in bold express that they are required.

**KB browser view**

Those APIs enable the KB browser view in the IDE

GET /model(resource, namespace, uri, version, token)

When a resource and a namespace are given, the model to which the resource belongs is returned.

When a uri is provided, the specific model is returned, and if a version is also provided, the versioned model is returned.

Example: resource=snow-docker-host,

namespace=<https://www.sodalite.eu/ontologies/workspace/1/snow/>

Get the model in which a specific resource is contained

```
{
  "data": [
    {
      "uri":
https://www.sodalite.eu/ontologies/workspace/1/bk4u2bgdn6upi9p5pb9b21gfg5/AADM\_ogrckmjo34itv3",
      "createdBy":
https://www.sodalite.eu/ontologies/workspace/1/bk4u2bgdn6upi9p5pb9",
      "createdAt": "2021-09-28T15:31:45.282+03:00",
      "dsl": "DSL text",
      "name": "snow.aadm.ttl"
    }
  ]
}
```

GET /models(type, namespace, token)

The IDE users can get all the application and resource models (type=AADM|RM) in a specific workspace (namespace). If no namespace is given the global models are returned.

Example: type = RM, namespace = <https://www.sodalite.eu/ontologies/workspace/1/openstack/>

All the resource models to the openstack workspace are returned.

```
{
  "data": [
    {
      "uri":
https://www.sodalite.eu/ontologies/workspace/1/thhkbhvrg6ajr9hgj7ekmlhqd/RM\_hmhf4l4kl81gcipsrbrk3lubq5",
      "createdBy":
https://www.sodalite.eu/ontologies/workspace/1/thhkbhvrg6ajr9hgj7ekmlhqd/27827d44-0f6c-11ea-8d71-362b9e155667",
      "createdAt": "2021-08-31T14:08:15.841+03:00",
      "dsl": "\"DSL\"",
      "name": "\"openstack_vm.rm\"",
      "description": "specification of resources for Openstack VM"
    }
  ]
}
```

DELETE /delete(uri, version, hard, token)

This API is called from the IDE for deleting a model. When only the uri is sent, the specific model gets deleted. When the uri, the version and hard = false are given, then the specific versioned model gets deleted. When the uri, and hard = true are given, then all the versions of an AADM model are deleted.

Example: Uri=<an aadm uri>

```
{
  "success": {
    "text": "Successfully deleted the model"
  }
}
```

Context-assistance and reuse**GET /namespaces**

This API is called by the IDE for returning all the workspaces in the KB.

Example:

In this example, the *docker*, *openstack*, *snow*, *radon*, *test*, *batch*, and *hpc* are the private workspaces in the KB. The uri of each workspace is returned since each workspace in the KB is saved as a named graph identified by a URI.

```
{
  "data": [
    "https://www.sodalite.eu/ontologies/workspace/1/docker/",
    "https://www.sodalite.eu/ontologies/workspace/1/openstack/",
    "https://www.sodalite.eu/ontologies/workspace/1/snow/",
    "https://www.sodalite.eu/ontologies/workspace/1/radon/",
    "https://www.sodalite.eu/ontologies/workspace/1/test/",
    "https://www.sodalite.eu/ontologies/workspace/1/batch/",
    "https://www.sodalite.eu/ontologies/workspace/1/hpc/"
  ]
}
```

GET /types(type, imports, token)

This API returns all the types saved in the KB in the global workspace and the private workspaces (*imports*). The type can get the following values: capability, data, node, relationship, interface, policy. The type represents what kind of nodes should be returned.

Example: imports=*docker* , type=*node*

All the node types in the global and docker workspaces are returned.

```
{
  "data": [
    {
```



```

"https://www.sodalite.eu/ontologies/workspace/1/docker/sodalite.nodes.DockerVolume": {
    "label": "sodalite.nodes.DockerVolume",
    "type": {

"https://www.sodalite.eu/ontologies/tosca/tosca.nodes.SoftwareComponent":
    {
        "label": "tosca.nodes.SoftwareComponent"
    }
},
    "namespace": "https://www.sodalite.eu/ontologies/workspace/1/docker/"
}
},
...]
```

GET /templates(imports, token)

All the templates in workspaces denoted in the imports parameter are returned.

Example: imports = snow

In this example, all the templates saved in the snow workspace are returned. For brevity, only the *snow-docker-volume-masks* is included in the json message.

```

{
    "data": [
        {

"https://www.sodalite.eu/ontologies/workspace/1/snow/v1.0/snow-docker-volume-masks": {
    "label": "snow-docker-volume-masks",
    "type": {

"https://www.sodalite.eu/ontologies/workspace/1/docker/sodalite.nodes.DockerVolume": {
    "label": "sodalite.nodes.DockerVolume",
    "namespace":
"https://www.sodalite.eu/ontologies/workspace/1/docker/"
    }
},
    "namespace": "https://www.sodalite.eu/ontologies/workspace/1/snow/",
    "version": "v1.0"
}
}, ...
]
}
```

The following APIs have been described in D3.1, and just included here for showing that authentication was included (*token*). The `template = false` is set by the IDE since the concepts from the type definitions are retrieved for assisting the user when authoring the templates. The `template=true` when the reasoner calls all the below APIs for building the `aadm.json` that will be used by the IaC builder for creating the blueprint.

GET /attributes(**resource**, `template`, **token**)

GET /capabilities(**resource**, `template`, **token**)

GET /properties(**resource**, `template`, **token**)

GET /interfaces(**resource**, `template`, **token**)

GET /requirements(**resource**, `template`, **token**)

GET /operations(**resource**, `template`, **token**)

This API is working as the above APIs (attributes, properties etc.). All the relevant operations of an interface type are returned for informing the user while adding operations in a template. The *template* is a boolean parameter, if *true* operations of a template are returned, if *false* operations of a type are returned.

Example: `resource=tosca.interfaces.node.lifecycle.Standard`

All the operations in a TOSCA normative interface type are returned.

```
{
  "data": [
    {
      "https://www.sodalite.eu/ontologies/tosca/create": {
        "description": "Standard lifecycle create operation",
        "definedIn":
          "https://www.sodalite.eu/ontologies/tosca/tosca.interfaces.node.lifecycle.Standard"
      }
    },
    ...]
```

GET /operationsFromNamespaces(`imports`, **token**)

This API returns the operations from interface types for a given workspaces in the *imports* parameter.

Example: All the operations in the global workspace are returned.

```
{
  "data": [
    {
      "https://www.sodalite.eu/ontologies/tosca/add_source": {
        "description": "operation to notify the target node of a source node which
is now available via a relationship.",
        "definedIn":
          "https://www.sodalite.eu/ontologies/tosca/tosca.interfaces.relationship.Configure"
      }
    }
  ]
}
```

```
    },  
    ...  
  ]  
}
```

GET /prop-attr-names(resource, element, token)

It returns the names of properties or attributes of a template. This data helps the user to complete the *get_attribute*, and *get_property* within the interfaces of a template.

Example: element = [prop](#), resource = [snow/snow-vm](#)

All the properties of the snow-vm are returned

```
{  
  "data": [  
    "https://www.sodalite.eu/ontologies/workspace/1/openstack/security_groups",  
    "https://www.sodalite.eu/ontologies/workspace/1/openstack/username",  
    "https://www.sodalite.eu/ontologies/tosca/name",  
    "https://www.sodalite.eu/ontologies/workspace/1/openstack/key_name"  
    ...  
  ]  
}
```

GET /valid-requirement-nodes(requirement, nodeType, imports, token)

This API was presented in D3.1. In this deliverable, an improved version of this API is presented.

Returns nodes that satisfy a certain requirement, when defining a node template of type nodeType. Since the subsumption hierarchy can include more than one requirement definition, the lowest in the hierarchy is picked. Then, the nodes of this specific type are returned.

```
{  
  "data": [  
    {  
      "https://www.sodalite.eu/ontologies/workspace/1/snow/snow-vm": {  
        "label": "snow-vm",  
        ...  
      }  
    },  
    {  
      "https://www.sodalite.eu/ontologies/workspace/1/snow/snow-docker-host":  
      {  
        "label": "snow-docker-host",  
        ...  
      }  
    }  
  ]  
}
```



```
}
```

GET /valid-requirement-nodes-type(requirement, nodeType, imports, token)

This API was presented in D3.1. In this deliverable, an improved version of this API is presented. Additional to the requirement/node, the requirement/capability is taken into account for matching nodes that can satisfy the specific requirement.

explain

Example:

requirement=[host](#)

nodeType=[docker%2Fsodalite.nodes.DockerizedComponent](#)

imports=[docker](#)

```
{
  "data": [
    {
      "https://www.sodalite.eu/ontologies/tosca/tosca.nodes.Compute": {
        "label": "tosca.nodes.Compute"
      }
    },
    {
      "https://www.sodalite.eu/ontologies/workspace/1/docker/sodalite.nodes.DockerHost":
        {
          "label": "sodalite.nodes.DockerHost",
          "namespace": "https://www.sodalite.eu/ontologies/workspace/1/docker/"
        }
      }
  ]
}
```

GET /is-subclass-of(nodeTypes, superNodeType, token)

It returns a list of the types in the *nodeTypes* that are subclasses of the *superNodeType*. In such a way, the types that are not subclasses of *superNodeType* are filtered out.

Example: nodeTypes=[openstack/sodalite.nodes.OpenStack.VM](#), [tosca.nodes.Compute](#)

superNodeType=[tosca.nodes.Compute](#)

The given nodeTypes are subclasses of the superNodeType.

```
{
  "data": [
    "tosca.nodes.Compute",
  ]
}
```



```

"openstack/sodalite.nodes.OpenStack.VM"
]
}

```

GET /capability-from-requirement(resource, requirement, template, token)

It returns the capabilities of a template/type that are assigned within a requirement/node of the template/type.

Example: resource=[docker/sodalite.nodes.DockerizedComponent](#)

requirement=[network](#)

template=[false](#)

In this example, the *requirement/network/node* of the *docker/sodalite.nodes.DockerizedComponent* type is the *sodalite.nodes.DockerNetwork* type, and this API returns its capabilities, in which case is the network.

```

{
  "data": [
    {
      "https://www.sodalite.eu/ontologies/workspace/1/docker/network": {"definedIn":
"https://www.sodalite.eu/ontologies/workspace/1/docker/sodalite.nodes.DockerNetwork",
      "specification": {
        "type": {

"https://www.sodalite.eu/ontologies/tosca/tosca.capabilities.Network": {
          "label": "tosca.capabilities.Network"
        }
      },
      "valid_source_types": [

"https://www.sodalite.eu/ontologies/workspace/1/docker/sodalite.nodes.DockerizedComponent"
      ]
    }
  ]
}
}

```

GET /testReasoner

An API for checking that all the components are up and communicating with each other, namely the Semantic Reasoner, the Defect predictor (D4.2), and the KB.

Response

Successfully connected to both defect predictor and graph-db

GET /rm(**rmIRI**, **token**)

The json of a resource model is returned. This API is used by the IDE for creating the DSL of the models that are saved by the Platform Discovery Service (D4.2). In such a way, the Resource Expert's work gets enabled as the discovered models are rendered in the IDE.

GET /aadm(**aadmIRI**, version, refactorer, **token**)

This API has been described in D3.1. It returns the AADM JSON that will be used by WP4 for building the blueprint. Two new variables were added: the version and the refactorer. The version represents a specific version of a model. refactorer = false when this API is called by the IaC builder. refactorer = true when called by the Refactorer [D5.1?] , since the Refactorer does not need the resources definition in the json, but only the aadm information. The Refactorer needs the aadm json for checking if any modification can be applied in the model.

Semantic Population Engine (SPE)

The Semantic Population Engine has been significantly updated since the first year for mapping new concepts to the Ontologies, and for enabling other external components to retrieve and save from/to the KB.

Software Dependencies

- RDF4J v3.0.0
- Java 11 or newer

Requirements

The application/infrastructure models to be expressed in TOSCA or exchange ttl format.

Composed Of

Mapping services of:

- The RDF-based exchange model on the SODALITE ODP (*exchange model mapper*)
- The TOSCA model on the RDF-based exchange model (Step 1) that is transformed to the SODALITE ODP (Step 2). At Step 2, the *TOSCA mapper* uses the exchange model mapper.

Roles that interact with the component (i.e. AOE, RE, QE)

There is no direct interaction with a user.

Depends on

The Semantic Population Engine depends on the Semantic Knowledge Base.

Docker image

https://hub.docker.com/r/sodaliteh2020/semantic_web

Repositories

<https://github.com/SODALITE-EU/semantic-reasoner>

Final version of the APIs

POST /saveAADM(**aadmTTL**, **aadmURI**, **aadmDSL**, **complete**, **namespace**, **name**, **version**, **token**)

This API was presented in D3.1. An improved version of this API is presented in this deliverable. The IDE sends a request to this API for saving an AADM by sending various parameters. One of them is the *aadmTTL* which contains the model in a lightweight version of the SODALITE ODP, the exchange ttl. The *aadmURI* represents the identifier that the user can optionally define, otherwise a random *aadmURI* is created by the Semantic populator. The *complete* flag, when disabled, potential omissions in the aadm are returned as error/suggestions depending if the omissions are required or not. The complete flag, when enabled, no error or suggestions are returned to the user for the omissions, but the Semantic Reasoner itself autofills the model. The *namespace* parameter represents the name of the workspace where the model will be saved, if no namespace is given, the model is saved in the global workspace. The *version* can optionally be given for saving a versioned model. The Semantic Population Engine, that translates the exchange model into the conceptual model of SODALITE and populates the KB, has been significantly updated for supporting new TOSCA constructs, the workspaces and the versioning of the aadm.

Example:

aadmTTL=<aadm exchange TTL>, aadmDSL=<aadm in DSL>, complete = **false**, namespace=snow, name=snow.aadm

```
{
  "suggestions": [],
  "uri":
  "https://www.sodalite.eu/ontologies/workspace/1/76o42nobs2m283sa9pavaah0ug/AADM_csac1260pqnp9",
  "version": "",
  "modifications": []
}
```

POST /optimizations(**aadmTTL**, **aadmURI**, **aadmDSL**, **complete**, **namespace**, **name**, **version**, **token**)

This API is responsible for saving a model that contains at least one template associated with an optimization DSL and returning suggestions related to optimizations. According to the capabilities offered by the host of the optimized application, suggestions for the optimizations are returned to the user. It takes the same parameters with the /saveAADM API. The only difference is that optimizations are checked for potential suggestions.

Example: aadmTTL=<aadmTTL>, aadmDSL=<aadmDSL>, namespace=optimization, name=optimization.aadm

In this example, since the num_gpus found in the optimisation DSL is greater than zero, it is proposed the glow be enabled.

```
"templates_optimizations": [
  {
    "type": "OptimizationMismatch",
    "info": {
      "path": "{\"app_type-ai_training\":{\"ai_framework-pytorch\":{}}}",

```



```

    "context":
    "https://www.sodalite.eu/ontologies/workspace/1/c9nsheajclsa0lv7hbc17jmka0/skyline_extractor",
    "description": "\"false\" (given value) != \"true\" (expected value)",
    "value": "{\"glow\":true}"
  }
}

```

POST /saveRM(**rmTTL**, **rmURI**, **rmDSL**, namespace, **name**, **token**)

This is a new API that saves a Resource Model to the KB. The *rmTTL* contains the model in the exchange ttl format. If the *rmURI* is not provided, a random URI is created for identifying the model. The *rmDSL* is sent for the model to be displayed in the IDE when retrieved. The *namespace* contains the name of the workspace where the model will be stored. The *name* contains the file name of the model.

Example: rmTTL=<rm exchange ttl>, rmDSL=<rm in DSL>, namespace=kube, name=kube_cluster.rm

In this example, we see that the response contains one warning regarding a bug detected by the Defect Predictor [D4.1], and the uri identifying the model.

```

{
  "warnings": [
    {
      "type": "HardcodedSecret",
      "info": {
        "name": "username",
        "context": "sodalite.nodes.Kubernetes.Cluster",
        "description": "The password or user name is hardcoded",
        "element_type": "Property"
      }
    }
  ],
  "uri":
  "https://www.sodalite.eu/ontologies/workspace/1/bjp5v3qrfbagomfj6c2rr4vj2s/RM_3hch4qq2j24pk61i1l7ourker8"
}

```

POST /saveTOSCA(**modelTOSCA**, **aadmURI**, **rmURI**, rmNamespace, aadmNamespace, rmName, aadmName)

This is a new API that is called by the Platform Discovery Service (D4.2) for saving discovered TOSCA Resources such as virtual machines, docker hosts etc. The Semantic Population Engine transforms the TOSCA model to Java objects, the Java objects to the exchange TTL, and then the exchangeTTL to the SODALITE ontologies. *modelTOSCA* can include both a resource model and an AADM. The *aadmURI* and *rmURI* can be optionally given otherwise the Semantic Population produces a random URI. The *rmNamespace* indicates to which workspace the resource model is saved, if not given the model is saved in the global workspace. The *aadmNamespace* indicates to which workspace the AADM is saved, if not given the model is saved in the global workspace. The



rmName and *aadmName* will be optionally given for identifying the name of the resource model file, and the *aadm* file. Finally, the backend services performing the transformation from the java objects to exchange ttl are used by the *Refactorer* - D5.2(17) for saving the refactored models to the KB.

Example: `modelTOSCA=<TOSCA models>`, `rmNamespace=test`, `aadmNamespace=test`, `rmName=test.rm`, `aadmName=test.aadm`

An example with a TOSCA model with both a RM and an AADM that is called by the PDS for being saved in the KB. The response returns the uri identifiers of both the models.

```
{
  "aadmuri":
  "https://www.sodalite.eu/ontologies/workspace/1/o85g3odqf54e7plqk9dlapf9v9/AADM_n06f8rk4fjrf2uia87k0dvd4ib",
  "rmuri":
  "https://www.sodalite.eu/ontologies/workspace/1/ls1tkbisvk5t5p3fhrtkhluohc/RM_7mc4jf40iufe7equo7esbpq0ph"
}
```

RDF Triple Store

Software Dependencies

- RDF4J v3.0.0
- Java 11 or newer
- GraphDB 9.7.0
- Windows 10

Requirements

- To allow remote access (HTTP protocol)
- To support storing, querying, management of Structured Data
- To support existing Semantic Web standards (RDF, OWL2, SPARQL)
- To provide a SPARQL endpoint.
- To support native RDF/OWL2 RL Reasoning

Composed Of

GraphDB (third-party software) is used as the underlying RDF triple store of SODALITE.

Roles that interact with the component (i.e. AOE, RE, QE)

There is no direct interaction with a user.

Depends on

N/A



Docker image

https://hub.docker.com/r/sodaliteh2020/graph_db

Repositories

It is a third-party standalone component. There is no repository.

Domain Models

Software Dependencies

- N/A

Requirements

- The necessary knowledge structures and vocabularies to be provided for modelling RMs and AADMs.

Composed Of

RDF graphs in different formats (e.g. Turtle, RDF/XML)

Roles that interact with the component (i.e. AOE, RE, QE)

N/A

Depends on

N/A

Repositories

<https://github.com/SODALITE-EU/semantic-models>