Preface

In the last few years more and more companies are organizing their software business adopting the "as a service" approach, that is, offering it for usage but keeping the control of its operation entirely within the boundaries of the company itself. This shows a number of advantages both from the technical and business perspectives and, at the same time, requires new ways of organizing and managing the lifecycle of software. The DevOps movement was born to address this requirement and highlights the importance of ensuring that operation and development are well coordinated and work together in a seamless way.

In this context, researchers and practitioners have started exploring the possibility of automating resource provisioning, software deployment, and ops-specific operations thanks to the development of *Infrastructural software*. Such infrastructural software, also called *Infrastructure as Code* or *IaC*, can be treated as any other type of software and, therefore, subject to versioning, debugging, verification, reuse, and the like. Nowadays, multiple IaC languages and frameworks have been developed, each of which has its own peculiarities, focuses on specific tasks – from provisioning to configuration, deployment and operation –, is able to manage specific resources, and requires a significant language-specific and operation-specific knowledge.

The SODALITE H2020 project aims at offering a contribution to this endeavor by offering a smart development environment for IaC and corresponding tools to support the automation of various deployment and operation phases. The aim of this book is to describe the approach and toolset proposed by this project. It presents also some case studies that show the practical utility of the approach and hints for the usage of SODALITE in other cases. The book is structured as follows:

- Chapter 1 provides an overview of the motivations for the development of a new approach and framework in the area of IaC, of the state of the art, and of the challenges to be addressed. Moreover, it describes the innovations introduced by SODALITE.
- Chapter 2 presents a general overview of the SODALITE approach and of the workflows it supports.

- Chapter 3 focuses on the problem of defining deployment models for complex application and presents the solution proposed by SODALITE in this context.
- Chapter 4 presents our approach to ensure the quality of IaC and to create optimized execution containers for components running on High Performance Computing (HPC) clusters.
- Chapter 5 presents the SODALITE runtime environment, including the features to support adaptation of running systems and identification of refactoring possibilities for the IaC associated to a certain software.
- Chapter 6 describes how the SODALITE approach can be used to deploy complex software across various platforms in the Cloud, Edge, and HPC domains, with a specific focus on OpenStack, AWS EC2, Kubernetes, and PBS Torque/Slurm.
- Chapter 7 describes the case studies in which the SODALITE approach has been used.
- Finally, Chapter 8 concludes the book by providing an overview of how the SODALITE results are being packaged in ready-to-use tools and then discussing about future research challenges.

Acknowledgments: Together with all authors of this book we are indebted to all other colleagues who joined forces with us for part of the SODALITE endeavor, to our external advisory board members, Nicolas Ferry, Andreas Metzger, Mark Parsons for their comments and new ideas and to our project reviewers for their continuous push that helped us increasing the quality and effectiveness of project results. Moreover, we are very thankful to our project officers, Luis Carlos Busquets Pérez, and Stefano Foglietta, who supported us in all phases of the project.

The work reported in this book is partially funded by the European Commission within the Horizon 2020 research and innovation programme, under grant agreement number 825480.

March 2022

Elisabetta Di Nitto
Jesús Gorroñogoitia Cruz
Indika Kumara
Dragan Radolović
Kamil Tokmakov
Zoe Vasileiou

Contents

Chapter 1
Orchestrating Heterogeneous Applications: Motivation and State of the Art

Elisabetta Di Nitto and Daniel Vladušič

Abstract This chapter presents the motivation for SODALITE highlighting the difficulties faced by developers of complex applications when they need to deploy such applications in execution contexts where the usage of heterogeneous resources (HPC, Cloud and Edge) coexist. An overview of the state of the art to highlight gaps and open issues is also presented.

1.1 Preliminaries

In recent years, the global market has seen a tremendous rise in utility computing, which serves as the backend for practically any new technology, methodology or advancement from healthcare to aerospace. General purpose GPUs are becoming common currency in datacentres while specialized FPGA accelerators, ranging from deep-learning specific accelerators to burst buffers technologies, are becoming "the big coin", enormously speeding up applications execution and likely to become common in the near future. We are entering a new era of heterogeneous, software-defined, high-performance computing environments. In this context, SODALITE aims to address this heterogeneity by focusing on how to deploy and operate complex software into environments that comprise accelerators/GPUs, configurable processors, and non-x86 CPUs such as ARMv8.

In our view, a complex software system is composed of several and different components built for different purposes, featuring different execution models (from microservices to batch jobs) and requiring different QoS. For example, consider a web application that runs an AI inference algorithm to recognize specific objects

Elisabetta Di Nitto
Politecnico di Milano, Italy, e-mail: `elisabetta.dinitto@polimi.it`

Daniel Vladušič
XLAB, Slovenia, e-mail: `daniel.vladusic@xlab.si`

within some images or to identify the products that a certain user will, likely, prefer. In this case, a heterogeneous setting would be the best choice for deploying such an application. More specifically, the microservices and web server will find their optimal configuration on the cloud, while at least part of the inference algorithm or its training phase may run more effectively on an HPC cluster based, for instance, on GPUs.

Having the application to be executed in a heterogeneous infrastructure can bring several advantages in terms of efficient use of the available resources and effective execution of the system. Nevertheless, being able to effectively deploy and operate application components in a heterogeneous environment today requires an in-depth knowledge of each target infrastructure, of the execution models each of them supports, and of the mechanisms that can be exploited to efficiently enable information exchange between the application parts deployed on different types of resources.

In general, Infrastructure as Code approaches do support effective deployment of applications but, at the same time, highlight a number of challenges. In the next sections, we will provide a brief analysis of the state of the art in the main relevant areas of modelling, deploying and operating complex applications (Section 1.2), we will then highlight the challenges that are left open by the available approaches (Section 1.3) and, finally, we will highlight the main innovations offered by SODALITE to cope with these challenges (Section 1.4).

1.2 State of the art analysis

1.2.1 Application Deployment Modelling

Approaches supporting application deployment assume that the application DevOps team develops a deployment model, that is, a specification of the components belonging to the application and their connectors, as well as their dependencies on a specific technological stack, if any. Infrastructure as Code approaches, such as TOSCA [9] and Ansible [1] do offer effective means to specify a deployment model. When this model is available, then an orchestrator can execute it and deploy the corresponding components on the available resources.

TOSCA is a standard IaC language that was designed to support a Cloud information model that can be extended through the definition of new node types and through inheritance. TOSCA itself is implementation agnostic. This means that the implementation of operations aiming at controlling the lifecycle of nodes (e.g., creation, scaling, deletion, . . .) can be defined in a wide spectrum of languages ranging from bash scripts and Python to infrastructure management tools like Chef, Puppet or Ansible. These three are all open-source tools mostly designed to help DevOps configure and manage the infrastructure. Both Chef and Puppet have been designed as an Agent-master solution and thus need agents installed on each node for

[1] https://www.ansible.com/

configuration. The offered IaC language is Ruby-like, which is usually considered difficult to learn. Ansible offers a simple and clean declarative IaC language which is widely accepted and easy to learn and adopt. Also, Ansible is characterized by a vast community support and probably the largest set of cloud infrastructure libraries support (Ansible Galaxy). Ansible is an inherently simple agentless approach to remote infrastructure management and is implemented through the standard Python Paramiko SSH library enabling the DevOps to manage any infrastructure accessible through SSH.

Wurster et. al. [14] propose the concept of the essential deployment metamodel (EDMM) that captures the essential parts of declarative deployment models. In a recent survey, Bergmayr et al. [2] reviewed the current approaches to modeling cloud applications. They observed that existing modeling languages lack interoperability, and, to cope with this, suggested to leverage the TOSCA standard. In [16], they identified an EDMM-compliant subset of TOSCA, to enable the transformation from TOSCA-based specifications of deployment models to those in the languages used by the industrial infrastructure as code (IaC) tools such as Ansible and Terraform.

As observed in [15, 2], there are several graphical modeling tools (IDEs) in existence for cloud infrastructure and deployment modeling, for example, Vino4TOSCA [5] and OCCIware [17] provide visual notations for TOSCA and OCCI (Open Cloud Computing Interface) modeling elements, respectively. OCCI is a standard for managing any cloud resources. In contrast, ARGON [12, 11], DICER [1], and SWITCH [13] provide the domain specific languages (DSLs) tailored to specific application domains, for example, public cloud infrastructures including their elasticity, data-intensive (big data) applications, and containerised microservice-based cloud-native applications.

1.2.2 Application Deployment and Operation

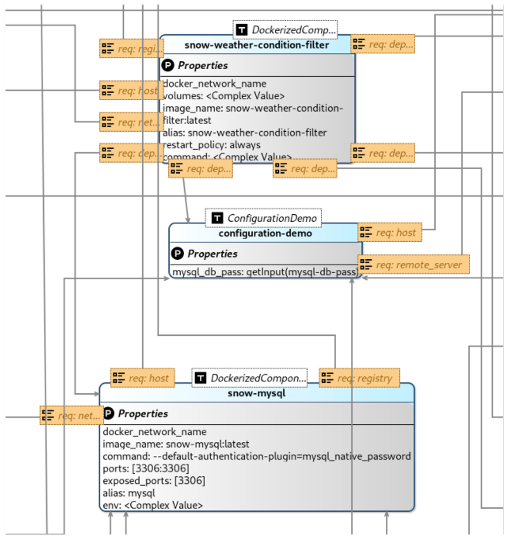
A common approach to enacting high-level or visual deployment models is to transform them into artifacts that can be used by an orchestrator or deployment automation tool. For example, ARGON and DICER employed model-to-model (M2M) transformations to convert the models in their DSLs into deployable IaC artifacts, for example, TOSCA blueprints and Ansible. Brabra et al. [4] also applied M2M transformations to transform TOSCA-based models into Docker and Docker compose configurations. Bernal et al. [3] proposed a UML profile to model the key elements of a cloud application and infrastructure, and used M2M transformations to translate UML-based application models to the configuration files for a cloud simulator, which enables the analysis of the performance of the application.

For what concerns the enactment of IaC, there exist TOSCA and OCCI based orchestrators or runtime environments for cloud applications [2, 17, 13], including multi-cloud [8, 7]. Two interesting approaches that focus on hybrid cloud and HPC applications are Croupier [6] and INDIGO [10]. Croupier is not fully compatible with the official TOSCA standard as it uses its own adaptation of the TOSCA model.

The INDIGO PaaS Orchestrator[2] allows instantiation of resources on the hybrid virtualized infrastructures (private, public clouds, virtual grid organizations) with the use of TOSCA YAML Simple Profile v1.0. It is integrated with other INDIGO services to enable best placement of the resources based on SLA and monitoring from the available list of cloud providers. In order to deploy, configure and update IaaS resources, the orchestrator uses an Infrastructure Manager (IM) that interfaces with multiple cloud sites in a cloud-agnostic manner. Although the INDIGO PaaS orchestrator allows to spin up a virtual cluster (e.g. managed by batch systems such as PBS Torque/Slurm/Mesos) using TOSCA, the workflow management of the jobs is not directly supported and it assumes the usage of workflow management systems (e.g., Kepler) on top of deployed virtual infrastructure. Similarly, the partial reconfiguration is done on IaaS resources and it does not operate on the application.

1.3 Open Challenges

From the brief overview in the previous section, it should be clear that approaches supporting the specification of deployment models and their execution to orchestrate the deployment of complex applications do exist and they include also TOSCA, a standardization effort that is raising the interest of multiple organizations both in academia and industry. However, when exploiting such approaches, a number of challenges must be faced.



```
docker_network_name
volumes: <Complex Value>
image_name: snow-weather-condition-filter:latest
alias: snow-weather-condition-filter
restart_policy: always
command: <Complex Value>
```

```
mysql_db_pass: getInput(mysql-db-pass)
```

```
docker_network_name
image_name: snow-mysql:latest
command: --default-authentication-plugin=mysql_native_password
ports: [3306:3306]
exposed_ports: [3306]
alias: mysql
env: <Complex Value>
```

Fig. 1.1: Graphical representation of the SNOW deployment model.

[2] https://github.com/indigo-dc/orchestrator

First, *defining a proper deployment model for a complex application is not an easy task*. As an example, Figure 1 shows a small portion of a deployment model that describes some components from SNOW, one of the SODALITE use cases. The description exploits the SODALITE Domain Specific Language, but any of the available IaC approaches would provide similar results. The figure is incomplete and refers only to three out of the about 10 components of the whole architecture. The lines in the figure show various kinds of relationships between the components of the SNOW architecture. Capturing all of them, together with all the needed properties is mandatory to enable the automation of the application deployment and configuration and gives an idea of the complexity of the specification effort. The problem we see is that current approaches do not provide guidance to the developers of such models that, as a consequence, must be very experienced.

Second, even when an expert able to master TOSCA and Ansible, or any other similar IaC approach, is available, still *this expert will need to have at his/her disposal the specification of the resources to be used for deployment*. In fact, every resource is assumed to be specified before its usage. This specification should include many peculiarities and details that vary from provider to provider, especially when we want to ensure optimized performance of the application to be executed. In some cases, the amount of available resources is not even known in advance and must be discovered on the fly. This is especially the case when using edge devices.

Finally, *every new type of resource, even different traditional cloud IaaS, offer different APIs and different access control mechanisms*. Thus, exploiting such resources and monitoring them and being able to adapt the application based on their status is, per se, a non-trivial task, even if nowadays these are supported by various experimental orchestrators and initiatives.

1.4 Innovations offered by SODALITE

SODALITE tries to address the problems described in the previous section by providing intelligent assistance during the deployment model creation phase and in enabling the end users to include in a deployment model pieces of information suitable to support the definition of QoS constraints, the optimization of used resources and a proper configuration of the execution and monitoring environment.

Moreover, SODALITE supports resource experts in modeling their resources and in automating the process of discovering new resources and deriving suitable models for them.

It offers light-weight execution environments, which are essentially cross-platform containers that enable the user to execute, with different performance, the same application components on heterogeneous resources in a seamless way and allow them to be built automatically.

Another important aspect concerning SODALITE is enabling design-time optimization of applications. To exploit HPC resources in the best possible way, the application code may need to be tuned and/or scaling actions may need to be executed (e.g., increasing the number of cores, accelerating with GPUs or coprocessors, enabling faster storage, etc.). Such actions must be tailored considering the type of application components to be deployed, their QoS requirements and the available resources. The SODALITE Application Optimizer, MODAK, focuses on these issues and offer a framework that, given the specification of a few constraints as part of a deployment model, is able to generate the scripts to be executed in an HPC environment to achieve an optimized execution of application components.

SODALITE supports also the identification of defects in deployment models and of possible reconfiguration options of running application configurations. Thanks to machine learning, SODALITE analyses the previous history of deployment models that had to be corrected to identify defects, thus building a taxonomy of defects that is then used to provide suggestions to DevOps experts. Defects include code smells, errors and anti-patterns.

At runtime, SODALITE enables on the fly optimization of applications by dynamically scaling in and out computational resources depending on the specific applications being considered, but also by identifying, through machine learning, possible configurations that perform better than others and suggest them to DevOps experts when the monitoring system reveals the presence of problems in the current configuration

Another considered aspect concerns the support to data placement-aware deployment and to data movement between HPC, Cloud and edge resources. These aspects are very important as they have a strong impact on application performance. SODALITE optimizes data movement at two different levels: single components and compositions of multiple components. For the former case, we explore asynchronous data transfer, caching, and prefetching of the data. For the latter case, we explore using efficient data movement across storage and network to improve the workflow performance.

Providing proper identity and access management is a crucial part of protecting both user data and sensible project information. There are two different facets we consider in the scope of SODALITE. The first one concerns the mechanisms that control access to the SODALITE platform itself. This is covered through a role-based Identity and Access Management (IAM) implementation (keycloak) for SODALITE users and other implementations for secret and credential management (e.g., Vault or similar). The second aspect concerns the possibility to model privacy and security-related resources, such as Virtual Private Networks, so that they can be instantiated and reused in the deployment models of specific systems and, thus, their deployment and configuration be automated as well.

1.5 Book objectives

The objective of this book is to: i) present the approach and tools constituting the SODALITE solution, ii) to describe how the approach can be used by a DevOps team, and iii) how it has been adopted by the three SODALITE case studies.

References

[1] M. Artac et al. “Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach”. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 156–15609.

[2] Alexander Bergmayr et al. “A Systematic Review of Cloud Modeling Languages”. In: *ACM Comput. Surv.* 51.1 (Feb. 2018). ISSN: 0360-0300. DOI: `10.1145/3150227`. URL: `https://doi.org/10.1145/3150227`.

[3] Adrián Bernal et al. “Improving cloud architectures using UML profiles and M2T transformation techniques”. In: *The Journal of Supercomputing* 75.12 (2019), pp. 8012–8058.

[4] H. Brabra et al. “Model-Driven Orchestration for Cloud Resources”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 422–429.

[5] Uwe Breitenbücher et al. “Vino4TOSCA: A visual notation for application topologies based on TOSCA”. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2012, pp. 416–424.

[6] J. Carnero and F. J. Nieto. “Running Simulations in HPC and Cloud Resources by Implementing Enhanced TOSCA Workflows”. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 431–438.

[7] József Kovács and Péter Kacsuk. “Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures”. In: *Journal of Grid Computing* 16.1 (Mar. 2018), pp. 19–37. ISSN: 1572-9184. DOI: `10.1007/s10723-017-9421-3`. URL: `https://doi.org/10.1007/s10723-017-9421-3`.

[8] Kyriakos Kritikos, Paweł Skrzypek, and Feroz Zahid. “Are Cloud Platforms Ready for Multi-cloud?” In: *Service-Oriented and Cloud Computing*. Ed. by Antonio Brogi, Wolf Zimmermann, and Kyriakos Kritikos. Cham: Springer International Publishing, 2020, pp. 56–73. ISBN: 978-3-030-44769-4.

[9] Paul Lipton et al. “Tosca simple profile in YAML version 1.3”. In: *OASIS Committee Specification* 1 (2020).

[10] Davide Salomoni et al. “INDIGO-DataCloud: A platform to facilitate seamless access to e-infrastructures”. In: *Journal of Grid Computing* 16.3 (2018), pp. 381–408.

[11] J. Sandobalin, E. Insfran, and S. Abrahao. “An Infrastructure Modelling Tool for Cloud Provisioning”. In: *2017 IEEE International Conference on Services Computing (SCC)*. 2017, pp. 354–361.

[12] Julio Sandobalin, Emilio Insfran, and Silvia Abrahao. “ARGON: A Tool for Modeling Cloud Resources”. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Ed. by Lars Braubach et al. Cham: Springer International Publishing, 2018, pp. 393–397. ISBN: 978-3-319-91764-1.

[13] Polona Štefanič et al. “SWITCH workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native ap-

plications". In: *Future Generation Computer Systems* 99 (2019), pp. 197–212. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.04.008. URL: http://www.sciencedirect.com/science/article/pii/S0167739X1831094X.

[14] Michael Wurster et al. "The EDMM Modeling and Transformation System". In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Ed. by Sami Yangui et al. Cham: Springer International Publishing, 2020, pp. 294–298. ISBN: 978-3-030-45989-5.

[15] Michael Wurster et al. "The essential deployment metamodel: a systematic review of deployment automation technologies". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 63–75. DOI: 10.1007/s00450-019-00412-x. URL: https://doi.org/10.1007/s00450-019-00412-x.

[16] Michael Wurster. et al. "TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies". In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,* INSTICC. SciTePress, 2020, pp. 216–226. ISBN: 978-989-758-424-4. DOI: 10.5220/0009794302160226.

[17] Faiez Zalila, Stéphanie Challita, and Philippe Merle. "Model-driven cloud resource management with OCCIware". In: *Future Generation Computer Systems* 99 (2019), pp. 260–277. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.04.015. URL: http://www.sciencedirect.com/science/article/pii/S0167739X18306071.

Chapter 2
The SODALITE Approach: an Overview

Luciano Baresi, Elisabetta Di Nitto and Daniel Vladušič

Abstract This chapter presents the main characteristics of SODALITE to give the reader an overall picture, which will be detailed in the following chapters.

2.1 Introduction

As discussed in the previous chapter, SODALITE focuses on the configuration, deployment and operation of complex applications. Often these are developed by specialists of particular application domains and particular development technologies that, however, are not necessarily expert of the resources from which applications could benefit for their execution. This implies that for such teams it is not easy to take care of IT-intensive tasks such as handling the deployment of complex applications on multiple heterogeneous infrastructures, making this process repeatable with no errors, fine tuning the execution of applications in order to keep performance and costs under control.

There are many evidences of the complexity of such tasks that have lead to the introduction of the DevOps lifecycle, to reinforce the importance and the advantages of a good cooperation between Dev and Ops, and to the emergence of the Infrastructure as Code (IaC) paradigm, which implies the possibility to write software that defines the way applications should be deployed, configured and executed.

While the literature presents several approaches that support some DevOps and IaC activities in a cloud environment, the main novelty of SODALITE is essentially

Luciano Baresi
Politecnico di Milano, Italy, e-mail: luciano.baresi@polimi.it

Elisabetta Di Nitto
Politecnico di Milano, Italy, e-mail: elisabetta.dinitto@polimi.it

Daniel Vladušič
XLAB, Slovenia e-mail: daniel.vladusic@xlab.si

to create a complete framework tackling multiple DevOps aspects and targeting multiple types of resources.

2.2 SODALITE main features

The envisaged platform is supposed to serve different users, who are experts in different aspects, related to the definition and operation of software-defined computing infrastructures, and requires resources to carry out the different activities. Figure 2.1 presents the use cases served by the features provided by SODALITE. Before discussing the use cases, we must introduce the different actors with which they interact. These can be grouped in *human actors* and *resources*.

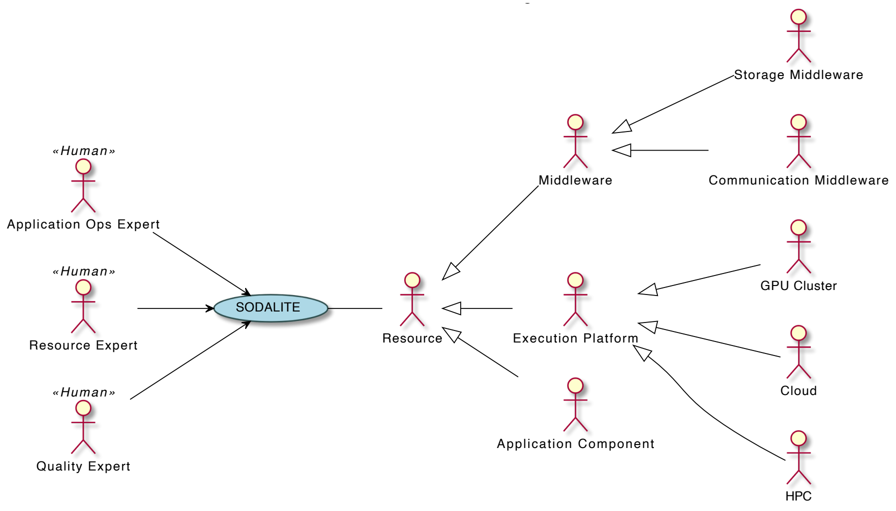


Fig. 2.1: Actors and use cases.

The envisaged human users are the following. To ease their harmonization in the context of a standard life-cycle, they can also be mapped onto the roles in charge of some of the processes defined in the ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes [1]:

- **Application Ops Expert** (AOE) is in charge of operating the application and, as such, is in charge of all the aspects that refer to the deployment, execution, optimization and monitoring of the application. He/she is supposed to know the applications to execute and the requirements on both the deployment/execution environment and the quality of services he/she is interested in. This role can correspond to ISO/IEC/IEEE role in charge of Operation processes and maintenance processes as they focus on the day-by-day operation.

- **Resource Expert** (RE) is in charge of dealing with the different resources required to deploy and execute the application. This role is in charge of application component technologies, of cloud, HPC, and GPU-based computing infrastructures, or of middleware solutions for both storing data and allowing components to communicate. This role can correspond to IEEE roles in charge of Infrastructure management and Configuration management processes, given they are supposed to allocate and manage resources and configurations.
- **Quality Expert** (QE) is responsible for the quality of service both provided by the execution infrastructure and required by the executing application. Being part of the SODALITE ecosystem, he/she is in charge of offering libraries of patterns for addressing specific performance and quality problems in the SODALITE applications. This role can can correspond to IEEE roles in charge of Quality Management and Quality assurance processes because they oversee the overall quality of deployed applications and thus of the project itself.

The resources of interest for SODALITE cover the ones needed to operate applications on SODALITE. These can be specialized in:

- **Application components** are the executables the applications of interest are partitioned in. These components can be based on diverse technologies and come both as black-boxes and as complete packages, that is, the executables come with source code and with any other external artifact needed to compile, deploy, and execute them.
- **Execution platforms** provide the means to execute the different application components. They can be cloud based elements (e.g., virtual machines or containers), HPC infrastructures, or clusters of GPUs.
- **Middleware frameworks** provide the underlying glue and help both store the different data and artifacts and make the different elements communicate. Among the others, middleware frameworks include communication elements such as VPNs (Virtual Private Networks) and any other element needed to configure the interaction between the other resources or the application components.

Identified use cases reflect the main activities human actors can trigger or participate in as part of the life cycle management of IaC. Lack of room does not allow us to provide the details of each single use case. Figure 2.1 provides a high-level view of the scope of the project while Table 2.1 lists all the use-cases covered by the project. The rationale behind the different use cases is the following:

- To make the SODALITE framework usable by AOEs, it must be populated with information concerning the resources that can be exploited at runtime. This requires modelling resources (UC13) and making them available, as part of the SODALITE Domain Specific Language, to AOEs. This activity is performed by Resource Experts who are also in charge of mapping the modelled resources into specific optimization patterns (UC12). These experts can then also search for the resources they need by querying the system for already-defined resources (UC17).

ID	Use case	AOE	RE	QE
UC1	Define Application Deployment Model	✓		
UC2	Select Resources	✓		
UC3	Generate IaC code	✓		
UC4	Verify IaC	✓		
UC5	Predict and Correct Bugs	✓		
UC6	Execute Provisioning, Deployment and Configuration	✓		
UC7	Start Application	✓		
UC8	Monitor Runtime	✓		
UC9	Identify Refactoring Options	✓		
UC10	Execute Partial Redeployment	✓		
UC11	Define IaC Bugs Taxonomy			✓
UC12	Map Resources and Optimisations	✓	✓	
UC13	Model Resources		✓	
UC14	Estimate Quality Characteristics of Applications and Workload			✓
UC15	Statically Optimise Application and Deployment	✓		
UC16	Build Runtime Images	✓		
UC17	Platform Resource Discovery		✓	
UC18	Deployment governance	✓		

Table 2.1: SODALITE use cases.

- The Quality Expert defines a bug taxonomy for IaC (UC11) that helps AOEs in predicting bugs (UC5). Moreover, he/she experiments with application components and prototypes to estimate their quality characteristics (UC14).
- AOEs start their activity by defining an application deployment model (UC1). This model includes the main components of an application and any constraint or requirement on their deployment, configuration or execution. At this point they can either rely on the resources the SODALITE system would assign by default, or they could select specific resources (UC2). After this step, they are ready to trigger the automatic generation of IaC code (UC3) and its verification (UC4) as well as bug prediction and correction (UC5) and static optimization (UC15) aiming at improving application performance. Of course, these activities may lead to some reiteration in the mentioned use cases until the point in which, as part of the IaC code generation, AOEs generate the needed runtime images (UC16). Then AOEs can trigger the execution of provisioning, configuration and deployment (UC6), start the application (UC7) and start monitoring the execution (UC8) with the purpose of checking that everything is working well and, in case of problems, of identifying possible refactoring and deployment improvement options (UC9). As a result of this identification, they can go back to the modelling and IaC generation/verification/optimization phases and, at this point, trigger a partial redeployment of the system (UC10). After completing a deployment, they can also take care of governing it (UC18).

All use cases are mandatory steps for a proper usage of SODALITE, with the exception of the following ones that concern steps that can either be triggered by the actors or can be skipped. **Select Resources** since default resources can be assigned to an application if no one is selected **Predict and Correct Bugs** as the AOE may be willing to exclude this automated correction and may want to take care of bugs by himself/herself. **Monitor runtime** as, while monitoring is highly beneficial, it may introduce an overhead that users may want to exclude. Of course, excluding monitoring implies that UC9 and UC10 (refactoring and redeployment) cannot be performed. **Identify Refactoring Options**, **Execute Partial Redeployment**, and **Statically Optimizer Application and Deployment** represent the most advanced features offered by SODALITE, but the user can still use the platform without exploiting them.

2.2.1 Workflows

This section presents the main workflows supported by the SODALITE platform. They are focused on three major primary users of SODALITE - Application Ops Experts, Resource Experts, and Quality Experts - plus a secondary user, the system administrator in charge of deploying and configuring the SODALITE platform itself. In the following we present the workflows associated with these types of users and highlight the artifacts produced in these workflows and where they are located during a normal execution of the SODALITE platform.

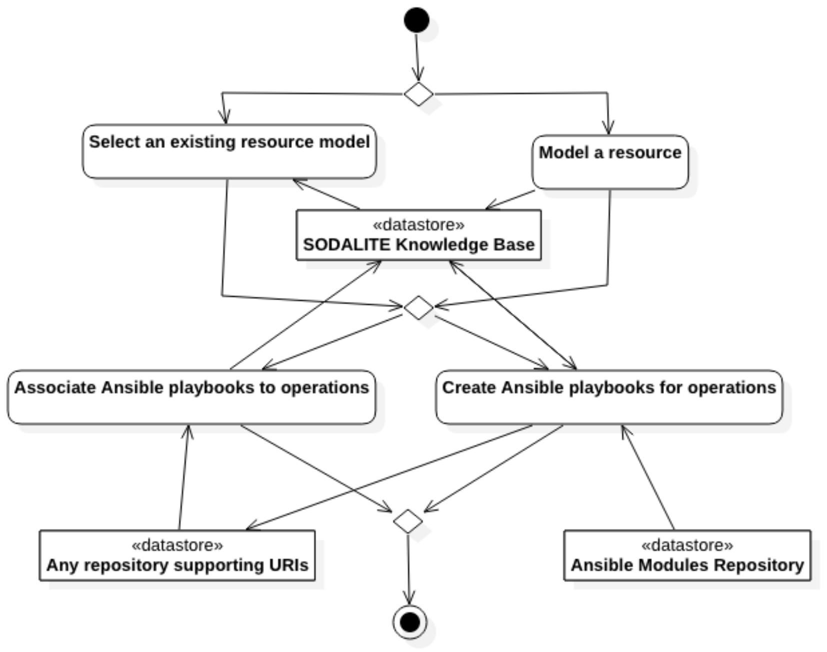

Select an existing resource model
Model a resource
«datastore»
SODALITE Knowledge Base
Associate Ansible playbooks to operations
Create Ansible playbooks for operations
«datastore»
Any repository supporting URIs
«datastore»
Ansible Modules Repository

Fig. 2.2: Workflow for the Resource Expert

Figure 2.2 presents the workflow typically followed by the Resource Expert. He/she is in charge of creating resource models and Ansible playbooks to support the execution of the corresponding operations. In the case a model of the resource under consideration is already available, for instance, because the Platform Discovery has automatically defined the resource, the Resource Expert will limit his/her work to the selection of a specific resource and to the creation or the selection, in case they are already available, of the Ansible Playbooks that implement the operations to be executed for that resource if needed.

The Resource Expert performs his/her activities by exploiting two SODALITE tools, the IDE for all modeling/editing activities and, indirectly, the Platform Discovery.

The Knowledge Base is the main data store used in this workflow. It includes the resource models and it is updated with the URL of the Ansible scripts associated to such resource models. The Ansible Modules Repository is an off-the-shelf directory offered by the Ansible community and including all available modules. The Ansible playbooks used or produced within the context of SODALITE can be made available on any datastore, including a git repository, that supports their identification through a proper URI.

Application Ops Experts are involved in two types of activities within the context of SODALITE, those concerning the design of AADMs and those concerning the execution of the corresponding TOSCA and Ansible scripts and the application runtime.

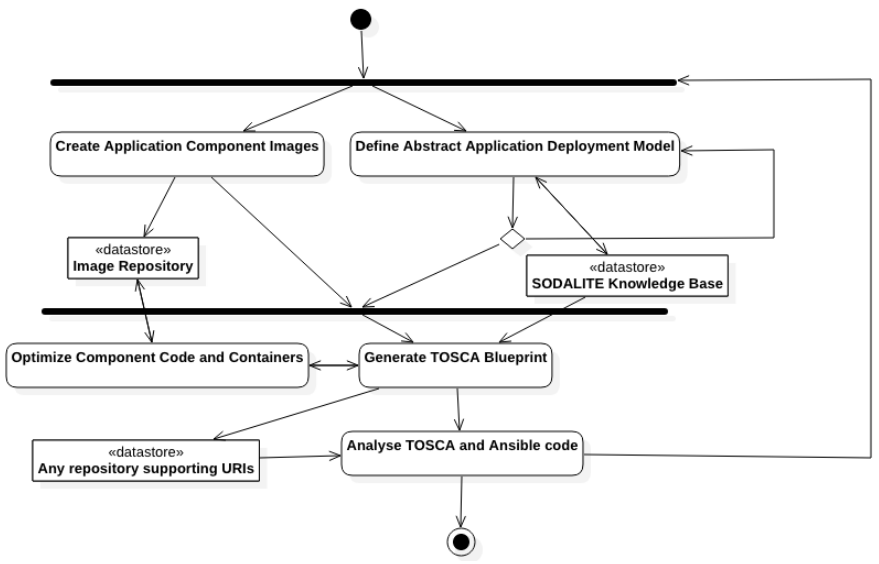

Create Application Component Images
Define Abstract Application Deployment Model
«datastore»
Image Repository
«datastore»
SODALITE Knowledge Base
Optimize Component Code and Containers
Generate TOSCA Blueprint
«datastore»
Any repository supporting URIs
Analyse TOSCA and Ansible code

Fig. 2.3: Design-time workflow for Application Operation Expert.

Figure 2.3 shows the design time activities performed by Application Ops Experts to prepare the deployment of a complex application. First they focus on preparing the images of the application components by packaging them in proper execution containers; this activity is supported by the Image Builder. In parallel, they define the Abstract Application Deployment Model (AADM) through the SODALITE IDE. This task is an iterative activity that requires interaction with the SODALITE Knowledge Base and terminates when the user is satisfied by their AADM. When images and the AADM are saved in the Image Repository and Knowledge Base, respectively, the AOE generates the TOSCA blueprint. If needed, the optimization of component code and associated containers is performed as part of this phase. The resulting TOSCA blueprint is stored in any repository, e.g. Git, that offers a URI-based mechanism for identifying its elements. Finally, the TOSCA Blueprint, together with the associated Ansible playbooks (defined by the Resource Experts) are analyzed to assess the presence of possible problems and bug smells that, if revealed, bring the AADM back into the modeling phase.

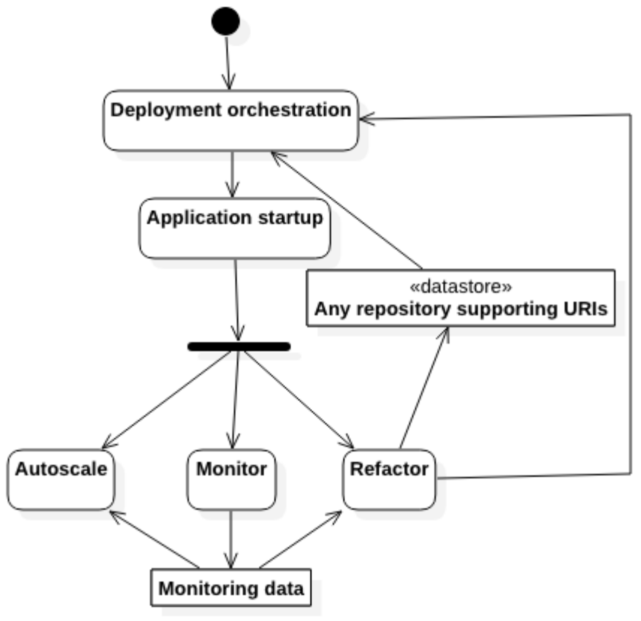


Fig. 2.4: Runtime workflow for Application Operation Expert.

Figure 2.4 describes the runtime activities that are overseen by AOEs. They are all automated, but their results can be inspected through proper dashboards. The process starts with the orchestration of a TOSCA blueprint and the associated Ansible Playbooks. The result of this step, when successful, is a complex application ready to start its execution. After execution starts, the continuous activities concerning monitoring, auto-scaling, and refactoring are performed. Refactoring can result in changes in the TOSCA blueprint that trigger a new deployment orchestration step. In this process, monitoring data are produced by the monitoring platform and exploited by the auto-scaling mechanism for short-term fine-tuning and by the refactoring for identifying longer term potential issues. TOSCA blueprints are retrieved and stored, when changed, in any suitable repository as already discussed in reference to the design time activities.

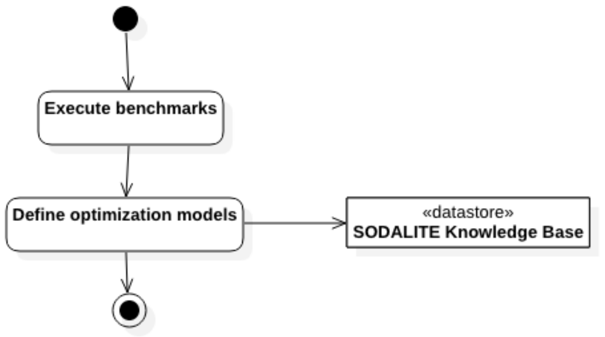


Fig. 2.5: Workflow for Quality Expert.

The Quality Expert is in charge of developing proper optimization models that constitute the inputs to the Application Optimizer (MODAK). He/she is assumed to run, externally to SODALITE, benchmarks to measure the characteristics of available resources. Based on these, he/she defines the optimization models based on the data acquired during the benchmark phase. The creation of Optimization Models is supported by the IDE, while the models are stored in the SODALITE Knowledge Base. Figure 2.5 provides an overview of the described workflow.

The last workflow associated with the usage of SODALITE concerns the activities carried out by the system administrator in charge of making the SODALITE platform available to other users. Given that this platform comprises multiple components, it is, by definition, a complex application. As such, its deployment and configuration have been automated through a proper TOSCA blueprint. This workflow is then completely automated.

2.3 SODALITE overall architecture

Figure 2.6 shows the SODALITE platform architecture. It is organized in three layers, the Modelling layer, the Infrastructure as Code layer, and the Runtime layer.

The Modelling layer includes a set of SODALITE domain ontologies, resulted by the abstract modelling of the related domains (applications, infrastructure, per-

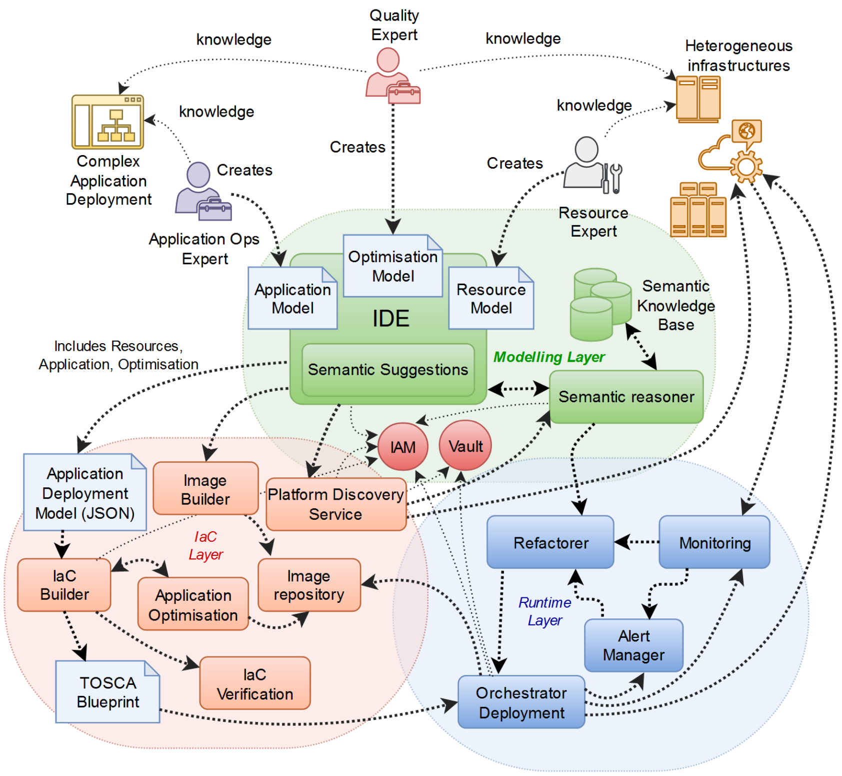


Fig. 2.6: Runtime workflow for Application Operation Expert.

formance optimization and deployment), and constituting the Semantic Knowledge Base. A dedicated middleware (Semantic Reasoner) is in charge of the population of data and the application of rule-based Semantic Reasoning. The IDE offers support for the final users for the design of deployment models.

The Infrastructure as Code Layer (IaC Layer) offers APIs and data to support the optimization, verification and validation process of both Resource Models (RM) and Abstract Application Deployment Models (AADM). Moreover, it prepares a valid and deployable TOSCA blueprint through the IaC Builder and offers the Platform Discovery Service, which supports the tasks of the Resource Expert by creating a valid TOSCA platform resource model to be stored into the SODALITE Knowledge Base.

The Runtime Layer of SODALITE is in charge of the (re)deployment of SODALITE applications into heterogeneous infrastructures, their monitoring and dynamic reconfiguration. It is composed of the following main blocks: i) An Orchestrator that receives the IaC description of the application to be deployed or re-deployed and executes it by deploying the application components on the target infrastructure.

ii) A Monitoring system which enables the users to visualize multiple metrics and the refactoring mechanism to initiate any needed recovery action. iii) A Refactoring component that identifies possibilities for increasing the efficiency of the system and proposes to the end users possible redeployment options.

SODALITE provides tools and methods to authenticate and authorize actions on API endpoints using open-source Identity Management and Secure Secret handling tools. While authorization is required - a single SODALITE endpoint can manage different infrastructures belonging to different domains. Apart from proper authentication and authorization of user actions, safe secret management across the whole deployment pipeline is also required and ensured by SODALITE.

As a basis for authorization the OAuth 2.0 protocol was chosen, which is the de-facto industry standard for authorization. As for IAM provider, SODALITE uses Keycloak - a popular and widely used open source tool which simplifies the creation of secure services with minimal coding for authentication and authorization. It allows wide customization of options exceeding the needs of SODALITE. Along with the basic authentication mechanism provided by Keycloak, SODALITE can also support such features as 2-factor authentication and seamless integration with third party identity providers like Google or GitHub.

Apart from properly authorising user's actions, the Security Pillar handles also infrastructure secrets like RSA keys, tokens, and passwords. This involves two points to be addressed: Security of data in use and security of data at rest.

The first point is mitigated by properly handling the secrets across the whole pipeline: unencrypted information is not stored, security critical parts are not logged, users are managed on virtual containers that host the SODALITE components. For addressing the second point, Hashicorp Vault was chosen, which is probably the most widely used open source tool for secret management.

2.4 A running example: Snow

This section is dedicated to an overview of the Snow example that will be used in the following chapters in all cases we want to exemplify the usage of the SODALITE features.

Snow exploits the operational value of information derived from public web media contents to support environmental decision-making in a-snow dominated context. An automatic system crawls geo-located images from heterogeneous sources at scale, checks the presence of mountains in each photo, identifies individual peaks, and extracts a snow mask from the portion of the image denoting a mountain. Two main image sources are used: we crawl touristic webcams in the Alpine area and search Flickr for geo-tagged user-generated mountain photos in the Alpine region.

Both image types carry, explicitly or implicitly, information about the location where the image is taken, but require estimating the orientation of the camera during the shot, identifying the visible mountain peaks, and filtering out images not suitable for snow analysis (e.g., due to fog, rain etc.).

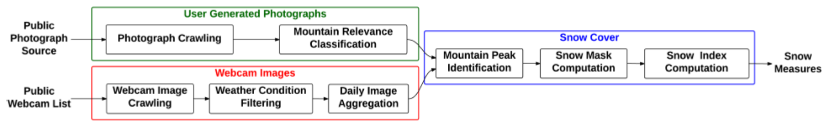


Fig. 2.7: Snow: Foreseen pipeline.

The two multimedia processing pipelines, shown in Figure 2.7, share common steps but also have differences.

Pictures from Flickr tagged with a location corresponding to a certain mountainous region do not ensure the presence of mountains. For this reason, the presence of mountains in every photograph is estimated and the non-relevant photographs are discarded. The process to classify an image first computes a fixed-dimensional feature vector, which summarizes the visual content, and then provides it to a Support Vector Machine (SVM) classifier to determine whether the image should be discarded or not. A dataset of images annotated with mountain/no mountain labels is needed to train the model.

Outdoor webcams represent a valuable source of visual content. They expose a URL which returns the most recent available image. In this case, the resulting images need to be filtered by the weather conditions, since these can significantly affect short- and long-range visibility. Additionally, snow cover changes slowly over time, so that one measurement per day is sufficient; for this reason, an aggregation of the images obtained during the day is desirable.

The distance between the shooting location and the framed mountains can be very high (tens of KMs). The photo geotag only is not sufficient for the analysis of the mountains. It is necessary to determine which portions of the image represent which mountains, identify the geographical correspondence of each pixel: estimate whether it is a terrain surface or sky, what is the corresponding geographical area, what are its GPS coordinates, altitude and distance from the observer. Once an image is geo-registered, the portion of the image that represents the mountain area can be analysed and divided into snow and non-snow areas. Mountain Image Georegistration (MIGR) is done by finding the correct overlap between the photograph and a 360-degree cylinder with a virtual mountain panorama, i.e., a synthetic image of the visible mountain skyline generated with a projection from DEM (Digital Elevation Model) data and from the camera shooting position.

A snow mask is defined as the output of a pixel-level binary classifier that, given an image and a mask M that represents the mountain area as inputs, produces a mask S that assigns each pixel of the mountain area a binary label denoting the presence of snow. Snow masks are computed using the Random Forest supervised learning classifier with spatio-temporal median smoothing of the output. To perform the supervised learning a dataset of images with an annotation at pixel level indicating if the pixel corresponds to the snow area is needed.

The pipeline produces a pixel-wise snow cover estimation from images, along with a GPS position, camera orientation, and mountain peak alignment. Thanks to the image geo-registration and orthorectification (using the associated topography data) it is possible to estimate the geographical properties of every pixel, such as its corresponding terrain area and altitude. Consequently, it is possible to compute the snow line altitude (the point above which snow and ice cover the ground) expressed in meters.

The virtual snow index for an image is defined as: $(x, y)|S(x, y) = 1vsi(x, y)$, where vsi is a virtual snow index function that transforms a pixel position into a snow relevance coefficient and can be defined as $vsi(x, y) = 1$ and $S(x, y) = 1$ indicates it will be calculated for each pixel that corresponds to the snow mask obtained in previous step.

2.5 Conclusion

In this chapter we have provided an overview of the SODALITE target users, of the workflows SODALITE supports for them and of the SODALITE architecture. We have also briefly described a case study, Snow, that will be used in the following chapters to exemplify specific aspects of our approach.

The individual components of the SODALITE toolset are presented in the following chapters. They are all available as open source software[1] and as containerized Docker images[2].

References

[1] "ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes". In: *ISO/IEC/IEEE 12207:2017(E) First edition 2017-11* (2017), pp. 1–157. DOI: `10.1109/IEEESTD.2017.8100771`.

[1] https://github.com/SODALITE-EU

[2] https://hub.docker.com/orgs/sodaliteh2020/repositories

Chapter 3
The SODALITE Model-driven Approach

Jesús Gorroñogoitia, Dragan Radolović, Zoe Vasileiou, Georgios Meditskos, Anastasios Karakostas, Stefanos Vrochidis, Michail Bachras

Abstract The specification of deployment topologies for complex applications distributed across multiple heterogeneous infrastructures is a difficult process that encompasses multiple modeling tasks, engaging several actors, including application ops experts, resource experts on the specification of the target infrastructure resources, quality experts on the application optimization, and application administrators on the deployment governance. SODALITE proposes a novel infrastructure as a code (IaC) modeling framework that provides a model driven engineering approach for the authoring of application- and infrastructure-level specifications, realizing an instantiation of an infrastructure as a code (IaC) modeling framework. This chapter introduces the SODALITE IDE and the IaC services. The IDE enables SODALITE expert roles to model (conforming to the SODALITE DSMLs) and generate IaC artefacts facilitating the app deployment. Experts are assisted in the modeling phase by the semantic knowledge inference and validation capabilities of a Knowledge Base (KB), which is populated with IaC descriptions for resources semi-automatically discovered from target heterogeneous infrastructures. The IDE leverages the SODALITE IaC services for automatic target image preparation and IaC artifacts generation upon deployment.

Jesús Gorroñogoitia
ATOS, Spain, e-mail: `jesus.gorronogoitia@atos.net`

Dragan Radolović
XLAB, Slovenia, e-mail: `dragan.radolovic@xlab.si`

Zoe Vasileiou · Georgios Meditskos · Anastasios Karakostas · Stefanos Vrochidis
Information Technologies Institute, Centre for Research and Technology Hellas, Greece, e-mail: `{zvasilei,gmeditsk,akarakos,stefanos}@iti.gr`

Michail Bachras
Politecnico di Milano, Italy, e-mail: `michail.bachras@mail.polimi.it`

3.1 Introduction

Cloud computing and infrastructure virtualization are one of the well-known leading technologies that companies are increasingly embracing.

The issues that virtualisation and expansion of infrastructure as a service based approach introduced are the scalability, inventory management, complex networking management, security policies, etc. Large scale virtual infrastructure systems are difficult to control and manage and therefore usually need high level code snippets, scripts or other advanced software artifacts to manage resources, services, deployments, upgrades, etc. The concepts of handling the code produced to manage and automate infrastructure provisioning and manage systems can be defined as Infrastructure as Code (IaC) management. The usage of IaC in the DevOps toolchain enables:

- Cost effectiveness as automation efforts reduce simple and repetitive tasks;
- Speed and efficiency as DevOps teams have tools for releasing infrastructure updates and services much faster than in manual configuration scenario;
- Immutable infrastructure as changes are applied by rebuilding resources instead of modifying the existing resources;
- Possibility of applying traceability, validation and testing to help reducing the number of errors, mitigating risks, and leading to robust setup for built in security.

However, the adoption of IaC implies also an increased complexity at it requires a deep knowledge of the target infrastructures and of multiple IaC scripting languages to support the initial application deployment and application lifeycle management, including upgrades, re-installations, reconfiguration, re-adaptation etc.

Many cloud resource management standards have been proposed for addressing those interoperability and portability issues, such as TOSCA[1], CIMI[2], and OCCI[3]. However, those standards still cause interoperability problems, since the modeling languages and the semantics differ among standards, resulting in non-reusable resources. Thus, there is a high need for modeling the application components and the resources in a standard machine-readable format.

To this end, Semantic Web technologies can promote interoperability, re-usability, and intelligent decision support to various cloud-based systems. In SODALITE, we have developed a rich conceptual meta-model that is based on the best practices on ontology engineering in order to create a formal abstraction of the application and the resources by representing the functional and non-functional requirements and interlink them with other dependencies, and relationships. This meta-model aims to create a formal abstraction of the application and of its resources by representing the functional and non-functional requirements and the dependencies among components.

[1] http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

[2] https://www.dmtf.org/standards/cloud

[3] https://occi-wg.org/

However, the ontological model is not preferred as a front-end language, as it is not user-friendly. Thus, a Domain Specific Language (DSL) has been designed to provide a programming-oriented way to describe these models by offering a lightweight language abstraction that hides the complexity of the ontologies. Accordingly, our SODALITE Integrated Development Environment (IDE) enables the application developer to create DSL-compliant models for her application and infrastructure resources. Then, the following processes for container-image preparation and IaC scripts generation result in the creation of a valid TOSCA CSAR blueprint.

The rest of this chapter is organized as follows. Section 3.2 presents the SODALITE Domain Specific Languages and the corresponding editors that are part of the IDE. Section 3.3 presents how the knowledge is represented in terms of ontologies. Section 3.4 shows the smart features offered by the IDE thanks to the ontological inference. Section 3.5 describes the procedure how a user can prepare container images of the applications. Section 3.6 presents the final steps toward the preparation of proper and ready to be executed Infrastructure as Code (IaC). Finally, Section 3.7 concludes the chapter.

3.2 The modeling approach and the IDE

The modeling of the deployment topology of complex applications across heterogeneous infrastructures engages a multidisciplinary team, consisting of several roles, namely Application Ops Experts (AOEs), Resource Experts(REs) and Quality Experts(QEs), introduced in section 2.2.1, who are involved in the specification of different deployment concerns. The SODALITE IDE supports these roles, through different editors, to address their modeling needs. Each editor is specialized in the creation of a kind of models, which are compliant to one of the SODALITE DSLs, that describe a concrete aspect relevant for specification of the application deployment topology. The SODALITE DSLs are described in the following paragraphs.

3.2.1 Abstract Application Deployment Model (AADM)

AOEs tackle the modeling of an application topology as a model instance of the AADM DSL. An AADM is a *topology*, that is, a connected graph of application components that declares the relationships among them and the requirements to the infrastructure resources they need. They can be resolved, either at design or deployment time, through the concrete selection of the suitable resources from a target infrastructure. As the ultimate purpose of this AADM DSL is to simplify the modeling of the application deployment topology, and its conversion into a TOSCA[12] blueprint by the IaC layer (see section 3.6), this DSL borrows modeling concepts for TOSCA topologies, such as node templates and policy definitions. Moreover, leveraging TOSCA in SODALITE DSLs largely simplifies their adoption

```
snow-weather-condition-filter:
  type: docker/sodalite.nodes.DockerizedComponent
  optimization: ai_training.tensor_flow
  properties:
    docker_network_name: get_property:
                             entity: SELF
                             property: snow/snow-docker-network.name
                             req_cap: snow/snow-weather-condition-filter.network
    volumes: [
      "/tmp/conf/config.json:/SnowWatch-SODALITE/config.json",
      "snow_volume_shots:/SnowWatch-SODALITE/data/shots",
      //"/tmp/masks: /SnowWatch-SODALITE/data/masks",
      "snow_volume_daily_shots:/SnowWatch-SODALITE/data/daily_shots"]
    image_name: "snow-weather-condition-filter:latest"
    alias: "snow-weather-condition-filter"
    restart_policy: "always"
    command: [
      "{{ ansible_date_time.date }}",
      "34 40 50 62 608 666 822 852 943 1307 6666"]
    registry_url: get_input: docker-registry-url
  requirements:
    host:
      node: snow/snow-docker-host
    dependency:
      node: snow/snow-mysql
    dependency:
      node: snow/snow-docker-volume-shots
    dependency:
      node: snow/snow-docker-volume-masks
    dependency:
      node: snow/snow-docker-volume-daily-shots
    dependency:
      node: snow/snow-configuration-demo
    network:
      node: snow/snow-docker-network
    registry:
      node: snow/snow-docker-registry
```

List. 3.1: Snow component described in an AADM model

among end-users already familiarized with this de-facto Cloud deployment language. Listing 3.1 shows an excerpt of an AADM for the Snow UC application. In there a `snow-weather-condition-filter` component is declared as an instantiation of the abstract type `DockerizedComponent`. It overrides some of the properties inherited from that type, in order to further specialize the component. Moreover, it declares an explicit dependency on another component, `snow-mysql`, and expresses some requirements such us the `registry` and `network` to be used, as well as the infrastructure resource where to `host` the component. Finally, it makes use of a specific optimization that is described in the next paragraph. Despite these requirements have been specified at design time in this example, the abstract nature of the AADM permits the resolution of the mandatory requirements in the topology at deployment time, assisted by the inference and reasoning capabilities of the *SODALITE ontology* (see section 3.4).

3.2.2 Optimisation model(OM)

QEs tackle the design of deployment optimizations as instances of the Optimization Model (OM) DSL (see section 4.4.2). Such instances can be bound to components declared in AADMs as in Listing 3.1. Listing 3.2 shows an example of optimization model for components that use the TensorFlow AI training library. At a top level, QEs build the optimisation model by specifying whether to enable optimal build for the target (*enable_opt_build*) and auto-tuning (*enable_autotuning*) and the application type (*app_type*). Then, QEs add section-specific options that the optimiser can use to select an optimised container and provide target-specific runtime options.

In the example, *opt_build* node specifies the node architecture of the target and *autotuning* specifies the tuner tool and its input. The application-specific options are listed in the *app_type-ai_training* node, with sub-node *config* listing the application configuration and *data* listing the data settings. The *ai_framework-tensorflow* shows the AI framework-specific options for optimisation. Based on the optimisation model, the application optimiser maps the optimal application parameters and deployment settings to that of the target hardware and selects an optimised container.

Optimization models are shareable and can be linked to any application component node declared in an AADM topology by setting the `optimization` parameter. As an example, the `snow-weather-condition-filter` component node, described in Listing 3.1, can be associated with the optimisation model (for a TensorFlow AI trainng application) in Listing 3.2. When a node in the AADM is associated to an optimization model, the semantic reasoner validates the association by cross checking entries like the application type and target architecture. At deployment time, the AADM and the associated optimisation model are passed on to the application optimiser that actuates the definition.

3.2.3 Resource Model(RM)

REs tackle the modeling of infrastructure resources as model instances of the Resource Model (RM) DSL. This DSL defines classes for describing new resource types, their capabilities, requirements, relationships and interfaces. Alike AADM DSL, RM DSL borrows concepts from TOSCA for the specification of resource types, their relationships, associated policies and other aspects. Listing 3.3 shows an example of new resource type, the `ConfigurationDemo`, a specialization of a `SoftwareComponent`. The RE declares a new property, `mysql-db-pass` and expresses the requirement `remote-server` restricted to `VM` resources that offer a `Compute` capability, being this requirement bound through the `DependsOn` relationship. Moreover, the RE specialises the `create` operation of the `Standard` interface, by declaring new inputs, namely the `remote-server` and the `mysql-db-pass`, and by providing an Ansible implementation for this operation. This can be done either directly writing or reusing an Ansible playbook or by exploiting the Ansible DSL and associated editor as described in Section 3.2.4.

```
optimization ai_training.tensor_flow:
  enable_opt_build: true
  enable_autotuning: true
  app_type: ai_training
  opt_build:
    cpu_type: arm
    acc_type: nvidia
  autotuning:
    tuner: myscript
    input: "/home/yosu/Projects/Sodalite/Git/ide
    dsl/org.sodalite.dsl.examples/optimization_v1/autotunig.at"
  app_type-ai_training:
    config:
      ai_framework: tensorflow
      type: translation
      distributed_training: true
      layers: 6
      parameters: 872684236
    data:
      location: "/some/data"
      basedata: imagenet
      size: 67
      count: 4389
      etl:
        prefetch: 100
        cache: 100
    ai_framework-tensorflow:
      version: "2.1"
      xla: false
```

List. 3.2: Optimization model example: AI training based on Tensorflow.

AADM, OM and RM are interlinked. The AADM imports both the OM and the RM so that application components in an AADM model can refer to optimization models and can be instantiated as component instances of the types defined within RM models. This split of metamodels for the different modeling concerns permits these three roles to focus only on one of those concerns by using a single metamodel.

3.2.4 Ansible Model(AM)

As mentioned in the previous section, the Resource Expert may decide to implement operations defined in a Resource Model as Ansible Models that are then translated into executable Ansible playbooks. The IDE, in fact, implements an Ansible DSL and offers to the users suggestion and verification facilities that allow them to create Ansible Models in an easy way.

For each specified operation within a Resource model, the user can select the option to generate the abstract Ansible model and, then, the corresponding Ansible script for further development. As a result, as shown in Figure 3.1, a folder is created for each defined node type, together with a subfolder with the name of each of its TOSCA interfaces. Finally, for each TOSCA operation, a `.ans` file, containing the

```
sodalite.nodes.ConfigurationDemo:
  derived_from: tosca.nodes.SoftwareComponent
  properties:
    mysql_db_pass:
      type: string
      description: "MYSQL database password"
      required: true
  requirements:
    remote_server:
      capability: tosca.capabilities.Compute
      node: openstack/sodalite.nodes.OpenStack.VM
      relationship: tosca.relationships.DependsOn
  interfaces:
    Standard:
      type: tosca.interfaces.node.lifecycle.Standard
      operations:
        create:
          inputs:
            remote_server:
              type: string
              default: get_attribute:
                         attribute: tosca.nodes.Compute.public_address
                         entity: SELF
                         req_cap:sodalite.nodes.ConfigurationDemo.remote_server
            mysql_db_pass:
              type: string
              default: get_property:
                         property: sodalite.nodes.ConfigurationDemo.mysql_db_pass
                         entity: SELF
            ansible_user:
              type: string
              default: get_attribute:
                         entity: SELF
                         attribute: sodalite.nodes.OpenStack.VM.username
                         req_cap: tosca.nodes.SoftwareComponent.host
          implementation:
            primary: "configure_demo.yml"
            dependencies: ["config.json.tmpl"]
```

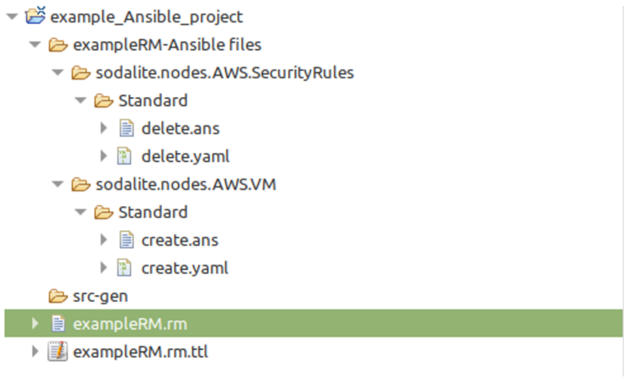
List. 3.3: Example of definition of a new resource type for Snow

abstract Ansible model is created. After this model is completed, the user can then generate the corresponding `.yaml` file, containing the Ansible concrete code.

During the Ansible Model creation task, the user is supported in all steps. For instance, the IDE suggests the names of the operations to be defined, based on the RM definition and highlights syntax errors. Moreover, it gives to the user the possibility to use as variables in the Ansible Model the inputs that have been defined in the Resource Model (see Figure 3.2).

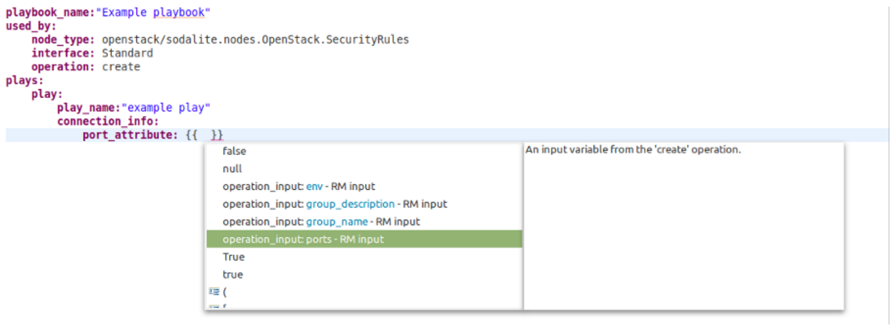
An interesting feature that enables reuse of preexisting code, is the possibility for the user to import and use in an Ansible Model the modules available within the Ansible Galaxy repository [4]. The IDE, in fact, first, enables the user in browsing through the Galaxy modules. Then, it provides content assistance for each Ansible module parameter, emphasizing the required ones. It also informs the user about inserting values for each parameter by displaying the value type that each parameter

[4] https://galaxy.ansible.com/



```
example_Ansible_project
  exampleRM-Ansible files
    sodalite.nodes.AWS.SecurityRules
      Standard
        delete.ans
        delete.yaml
    sodalite.nodes.AWS.VM
      Standard
        create.ans
        create.yaml
  src-gen
  exampleRM.rm
  exampleRM.rm.ttl
```

Fig. 3.1: Ansible Model folder structure.



```
playbook_name:"Example playbook"
used_by:
    node_type: openstack/sodalite.nodes.OpenStack.SecurityRules
    interface: Standard
    operation: create
plays:
    play:
        play_name:"example play"
        connection_info:
            port_attribute: {{  }}
```

```
false
null
operation_input: env - RM input
operation_input: group_description - RM input
operation_input: group_name - RM input
operation_input: ports - RM input
True
true
(
```

An input variable from the 'create' operation.

Fig. 3.2: Ansible Model editing support.

expects and presenting the acceptable values and the official description that helps the user understand its purpose. Moreover, the Ansible editor's content assistance offers suggestions for *standalone roles* available on Ansible Galaxy and *Ansible roles* included in Ansible collections.

The validation mechanisms are a significant addition to the Ansible editor because they allow the end-user to identify mistakes in the Ansible model directly and avoid the repetitive execution of the Ansible script before fixing all the defined errors.

3.2.5 Monitoring Alerting Rule Model

AOEs can also describe the rules that determine the triggering of alerts when the monitoring of their deployed applications reveal some anomalous situations. When the rule condition applies, the associated alert is dispatched by the SODALITE `Monitoring` (see section 5.4) and captured by `Refactoring` (see section 5.5), which eventually computes corrective actions on the affected deployed application.

```
group: alert_rules
  alert: OutOfDiskSpace
    expr: ((((node_filesystem_size_bytes {} * 100) / node_filesystem_size_bytes {}) <
    ↪  25) and on(instance, device, mountpoint) node_filesystem_readonly{}) == 0
    for: '2m'
    labels:
      severity: 'warning'
    annotations:
      summary: 'Disk space running low (instance {{ $labels.instance }})'
      description: "'Available disk space is low (< 25% left)\n  VALUE = {{ $value }}\n
      ↪  LABELS: {{ $labels }}'"

  alert: CoresHighCPU
    expr: ((count by (instance) (((1 - rate (node_cpu_seconds_total{mode="idle"}[1m])) >
    ↪  0.75)) / count by (instance) (node_cpu_seconds_total{mode="idle"})) or on()
    ↪  vector(0)) > 0.5
    for: '5m'
    labels:
      severity: "warning"
    annotations:
    summary: 'More than 50% of cores are above 75% load (instance {{ $labels.instance
    ↪  }})'
    description: "'Percentage of cores above 75% load is > 50%\n  VALUE = {{ $value }}\n
    ↪  LABELS: {{ $labels }}'"

  alert: LowReceivedNetworkTraffic
    expr: sum by (instance) (increase (node_network_receive_bytes_total{}[1h])) <
    ↪  30000000
    for: '5m'
    labels:
      severity: 'warning'
    annotations:
      summary: 'Low incoming network traffic (instance {{ $labels.instance }})'
      description: "'Received network traffic in the last hour is < 30MB\n  VALUE = {{
      ↪  $value }}\n  LABELS: {{ $labels }}'"
```

List. 3.4: Example of definition of new monitoring alerting rules

Listing 3.4 shows examples of alerting rules for Snow UC. These alerting rules are based on Prometheus PromQL[5] language. 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 endpoint that captures it, that is, the `Refactoring`.

[5] https://prometheus.io/docs/prometheus/latest/querying/basics/

3.2.6 The IDE

SODALITE DSLs modeling is supported by the SODALITE IDE[6], which is based on Eclipse[7]. The IDE offers textual and graphical editors for creating AADM models, and textual editors for OM and RM and Ansible models. The DSL grammars, parsers, serializers and textual editors have been developed by using XText[8]. The graphical viewers and editors, and the form-based property views have been implemented by using Sirius[9]. The synchronicity between textual and graphical editors has been set up by using the XText-Sirius Integration[10]. Textual editors for AADM, OM, RMs and Ansible models are intended for skilled modelers who require fast and high modeling productivity. These editors provide context-aware intelligent content-assistance to guide modelers through the syntax and content of the SODALITE DSLs, suggesting possible elements to be inserted into the model at the point of typing (see section 3.4). A graphical editor (see Figure 3.3) and its accompanying form-based editors for AADM are also available, and support similar intelligent content-assistance. They are intended for modelers who prefer the visual modeling based on a canvas where to drag and drop entities selected from a palette, but also for textual modelers that appreciate a visual representation of the AADM model. They are synchronized with the textual editor so that changes in the textual model are immediately reflected (upon saving) in the graphical and the form-based property views, and vice versa. The AADM visual representation resembles the one adopted in TOSCA documentation (which is not a standardized one).

The IDE only requires from AOEs and RMs the provision of mandatory information they own about their application topology and/or required infrastructure resources, and relies on the `Knowledge Base (KB)` to complete the model either at the design or deployment time. The IDE also exploits the reasoning engine to assist modelers during the authoring of AADM or RM models, by suggesting them suitable choices to fulfil certain model entities, including the overriding of properties inherited from super types, the resolution of mandatory requirements, the proper selection of component types, etc (see section 3.4). AADM and RM models are stored into the SODALITE KB to be reused and shared with other modelers.

The IDE textual editors conduct syntactic validation (i.e. to ensure DSL compliance) during the modeling phase. Semantic validation is conducted by the reasoning engine and by the optimization sub-system. All detected errors and warnings, as well as possible fixes, are presented to the user in the IDE and associated to the affected component.

Besides supporting DSL modeling, the IDE offers an user interface for most of the SODALITE user-driven processes, namely:

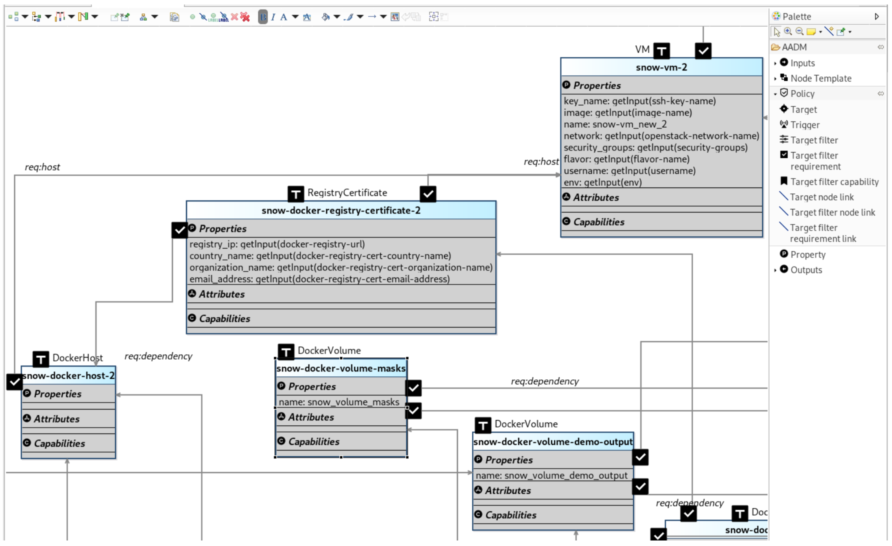
[6] https://github.com/SODALITE-EU/ide

[7] https://www.eclipse.org

[8] http://www.eclipse.org/Xtext/

[9] https://www.eclipse.org/sirius/

[10] https://github.com/altran-mde/xtext-sirius-integration



snow-vm-2

key_name: getInput(ssh-key-name)
image: getInput(image-name)
name: snow-vm_new_2
network: getInput(openstack-network-name)
security_groups: getInput(security-groups)
flavor: getInput(flavor-name)
username: getInput(username)
env: getInput(env)

snow-docker-registry-certificate-2

registry_ip: getInput(docker-registry-url)
country_name: getInput(docker-registry-cert-country-name)
organization_name: getInput(docker-registry-cert-organization-name)
email_address: getInput(docker-registry-cert-email-address)

snow-docker-host-2

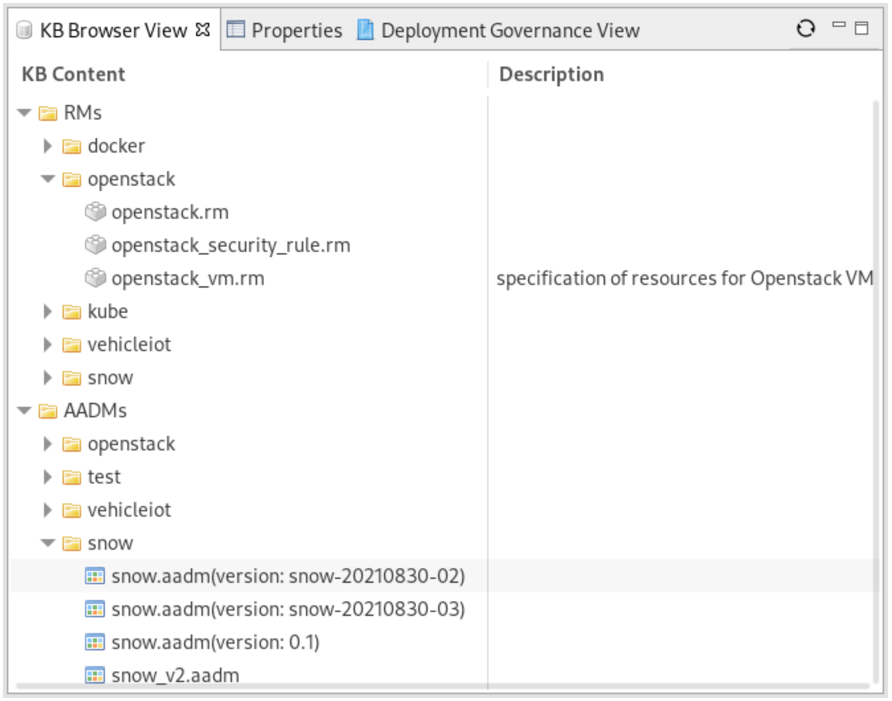
snow-docker-volume-masks

name: snow_volume_masks

snow-docker-volume-demo-output

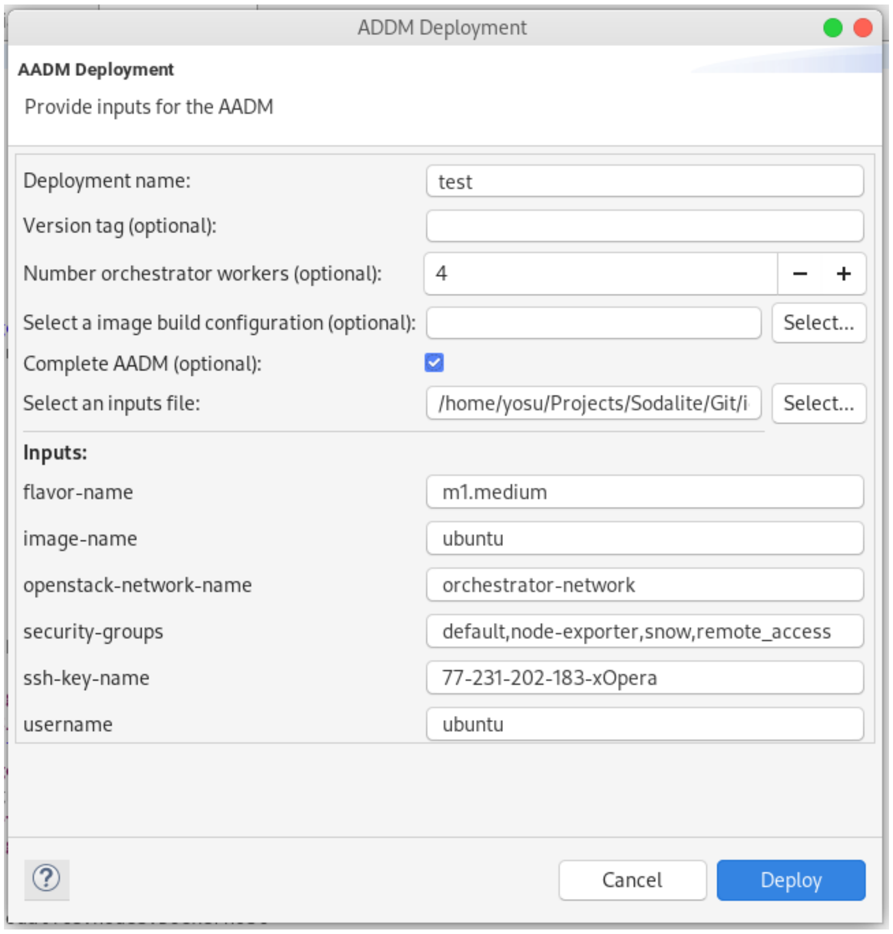
name: snow_volume_demo_output

Fig. 3.3: Excerpt of Snow AADM graphical view


KB Browser View | Properties | Deployment Governance View

KB Content	Description
RMs	
docker	
openstack	
openstack.rm	
openstack_security_rule.rm	
openstack_vm.rm	specification of resources for Openstack VM
kube	
vehicleiot	
snow	
AADMs	
openstack	
test	
vehicleiot	
snow	
snow.aadm(version: snow-20210830-02)	
snow.aadm(version: snow-20210830-03)	
snow.aadm(version: 0.1)	
snow_v2.aadm	

Fig. 3.4: IDE KB View


ADDM Deployment

AADM Deployment

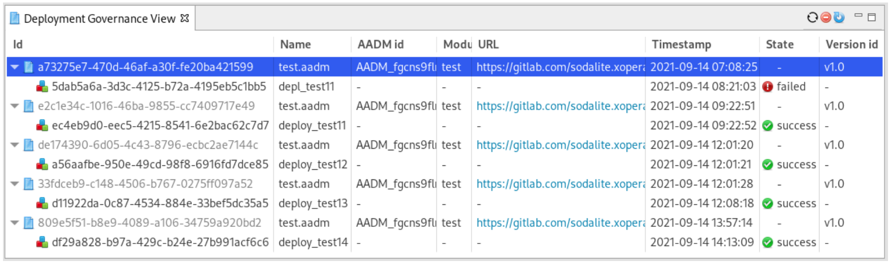
Provide inputs for the AADM

Deployment name:	test
Version tag (optional):	
Number orchestrator workers (optional):	4
Select a image build configuration (optional):	
Complete AADM (optional):	✓
Select an inputs file:	/home/yosu/Projects/Sodalite/Git/i

Inputs:

flavor-name	m1.medium
image-name	ubuntu
openstack-network-name	orchestrator-network
security-groups	default,node-exporter,snow,remote_access
ssh-key-name	77-231-202-183-xOpera
username	ubuntu

Fig. 3.5: Deployment Wizard


Deployment Governance View

Id	Name	AADM id	Modu	URL	Timestamp	State	Version id
a73275e7-470d-46af-a30f-fe20ba421599	test.aadm	AADM_fgcns9fl	test	https://gitlab.com/sodalite.xopera	2021-09-14 07:08:25	-	v1.0
5dab5a6a-3d3c-4125-b72a-4195eb5c1bb5	depl_test11	-	-	-	2021-09-14 08:21:03	failed	-
e2c1e34c-1016-46ba-9855-cc7409717e49	test.aadm	AADM_fgcns9fl	test	https://gitlab.com/sodalite.xopera	2021-09-14 09:22:51	-	v1.0
ec4eb9d0-eec5-4215-8541-6e2bac62c7d7	deploy_test11	-	-	-	2021-09-14 09:22:52	✓ success	-
de174390-6d05-4c43-8796-ecbc2ae7144c	test.aadm	AADM_fgcns9fl	test	https://gitlab.com/sodalite.xopera	2021-09-14 12:01:20	-	v1.0
a56aafbe-950e-49cd-98f8-6916fd7dce85	deploy_test12	-	-	-	2021-09-14 12:01:21	✓ success	-
33fdceb9-c148-4506-b767-0275ff097a52	test.aadm	AADM_fgcns9fl	test	https://gitlab.com/sodalite.xopera	2021-09-14 12:01:28	-	v1.0
d11922da-0c87-4534-884e-33bef5dc35a5	deploy_test13	-	-	-	2021-09-14 12:08:18	✓ success	-
809e5f51-b8e9-4089-a106-34759a920bd2	test.aadm	AADM_fgcns9fl	test	https://gitlab.com/sodalite.xopera	2021-09-14 13:57:14	-	v1.0
df29a828-b97a-429c-b24e-27b991acf6c6	deploy_test14	-	-	-	2021-09-14 14:13:09	✓ success	-

Fig. 3.6: Deployment Governance View

- *Model Management in KB.* AADMs and RMs can be saved (wizard assisted) into a remote shareable `Knowledge Base` for further reuse by other users. In case of

AADMS, versioning is available to distinguish between different versions of the same application deployment. Moreover, the IDE offers a `KB Browser View` to browse all those available models in the `Knowledge Base` for which the user has read permissions. Selected models can be retrieved into the user's local Eclipse workspace for further edition, or deleted (if user's has write permissions) when not needed (see Figure 3.4).

- *AADM Deployment.* AoEs can select AADMs in the IDE `project explorer view` for deployment. A popup wizard guides the user through the deployment process, requesting the AADM inputs, as well as the `orchestrator` configuration (see section 5.2), such as the number of workers for parallel deployment (see Figure 3.5). Optionally, AoEs can request the wizard to complete the AADM by resolving unfulfilled component requirements, leveraging the `Knowledge Base` inference capabilities (`Complete AADM` check box in wizard). The deployment process continues in the IDE background notifying the user about its status.
- *Deployment Governance.* AoEs can browse all their owned deployments in the IDE `deployment governance view` (see Figure 3.6), which shows them grouped by application (i.e. blueprint in the `orchestrator` terminology). Details of each deployment are shown, including external links to associated monitoring dashboards, which are accessible in Web browsers. Failed deployments can be resumed for initial state or from the first failing node. Selected deployment and empty blueprints can be removed when unneeded.
- *Image building.* Images for VM creation, required by applications to be deployed, can be created by the `image builder` and registered into the `image registry` from the IDE by providing a image file descriptor (see section 3.5).
- *Resource discovery.* Target infrastructure resources can be automatically discovered and their associated RMs created by the `Platform Discovery Service` (see 2.2.1) and stored into the `Knowledge Base`. This process can be triggered from the IDE for selected infrastructure types.

3.3 Knowledge Representation and Ontologies

Semantic Web technologies, and particularly, ontologies represent the domain knowledge in a formal and abstracted manner that fosters advanced reasoning. Therefore, in SODALITE, ontologies were adopted for capturing the TOSCA meta-model that contain complex semantics such as subsumption hierarchies and multi-role concepts. Through the use of ontologies, SODALITE mainly aims to (i) follow a common, extensible, and formal standardised model in order to describe cloud-related concepts (ii) capture both the structural and semantic relationships in an unambiguous manner by managing the information in the form of Knowledge Graphs (iii) create an interpretation and validation layer, for example, for inferring validation errors and smells[2] by following the semantics of TOSCA [3], and by reusing existing rule languages and logic-based frameworks.

3.3.1 Background in Ontologies

An ontology is a formal, explicit specification of a shared conceptualisation [10]. The world "ontology" is used with different meanings in different communities [14]. Namely, both the philosophy and computational field share in common the attempt to represent the knowledge formally as a set of concepts along with the relations among them. They have become such popular mainly due to the promise of the semantic interoperability and common understanding among different parties [5]. Their expressivity and level of formalisation depend on the knowledge representation language used. The Semantic Web is an extension of the current Web by annotating the resources with meta-information for establishing a common representation across heterogeneous sources. Ontologies play a key role within the Semantic Web by enabling the weaved knowledge to be interpreted into a machine-understandable format. In pursuit of that objective, the Web Ontology Language (OWL) is a key Semantic Web Ontology Language that is part of the W3C standard and was designed to represent complex knowledge.

The OWL language has been heavily influenced by the Description Logics. Description Logics (DLs) is a popular knowledge representation formalism. A DL knowledge base consists of two different types of statements, an ABox (assertional knowledge) and a TBox (terminological knowledge) [4]. The TBox describes the structure of the data stating general properties of concepts and roles, in other words the ground truth, while the ABox contains the instances of the concepts defined in the TBox. For instance, the TBox axiom Compute $\sqsubseteq$ Root asserts that all objects that belong to the concept Compute, are members of the concept Root too. The Abox are the real world entities, the instances of the TBox classes. For example, Compute(vm) and hasProperty(vm,name) express that vm has a property, which is described by the name instance. For representing the data of the Semantic Web, a set of web-based knowledge representation languages has been developed. Some of those languages are: RDF (Resource Description Format), RDF(S) (RDF Schema), and OWL [1].

- **RDF:** The RDF is a language for describing resources on the Web, was originally released as a W3C recommendation in 1999, and updated in 2004 and in 2014. The RDF data model is based on graphs, as opposed to tuples that underlie relational data models. In RDF, a data graph is constructed by the union of a number of three part assertions, called triples. A triple is composed of three parts: a *subject*, a *predicate*, and an *object*.
- **RDF(S):** RDF provides the basis for the Semantic Web, but it is limited in expressing the definition of the resources and their relations. RDF(S) is a standard that released along with the second generation of RDF in 2004 (and updated in 2014) and based on RDF. It defines classes and properties extending the base RDF vocabulary and provides support for more expressive knowledge modelling semantics. Using the RDF(S) vocabulary, it is possible to model lightweight ontologies, consisting of concepts, relations and their hierarchies.
- **OWL**: The OWL was developed simultaneously with RDFS to provide more high-level expressiveness. It is a knowledge representation language widely used

within the Semantic Web community for creating ontologies. OWL has three expressive sub-languages: OWL Lite, OWL DL, and OWL Full. OWL Full is the most expressive , but due to the high degree of expressiveness, it is undecidable. As a result, OWL DL is primarily used as a more expressive dialect. However, due to the tree model property [16], OWL can model only domains where objects are connected in a tree-like manner. This constraint is quite restrictive for many real-world application, therefore, the W3C working group produced the OWL2 [9]. OWL2 is a revised extension of OWL, commonly referred as OWL1. Some of the most prominent extensions are constructs for specifying cardinality, and value restrictions, and complex property inclusion axioms (property chains). OWL2 is divided into three different profiles, OWL2 EL, OWL2 QL, and OWL2 RL.

The SODALITE capitalizes on the most expressive features of OWL2:

- **Meta-modeling**: It is the practice of treating individuals as classes. One feature of meta-modeling is that it must be possible to assign properties to classes in the model. This way it is possible to assert the membership of classes in meta-classes and interconnect them via meta-roles [8]. The direct semantics of OWL2 do not allow for meta-modeling, but OWL2-DL supports it by the use of punning [11]. Punning allows for using the same identifier e.g. for an individual and a class. The main motivations for meta-modelling are that a model often needs to play more than one role in an application, reusability is promoted, and complex situations can be modeled more effortlessly.
- **Ontology Design Patterns:** An Ontology Design Pattern (ODP) models a recurrent ontology design problem. ODPs can be viewed as small, modular and reusable, and templates based on these patterns or other regularities in the ontology [15]. They can be also viewed as as a way of bottom-up pattern finding that is then reused across the ontology and offerred a a 'best practices' design solution. For achieving better degree of knowledge sharing, reuse and interoperability, SODALITE's conceptual model reuses the DnS ontology [6] pattern of DOLCE+DnS Ultralite (DUL) ontology [7].

3.3.2 SODALITE Conceptual Model

In SODALITE we propose an ontology-based framework [13] for capturing and interlinking TOSCA-based descriptions of cloud applications and resources. In this section, it will be described the conceptual model and the modeling decisions for implementing the ontology-based semantic abstraction layer of SODALITE.

The SODALITE semantic models include the SODALITE meta-model and the domain ontology: (i) The SODALITE meta-model is the formal ontology pattern that is used in all the different levels of abstraction (ii) The domain ontology, which is Tier 0, contains the TOSCA normative types and provides the vocabulary that will be used in the other two modelling layers (tiers), namely Tier 1 (resources) and Tier 2 (pattern instantiations).

3.3.2.1 Tiers

SODALITE follows a modular, 3-tier approach to capture the knowledge. In all the tiers, the SODALITE ODP is used in order to foster a unified representation paradigm for enabling a unified and complete model that promotes interoperability, extensibility and smooth knowledge management. The tiers are described in detail as follows:

- **Tier 0:** This is the static tier of the model. It contains the TOSCA vocabulary, namely the TOSCA meta-model in the conceptual SODALITE ODP. For example, all the TOSCA built-in types are modelled in this tier with all their relationships and associations.
- **Tier 1:** This tier involves all the custom resource types created by Resource Experts. Those custom types extend the TOSCA built-in types.
- **Tier 2:** These are the instances composing the Abstract Application Deployment Model (AADM), named also as "templates", and are reusable combinations of Tier 0 and Tier 1 types. Application Ops Experts create the AADM using the DSL in the SODALITE IDE (see section 3.2).

3.3.3 Descriptions and Situations Pattern (DnS)

For better degree of knowledge sharing and reuse, the SODALITE ODP is a specialised instantiation of the Descriptions and Situations (DnS) ontology pattern that is part of DOLCE+DnS Ultralite (DUL). The Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) focuses on cognitive issues and aims at capturing ontological categories underlying natural language and human common sense. The DOLCE + DnS Ultralite is a light version which has been simplified and improved. In SODALITE, we adopted the DnS design pattern which is part of the aforementioned light version of DOLCE ontology. The purpose of DOLCE + DnS Ultralite ontology is to provide the basis for easier interoperability among middle and lower level ontologies.

The DnS pattern is presented in Figure 3.8. This pattern captures the notion of "Situation" out of the state of affairs, with their interpretation being provided by a "Description".

- **Situation:** a set of domain entities that are involved in a specific pattern instantiation
- **Description:** represents the descriptive context of a situation that defines the concepts that classify the domain entities of a specific pattern instantiation. In such a way, views are created on the situtations.
- **Concepts and parameters:** Classify domain entities describing how they are interpreted in a particular situation. Each concept might have one or more parameters for describing additional descriptive context.

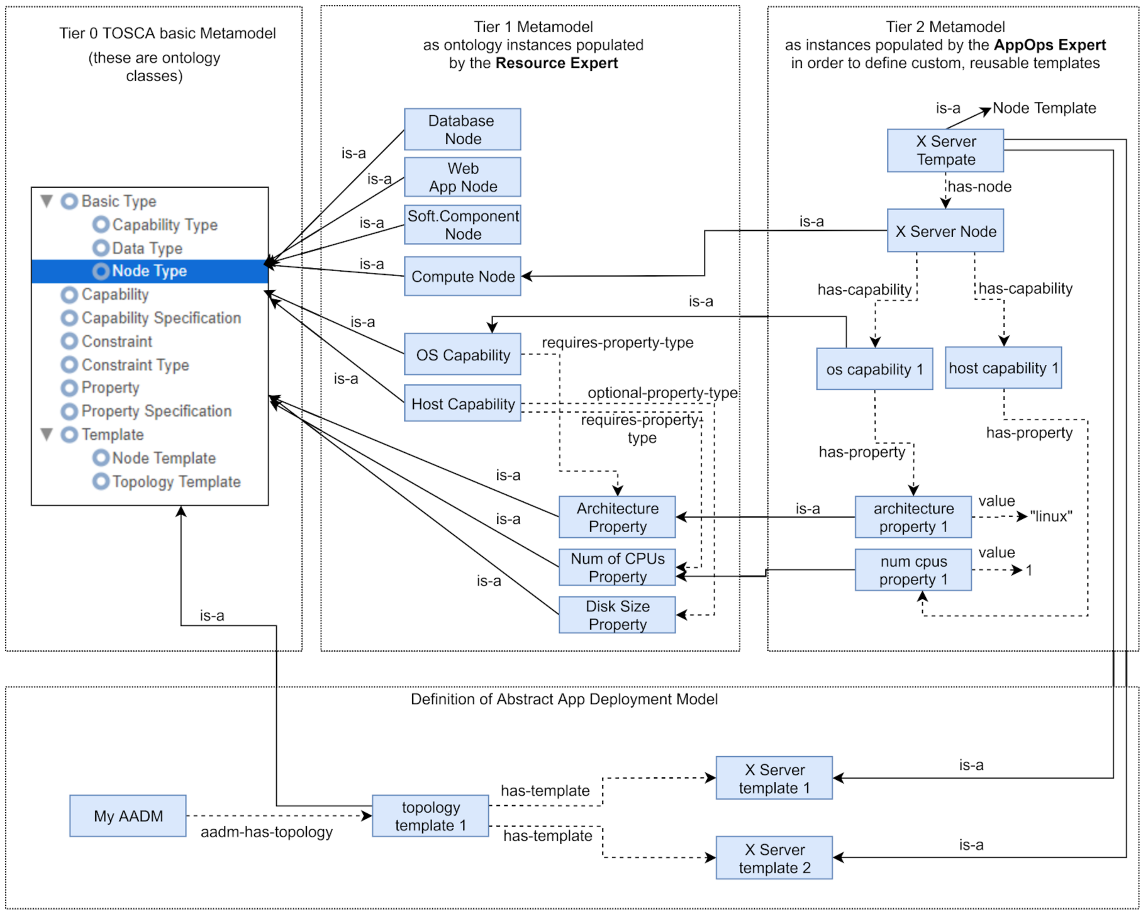


Fig. 3.7: Overview of the SODALITE modelling layer

3.3.4 SODALITE Meta-Model

The SODALITE meta-model, which is depicted in Figure 3.9, extends the core DnS pattern by specializing the core DUL concepts in order to model TOSCA definitions.

- **SodaliteSituation:** It represents a situation such as a node template or type (SodaliteSituation ⊑ dul:Situation).
- **SodaliteDescription:** Each situation is related with (soda:hasContext) a description (SodaliteDescription ⊑ dul:Description) that contains the attributes properties, interfaces, requirements e.t.c.
- **SodaliteConcept and SodaliteParameters:** Each description has a specification (soda:specification ⊑ dul:defines) that involves one or more zero concepts such as attributes, properties, requirements e.t.c) and each Concept has one or more

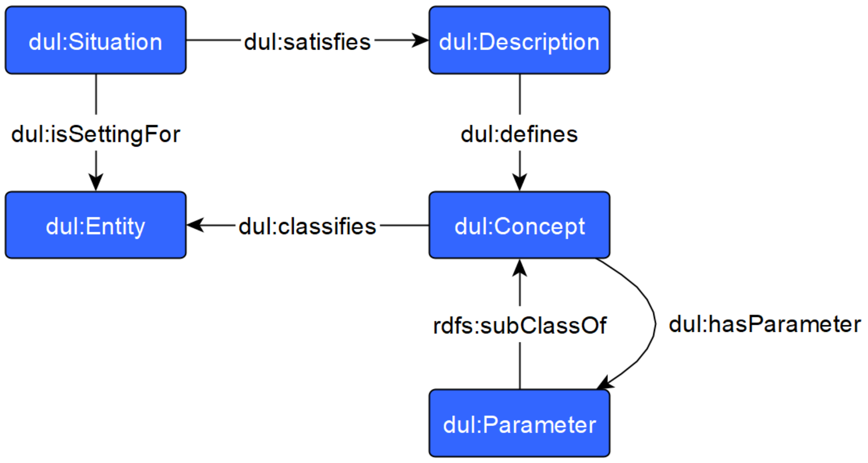


Fig. 3.8: Core DnS pattern in DuL

parameters (soda:hasParameter ⊑ dul:hasParameter) for describing more nested content. Each SodaliteConcept classifies one SodaliteEntity (soda:SodaliteEntity ⊑ the dul:Entity).

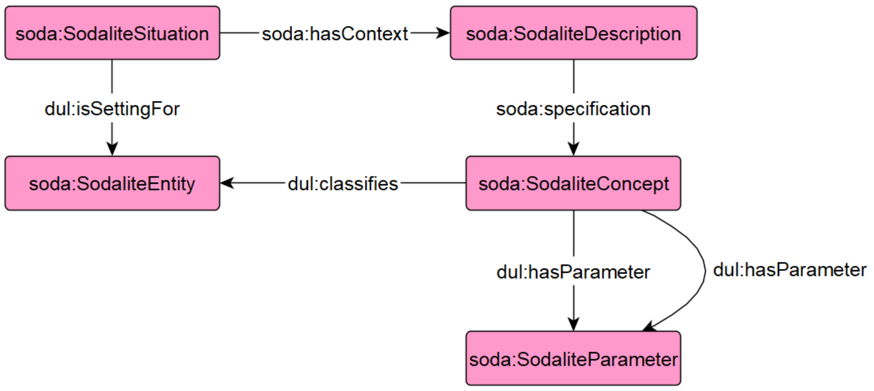
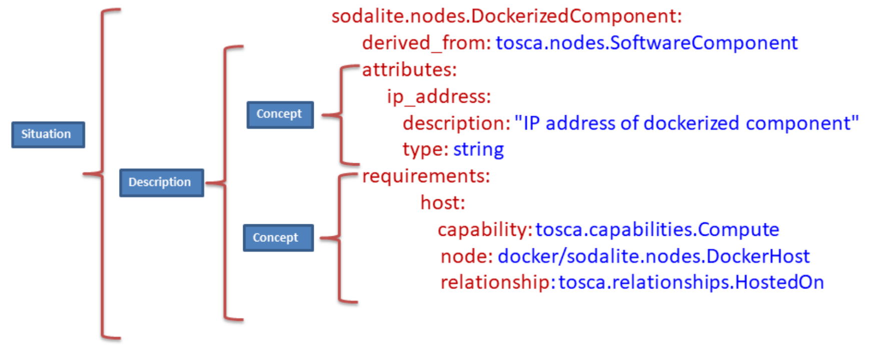


Fig. 3.9: SODALITE meta-model (extension of DUL)

3.3.5 Example: Tier 1, Tier 2

In Tier 1, it is noteworthy the OWL2 meta-modelling capabilities of the resources, as they are both classes and instances. For example, in Figure 3.11, it is presented that the `sodalite.nodes.DockerizedComponent` is a class (`rdfs:subClassOf tosca.nodes.SoftwareComponent`), and also an instance, as it participates in a property assertion `soda:hasContext`. By using the meta-modelling (punning), we promote reuse as we can have subsumption hierarchies since the node types can be classes, and also have descriptive context (e.g. properties, capabilities) by the role of instance. Henceforth, the templates, which are instances of the types, inherit this descriptive knowledge of the types leading to more comprehensive and reusable knowledge components. Especially, in this domain where the knowledge is captured in different levels, namely normative types, resources and applications, the adoption of the meta-modelling enables this knowledge representation. Some examples of the Tier 1 and Tier 2 follow:

- **Tier 1:** In Figure 3.10, the SODALITE ODP capabilities are demonstrated through an instantiation of a pattern that captures the definition of a node type. The depicted node type is custom and inherits a normative TOSCA node type, the `tosca.nodes.SoftwareComponent`.



```
sodalite.nodes.DockerizedComponent:
    derived_from: tosca.nodes.SoftwareComponent
    attributes:
        ip_address:
          description: "IP address of dockerized component"
          type: string
    requirements:
            host:
              capability: tosca.capabilities.Compute
              node: docker/sodalite.nodes.DockerHost
              relationship: tosca.relationships.HostedOn
```

Fig. 3.10: Example TOSCA custom node type and high-level assignment of SODALITE ODP concepts

Since the purpose is to show the SODALITE ODP, only an excerpt of the node type is depicted. It is presented the correspondence between the TOSCA elements and the ODP concepts. Namely, each node type is a Situation, that contains a Description containing various concepts. The same rationale is followed in the static layer, Tier 0.

Figure 3.11 depicts an excerpt from the Knowledge Graph of the aforementioned node type. More specifically, the requirement and the attribute are captured as instances of the tosca:Requirement and tosca:Attribute correspondingly. Each concept classifies the property that is modelled, for example `tosca:host`. Any other nested information is captured through the definitions of the SodaliteParameters. Each concept, such as `ex:SodaliteParameter_2` in our example, can have a description by using the `dcterms:description` property of Dublin Core [11].

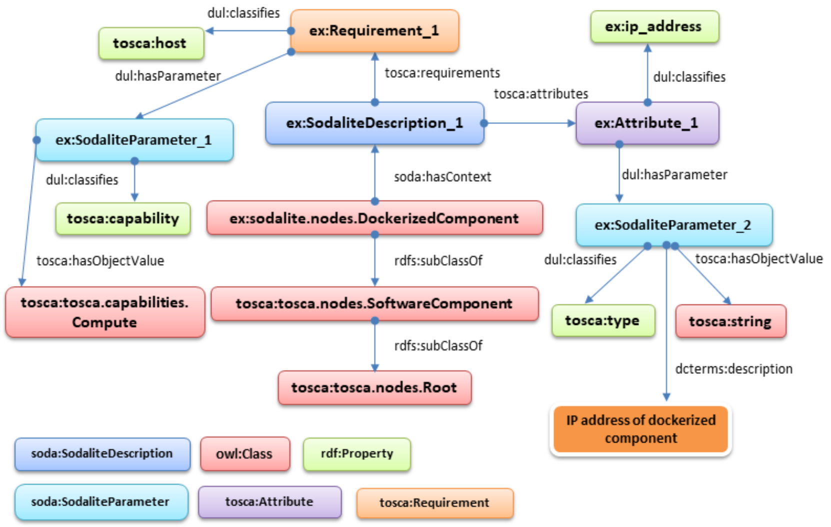


Fig. 3.11: Example node type

- **Tier 2:** In Figure 3.12, an excerpt of the `snow-weather-condition-filter` node template's knowledge graph is depicted, and its full version is shown in Listing 3.1.

```
snow-weather-condition-filter:
  type: docker/sodalite.nodes.DockerizedComponent
  properties:
    image_name: "snow-weather-condition-filter:latest"
    alias: "snow-weather-condition-filter"
```

List. 3.5: Example node template

The ODP is used similarly with the Tier 1. Precisely, each template is captured as a situation that has a description. Each description contains concepts such as properties, attributes, requirements etc. Each concept classifies a property and represents the nested knowledge as parameters. Listing 3.5 de-

[11] https://www.dublincore.org/specifications/dublin-core/dcmi-terms/

picts the definition of the snow-weather-condition-filter as an instance of the `sodalite.nodes.DockerizedComponent` resource.

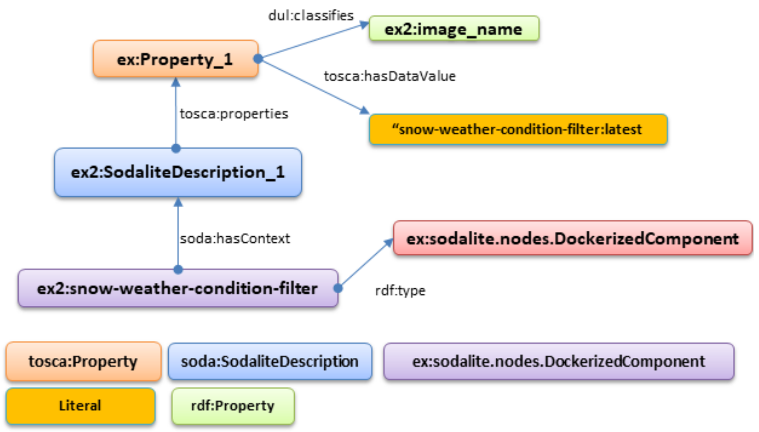

dul:classifies
ex2:image_name
ex:Property_1
tosca:hasDataValue
"snow-weather-condition-filter:latest
tosca:properties
ex2:SodaliteDescription_1
soda:hasContext
ex:sodalite.nodes.DockerizedComponent
ex2:snow-weather-condition-filter
rdf:type
tosca:Property
soda:SodaliteDescription
ex:sodalite.nodes.DockerizedComponent
Literal
rdf:Property

Fig. 3.12: Example node template ontology

3.4 A smart environment for developing IaC

The smartness in the SODALITE Modelling layer primarily arises from the semantic inference capabilities of the KB. This task is performed by the Semantic Reasoner. In this section we focus on the main features it offers.

3.4.1 Context-aware content assistance

Smart context-aware content assistance is offered to the user by means of suggestions during the authoring of the models. This modelling assistance is mostly based on the inference and reasoning capabilities of the KB. Since the models are saved as interconnected RDF knowledge graphs, the information is represented as a network of relationships that capture both the structural and semantic relationships in an definite manner. Thereby, the KB can be easily reasoned for supporting discovery, reuse, and validation. In respect of the validation of the models, in the section 4.2, we describe the intelligent services checking the validity of the AADM during the design phase. Regarding the discovery and reuse, some of the cases that the KB suggests to the modelers are:

- *Available infrastructure resources that satisfy the requirements of the application components*. Thus, the requirements of an application component can be completed more easily, as the KB can suggest suitable resources that can serve as a host, dependency, network and other requirements. An example SPARQL query, that is used for discovering the aforementioned resources, is depicted in Listing 3.6, which retrieves the type (`value`) that is defined in a specific requirement (`requirementName`) of a resource definition. Then templates of this specific type are discovered in order to satisfy requirements of an application component.

```
select ?r ?value
where {
        ?node soda:hasContext ?ctx.
        ?node rdfs:subClassOf tosca:tosca.entity.Root.
        ?ctx tosca:requirements ?r .
        ?r DUL:classifies ?p.
        ?r DUL:hasParameter [DUL:classifies tosca:node; DUL:hasRegion ?value] .
        FILTER (STRENDS (str(?p), ?requirementName)) .
}
```

List. 3.6: SPARQL Query returning the value of a specific requirement of a type definition

- *Concepts*. Concepts such as the interfaces, properties, and capabilities. They can be assigned to the infrastructure and application components depending from which type they are derived or instantiated. In Listing 3.7, a SPARQL query that discovers properties for an application component is depicted.

```
select distinct ?concept ?property ?value
where {
        ?resource soda:hasContext ?context .
        ?context tosca:properties ?concept .
        ?concept DUL:classifies ?property .
        OPTIONAL {?concept tosca:hasValue ?value .}
}
```

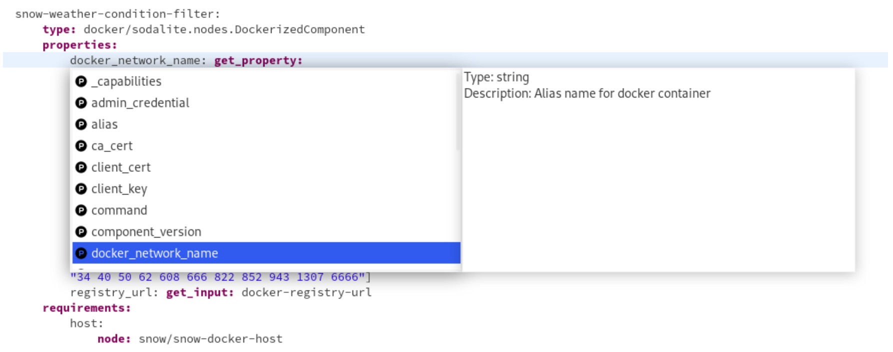
List. 3.7: SPARQL Query returning the properties of a resource type

- *Infrastructure and application components that belong to a specific group.* For example dockerized application components, or components of a specific use case e.g. snow. In Listing 3.8, a SPARQL query returning all the templates that are saved in the workspace of the snow use case. By leveraging how the data set is split in a Knowledge Base, the models can be grouped in graphs enabling the reusability as the users can discover models and resources among them.

```
select distinct ?template {
        ?template a soda:SodaliteSituation .

        GRAPH <https://www.sodalite.eu/ontologies/snow/>{
                        ?template rdf:type ?type .
        }
}
```

List. 3.8: SPARQL Query returning the templates in the snow workspace

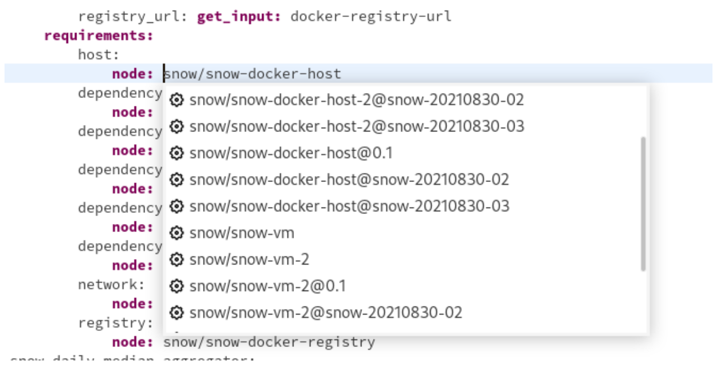


```
snow-weather-condition-filter:
    type: docker/sodalite.nodes.DockerizedComponent
    properties:
        docker_network_name: get_property:
```

Type: string
Description: Alias name for docker container

```
        "34 40 50 62 608 666 822 852 943 1307 6666"]
        registry_url: get_input: docker-registry-url
    requirements:
        host:
            node: snow/snow-docker-host
```

Fig. 3.13: Content assistance for properties



```
        registry_url: get_input: docker-registry-url
    requirements:
        host:
            node: snow/snow-docker-host
        dependency
            node:
        dependency
            node:
        dependency
            node:
        dependency
            node:
        dependency
            node:
        network:
            node:
        registry:
            node: snow/snow-docker-registry
```

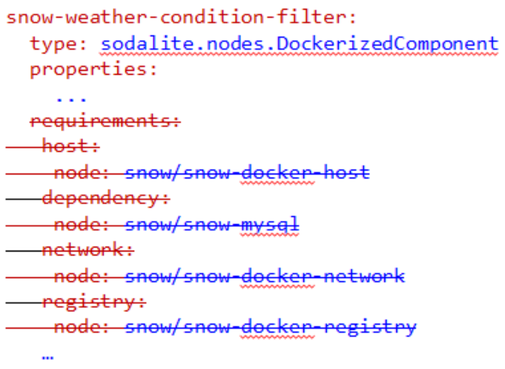
snow/snow-docker-host-2@snow-20210830-02
snow/snow-docker-host-2@snow-20210830-03
snow/snow-docker-host@0.1
snow/snow-docker-host@snow-20210830-02
snow/snow-docker-host@snow-20210830-03
snow/snow-vm
snow/snow-vm-2
snow/snow-vm-2@0.1
snow/snow-vm-2@snow-20210830-02

Fig. 3.14: Content assistance for requirements

The IDE relies on the `KB Reasoner` to assist DSL modelers with context specific suggestions at the point of edition. Upon request, the IDE displays a drop-down list of suggestions obtained from the `KB Reasoner` after sending a query that include the edition context. The user can browse the list, select one option, which will be inserted at the cursor point. In Figure 3.13 IDE offers the properties defined in the hierarchy of type `sodalite.nodes.DockerizedComponent`, whose defaults can be overridden for the instance `snow-weather-condition-filter`. In Figure 3.14 IDE offers available infrastructure resource instances (defined locally within the AADM or available in the KB) that can fulfill the `host` requirement of the `snow-weather-condition-filter` component.

3.4.2 Abstraction

Since the creation of the deployment model requires the acquaintance with various IaC languages, their execution mechanisms, and with different heterogeneous infrastructures, the abstraction of the SODALITE DSL is uppermost for lessening the modeller's effort. In pursuit of making the DSL as abstract as possible, we apply deep inference reasoning on the interlinked knowledge that is saved in KB so as to concretize the AADM. Precisely, with the abstracted DSL, some information in the deployment model can be omitted by the modeller.



```
snow-weather-condition-filter:
  type: sodalite.nodes.DockerizedComponent
  properties:
    ...
  ~~requirements:~~
~~   host:~~
~~     node: snow/snow-docker-host~~
~~   dependency:~~
~~     node: snow/snow-mysql~~
~~   network:~~
~~     node: snow/snow-docker-network~~
~~   registry:~~
~~     node: snow/snow-docker-registry~~
   …
```

Fig. 3.15: Abstraction DSL example

The abstraction primarily focuses on the requirements of an application component. The user can totally omit the requirements where, for instance, it is defined where an application can be hosted, which network is needed, which resources are dependencies such as database.In Figure 3.15, an excerpt of the snow-weather-condition-filter is depicted, where it is shown that all the requirements can be totally omitted by the modeller, and the KB can autofill the model.

3.5 Preparation of Container Images

Unlike the usage of Hypervisors for provisioning and virtualizing underlying hardware through virtual machines, container engines virtualize the operating system (such as Linux or Windows) having each container holding only the application and

```json
{
  "source_type": "git",
  "source_repo": {
    "url": "https://gitlab.com/wds-co/SnowWatch-SODALITE.git",
    "username": "repo_username",
    "password": "repo_password",
    "dockerfile": "WebCamCrawler/Dockerfile",
    "workdir": "."
  },
  "target_image_name": "snow-webcam-crawler",
  "target_image_tag": "latest"
}
```

List. 3.9: Example of definition of an application image build type for Snow

its libraries and the needed dependencies. Containers can leverage the features and resources of the host OS making them small, fast, and portable. Being lightweight containers improve CPU and memory utilization of physical machines. Containers also facilitate building of granular applications making them an ideal choice for service oriented architecture approach such as microservices architectures, making components independently deployable. At the same time containers provide the DevOps teams the level of flexibility and portability making them a great choice for running applications on heterogeneous environments such as multi-cloud, HPC or Edge.

Having the possibility to create deployable images and the tools for building these application images within SODALITE, a user can leverage transportability of application deployments between different systems and architectures. SODALITE uses the `Image Builder` component to pre-build application images for targeting an OS virtualizer such as Docker[12]. The Image Builder component itself is a dockerized REST API encapsulation of the xOpera lightweight orchestrator and a TOSCA/Ansible blueprint that is executed by the orchestrator. It can be configured to run different image building workflows enabling the user to build the application from source or tar images and push the created image to a Docker registry. Figure 3.16 shows the architecture of the `Image Builder` component. The image building workflows for building runtime images are running prior to deployment of the TOSCA blueprint, before the orchestrator starts with the execution of the blueprint deployment e.g., provisioning the infrastructure and deployment of the application. The building process can be automated through REST API calls or run manually from SODALITE Smart IDE (Section 3.2). In this case the user is able to check periodically for the status of the image building process. Since the image building process can take some time the REST API provides an asynchronous implementation of the image building. Listing 3.9 shows an example of a JSON build parameters for the `Snow WebCamCrawler` component.

The encapsulation of the xOpera lightweight orchestrator and TOSCA blueprints into the REST API enables the image building functionality to be accessible from

[12] `https://www.docker.com/`

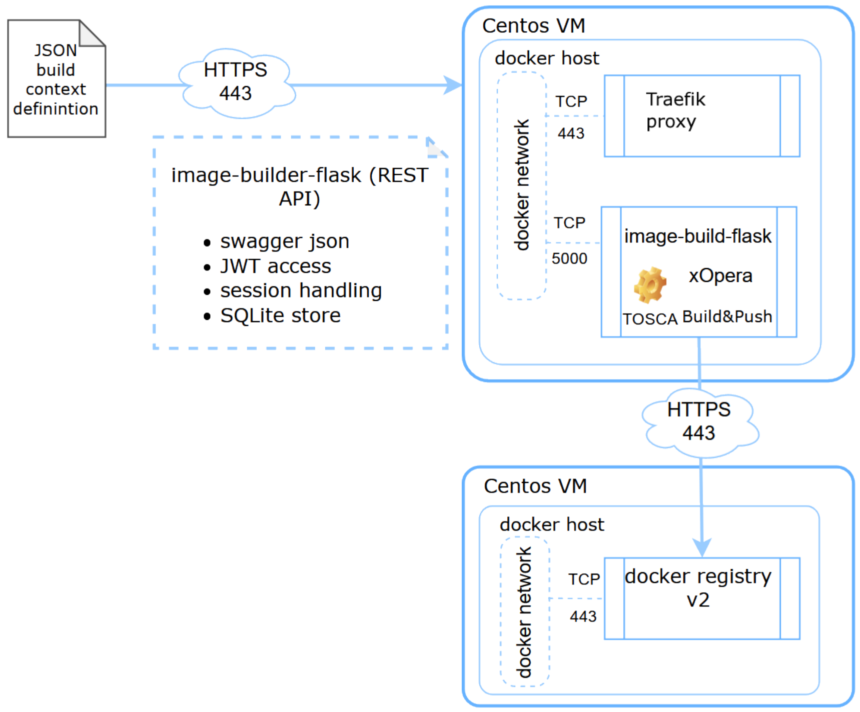


Fig. 3.16: Image Builder Architecture

any component in SODALITE framework or be just reused as a separate blueprint if needed. The extendable nature of these TOSCA blueprints provides a high level of reusability of the code for supporting the image building process. Image Builder implements the process of building the images through TOSCA v1.3 blueprint. Image builder also supports session handling and authentication/authorization by JWT tokens making it easy to integrate with Identity and Access Management providers.

An important innovative feature has been implemented giving the user the possibility to create multiple image variants in a single image building workflow run. Image builder exposes its functionalities through both multifunctional CLI and REST API, and can be also used in a CI/CD scenario. The image is regularly pushed to the public DockerHub registry under *sodaliteh2020/image-builder-api*[13]. The source code and extensive information on how to build, use and deploy the `Image builder` is provided in the Image Builder GitHub repository[14].

[13] https://hub.docker.com/r/sodaliteh2020/image-builder-api
[14] https://github.com/SODALITE-EU/image-builder

3.6 Infrastructure as Code preparation

The SODALITE runtime layer relies on the declarative OASIS TOSCA standard v1.3 [15] to receive the definition of infrastructural resources needed for a certain application, their relationships, and the mapping of application components onto these infrastructural elements. TOSCA is implementation agnostic, meaning that it does not suggest a recipe for the implementation of the node operation lifecycle, but, instead, specifies the interfaces associated to these lifecycle operations and can rely for their implementation on practically any high or low level language. SODALITE has chosen Ansible as a high level configuration management declarative language for the implementation of lifecycle operations for having maximum impact on DevOps teams that already use configuration management tools.

The goal of IaC preparation is the one of generating correct TOSCA code from AADMs written in the SODALITE DSL.

All SODALITE models, including abstract application deployment models (AADMs) and resource models (RMs), are stored in the semantic Knowledge Base (KB), as explained in Section 3.3. All the interactions to the KB are implemented through the Semantic Reasoner API, which handles the export of the models into a JSON model definition (AADM JSON). The AADM JSON model definition is self-sufficient and provides all the node definitions, optimization models (Section 3.2.2), application deployment topology definitions, as well as references to Ansible playbooks needed for the configuration of the application deployment and the creation of a valid self-contained TOSCA v1.3 blueprint in the Cloud Service Archive (CSAR) format, defined in the OASIS TOSCA standard. A CSAR contains definitions of all TOSCA node, relationship, artifacts types and templates, needed to deploy an application.

`IaC Blueprint Builder` is the component that takes the AADM JSON model definition and creates a valid CSAR, deployable through the SODALITE Orchestrator (xOpera) via its REST API endpoint. The IaC Blueprint Builder is implemented in Python and encapsulates three distinct subcomponents:

- *Swagger REST API* exposes and implements a REST API that forwards the calls for parsing of AADM JSON and creation of the TOSCA CSAR;
- *Abstract Model Parser* internally parses an AADM JSON and builds a TOSCA blueprint representation as a tree data structure,
- *IaC Blueprint Builder* creates a TOSCA blueprint based on the internal tree representation, packs the blueprint in a CSAR and registers the CSAR with the Orchestrator via xOpera REST API endpoint.

Additional actions are performed by the `IaC Blueprint Builder`, in cases an AADM JSON contains a reference to an Optimization Model (OM). The `Abstract Model Parser` extracts the node template, which has an optimization model associated to it, and creates an request to MODAK (Section 4.4) to return the reference

[15] https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html

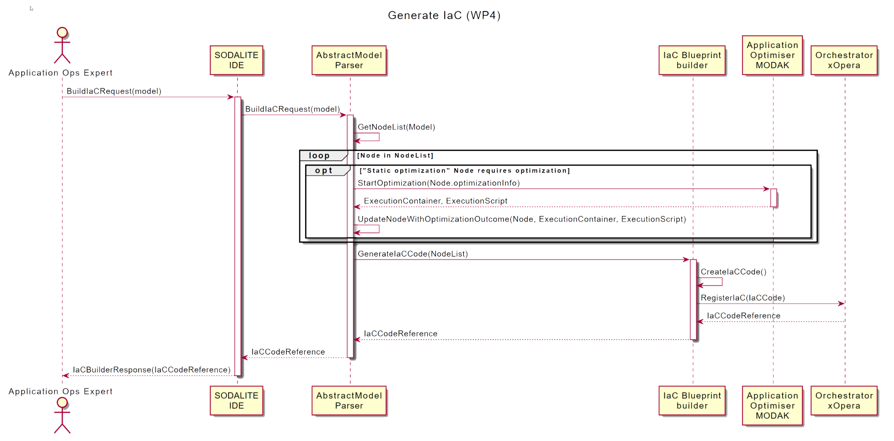


Fig. 3.17: UML sequence diagram showing the IaC generation

to the optimized container image and content of the optimized job script. The Parser then substitutes the returned values with respective properties (container runtime and job script content) of the node template, allowing the Orchestrator to retrieve and use the optimized container images and job scripts to deploy an application. Figure 3.17 presents a complete sequence diagram, where IaC generation from an AADM JSON and internal and external components interactions are outlined.

The IaC Blueprint Builder has been integrated into the SODALITE security pillar to check authentication and authorization of the calling client with the IAM Service API (Keycloak). The IaC Blueprint Builder is dockerized and its container image can be found in the public DockerHub under *sodaliteh2020/iac-blueprint-builder*[16]. Additionally, the source code is available in IaC Blueprint Builder GitHub repository[17].

3.7 Conclusion and Future Work

In this chapter, we have presented the SODALITE design-time IaC modeling framework. It permits SODALITE roles to model different facets of the deployment topology of complex applications across heterogeneous infrastructures. It adopts a combined meta-modeling (based on DSLs) and semantics (based on ontology graphs) approach, which leverages the high expressiveness of DSL human-centric languages and the deep knowledge inference and reasoning capabilities of ontology graphs. We have introduced the SODALITE DSLs, designed for modeling different aspects of optimized applications and the required infrastructure resources. We have

[16] https://hub.docker.com/r/sodaliteh2020/iac-blueprint-builder
[17] https://github.com/SODALITE-EU/iac-blueprint-builder

also introduced the IDE that offers textual and graphical editors for creating DSL model instances and for accessing the main SODALITE processes. The joint collaboration of the IDE and the KB helps the modeler with the intricacies of the models, helped by the context-aware content assist, the semantic validation and the smell detection. We have also introduced the ontology schema, created for assisting on the modeling of DSL model instances, split into tiers that accounts for the different facets of the deployment topology, leveraging the Descriptions and Situation Pattern. And finally, we have introduced the IaC framework, which consumes DSL models to generate the artifacts required for the application deployment by the Orchestrator of the SODALITE Runtime Layer. As future work, we plan to extend SODALITE DSLs and the semantic inference to cover the unsupported modeling capabilities of the TOSCA standard, to extend the content assistance to offer smarter recommendations and wider semantic validation, as well as the graphical modeling support for RMs.

References

[1] Stefan Biffl and Marta Sabou. *Semantic Web Technologies for Intelligent Engineering Applications*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 3319414887.

[2] H. Brabra et al. "On semantic detection of cloud API (anti)patterns". In: *Information and Software Technology*. Vol. 107. 2019, pp. 65–82.

[3] A. Brogi, A. D. Tommaso, and J. Soldani. "Sommelier: A Tool for Validating TOSCA Application Topologies". In: *5th International Conference on Model-Driven Engineering and Software Development*. 2017, pp. 1–22.

[4] Giuseppe De Giacomo and Maurizio Lenzerini. "TBox and ABox Reasoning in Expressive Description Logics". In: vol. 1996. Oct. 1996, pp. 37–48.

[5] Dieter Fensel. "Ontologies". In: *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer Berlin Heidelberg, 2001, pp. 11–18. DOI: 10.1007/978-3-662-04396-7_2.

[6] Aldo Gangemi and P. Mika. "Understanding the Semantic Web through Descriptions and Situations". In: *OTM*. 2003.

[7] Aldo Gangemi et al. "Sweetening Ontologies with DOLCE". In: *EKAW*. 2002.

[8] Birte Glimm, Sebastian Rudolph, and Johanna Völker. "Integrated Metamodeling and Diagnosis in OWL 2". In: Dec. 2010, pp. 257–272. DOI: 10.1007/978-3-642-17746-0_17.

[9] Bernardo Grau et al. "OWL2: The Next Step for OWL". In: *SSRN Electronic Journal* (Jan. 2008). DOI: 10.2139/ssrn.3199412.

[10] Thomas R. Gruber. "A translation approach to portable ontology specifications". In: *Knowledge Acquisition* 5.2 (1993), pp. 199–220. DOI: https://doi.org/10.1006/knac.1993.1008.

[11] Nophadol Jekjantuk, Gerd Gröner, and Jeff Pan. “Modelling and Reasoning in Metamodelling Enabled Ontologies”. In: vol. 4. 2010, pp. 277–290. DOI: `10.1007/978-3-642-15280-1_8`.

[12] Paul Lipton et al. “Tosca simple profile in YAML version 1.3”. In: *OASIS Committee Specification* 1 (2020).

[13] Georgios Meditskos et al. “A Pattern-based Semantic Lifting of Cloud and HPC Applications using OWL 2 Meta-modelling”. In: *Proceedings of the 4th Special Session on High Performance Services Computing and Internet Technologies*. 2020.

[14] Steffen Staab and Rudi Studer. *Handbook on Ontologies*. Jan. 2003. ISBN: 978-3540408345. DOI: `10.1007/978-3-540-92673-3`.

[15] Tania Tudorache. “Ontology engineering: Current state, challenges, and future directions”. In: *Semantic Web* 11 (Dec. 2019), pp. 1–14. DOI: `10.3233/SW-190382`.

[16] Moshe Y. Vardi. “Why is Modal Logic So Robustly Decidable?” In: *Descriptive Complexity and Finite Models*. 1996.

Chapter 4
Quality Assurance and Design-time Optimization

Indika Kumara, Alfio Lazzaro, Nina Mujkanovic, Zoe Vasileiou, and Damian A. Tamburri

Abstract Heterogeneous applications are getting more and more complex, making the authoring of their deployment models an error-prone and demanding task. Heterogeneous resources also make performance optimization of applications complex. In this chapter, we will present the quality assurance and application optimization support of the SODALITE framework, which offers the capabilities for verifying deployment models, detecting bugs and smells in them, and optimizing application components for specific hardware resources. This chapter discusses how the above-mentioned capabilities of the SODALITE framework can be used to develop optimized, defect-free deployment models.

4.1 Introduction

The SODALITE modeling layer produces the deployment model of an application in terms of Infrastructure as Code (IaC) scripts. IaC simplifies the provision and configuration of the IT infrastructure at scale. As the size and complexity of IaC

Indika Kumara
Jheronimus Academy of Data Science, Tilburg University, The Netherlands, e-mail: i.p.k.weerasinghadewage@tilburguniversity.edu

Alfio Lazzaro
HPE HPC/AI EMEA Research Lab, Switzerland, e-mail: alfio.lazzaro@hpe.com

Nina Mujkanovic
HPE HPC/AI EMEA Research Lab, Switzerland, e-mail: nina.mujkanovic@hpe.com

Zoe Vasileiou
Information Technologies Institute, Centre for Research and Technology Hellas, Greece, e-mail: zvasilei@iti.gr

Damian A. Tamburri
Jheronimus Academy of Data Science, Technical University Eindhoven, The Netherlands, e-mail: d.a.tamburri@tue.nl

projects increase, it is critical to maintaining the code and design quality of IaC Scripts [6, 7]. According to a recent report on Cloud Threat [1], nearly 200,000 insecure IaC templates were found among IaC scripts used by a set of enterprises, and 65% of cloud incidents are due to misconfigurations. Thus, the detection and correction of defective and erroneous IaC scripts are of paramount importance. To address this problem, the SODALITE platform offers a set of tools to detect defects such as errors and code smells.

In addition to the generation of the IaC scripts, the deployment process can also create container images for application components. Software application developers and users are now targeting diverse computing platforms, such as on-premise supercomputers and clouds with heterogeneous node architectures. Compute intensive applications such as Artificial Intelligence (AI) training that use High-Performance Computing (HPC) have specific requirements for specialized execution environments, including computing accelerators, high speed interconnects, and fast memory and storage. Even if software-defined environments provide flexibility and portability, we still need applications to use and benefit from these diverse resources optimally. For example, AI training frameworks require target-specific libraries and drivers to be configured. In the context of HPC infrastructures, with various hardware and software dependencies and libraries, building or selecting an optimized container for deploying AI-based components is crucial. The same concepts apply to Message Passing Interface (MPI) applications, where the applications have to efficiently use the network to get performance and parallel scalability. To address these issues, SODALITE offers an application optimizer called MODAK that maps the optimal application parameters to the infrastructure target by building or selecting an optimized container and then encoding optimizations in a job script.

The rest of this chapter is organized as follows. Section 4.2 presents the support for validating the deployment topology of an application and verifying arbitrary constraints on the components and their properties. Section 4.3 discusses the detection of smells and bugs in IaC using rule-based and data-driven approaches. Section 4.4 presents the MODAK tool in detail, and Section 4.5 concludes the chapter.

4.2 Verifying IaC

Validation services are provided to the user during the authoring process of the deployment models. Based on the deployment models saved as interconnected Knowledge Graphs, described in Chapter 3, powerful semantic queries can run upon the Knowledge Base using strong inference for uncovering new information out of existing relations. Additional to advanced context-aware searching, matchmaking, and reuse, described in the previous chapter, pre-deployment validation is a crucial component that ensures a reliable IaC deployment model.

[1] https://www.paloaltonetworks.com/prisma/unit42-cloud-threat-research-1h21

The validation of the AADM, during the design phase, is aiming at checking the consistency of the structures. In TOSCA, the type system supports inheritance as a type can extend another, inheriting all its concepts (e.g. properties, capabilities). Each template of the AADM is an instance of a specific type, namely an infrastructure resource or software component, and gets validated against this type definition.

4.2.1 Validation cases

Using custom reasoning logic, semantic validation errors can be inferred with regards to the TOSCA type definition. The assigned values to the component templates are validated against the corresponding type schema.

4.2.1.1 Topology Validation

There are errors in the deployment model that are onerous to be manually detected as it is needed to manually check all inter-node relationships in a TOSCA application topology and their interconnection constraints. Based on the validity conditions of the Sommelier[3], an open-source validator of TOSCA application topologies, our services are validating the interconnections of the deployment model. 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).

In TOSCA [9], various components, such as an application, a database, are modeled as templates and are instances of types, such as node types, relationship types, and capability types. The node types contain the definitions of the requirements of a component, the capabilities that are offered for other components. The capability types express the capabilities and `valid_source_types` (valid names of Node Types that are supported as valid sources of any relationship). The relationship types denote the explicit relationships between the nodes, or alternatively implicit relationships are declared through requirements.

4.2.1.2 Required Properties

In the type schema, it is optionally to be defined if a property is required to be assigned to a template by the *required* key. Therefore, if there exists a property definition in a type and *required* equals true, and there is no *default* value, then such a property should be assigned to the templates being instances of this type. In Listing 4.1, a TOSCA node type definition is depicted with the *name* mandatory property. In Listing 4.2, a SPARQL query detecting the required properties is shown.

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

List. 4.1: Excerpt of a TOSCA node type definition with a required property

```sparql
select distinct ?property
where {
    ?resource soda:hasInferredContext ?context .
    ?context tosca:properties ?concept .
    ?concept DUL:classifies ?property .
    {
        ?concept DUL:hasParameter [DUL:classifies tosca:required; tosca:hasDataValue
        ↪  true].
    } UNION {
        FILTER NOT EXISTS {?concept DUL:hasParameter
                            [DUL:classifies tosca:required; tosca:hasDataValue []]}.
    }
}
```

List. 4.2: SPARQL Query detecting required properties

4.2.1.3 Property Values

Each property definition of the node type includes a type of the assigned property value. There are various property types such as `string`, `integer`, `list`, and `map`. A node type that has two properties with the type string and integer defined is shown in Listing 4.3. Rule-based reasoning infers if the assigned template property values are valid according to the type, using SPARQL queries upon the Knowledge Graphs.

```yaml
sodalite.nodes.DockerizedComponent:
    derived_from: tosca.nodes.SoftwareComponent
    properties:
        client_key:
            description: "Path tot he client's TLS key file."
            type: string
            required: false
            default: ""
        sleep:
            description: "Sleep after image is deployed"
            type: integer
            required: false
            default: 0
```

List. 4.3: Part of a TOSCA node type with properties of different types

4.2.1.4 Constraints

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 with valid values, shown in Listing 4.4 or a given range (e.g. greater than, less than), or as complex as an object of a custom type. In Listing 4.5, a SPARQL query is shown that retrieves the properties of a type that have constraints with a list.

```
sodalite.datatypes.modak.optimisation.opt_build:
    derived_from: tosca.datatypes.Root
    properties:
        cpu_type:
            type: string
            constraints:
                - valid_values: [ "x86", "arm", "amd", "power" ]
        acc_type:
            type: string
            constraints:
                - valid_values: [ "nvidia", "amd", "fpga" ]
```

List. 4.4: Part of a TOSCA data type with property value constraints

```
select distinct ?constraint ?constr_type ?value ?listvalue
where
{
    ?var soda:hasInferredContext ?context .
    ?context tosca:properties ?concept .
    ?concept DUL:classifies ?property .
    ?concept DUL:hasParameter [DUL:classifies tosca:CONSTRAINTS; DUL:hasParameter
        [DUL:classifies ?constraint; tosca:hasValue ?listvalue]].
    ?listvalue rdf:type tosca:List .
    ?concept DUL:hasParameter [DUL:classifies tosca:type; tosca:hasValue ?constr_type].
}
```

List. 4.5: SPARQL Query returning only the constraints of a type including a list

4.3 Detecting Smells and Linguistic Anti-patterns in IaC

SODALITE developed the tools that can detect such smells and linguistic anti-patterns in IaC. A software smell is any characteristic in the artifacts of the software that possibly indicates a deeper problem or quality issue [11]. Linguistic anti-patterns are recurring poor practices concerning inconsistencies among the naming, documentation, and implementation of an entity, which have shown to be a good proxy for defect prediction [1].

4.3.1 Semantic Approach to Detecting Smells

SODALITE proposes a semantic rule-based approach to detect the smells and antipatterns in IaC, for example, smells in TOSCA blueprints [8]. Our framework facilitates the generation of knowledge graphs to capture TOSCA-based deployment models. The aim is to map IaC code constructs to self-contained, independent, and reusable knowledge components, amenable to analysis and validation using Semantic Web standards, such as SPARQL. A semantic approach helps us to deal with structure and semantic relations over various types of resources, their relationships, and properties. The semantic reasoning process is able to draw new and hidden knowledge from the existing information.

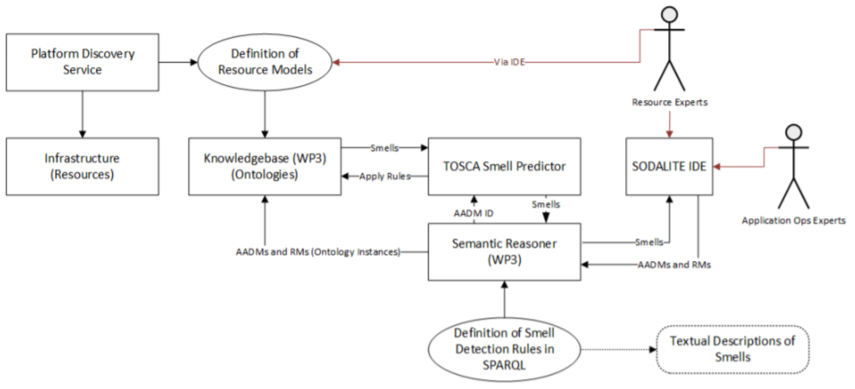

Platform Discovery Service
Definition of Resource Models
Via IDE
Resource Experts
Infrastructure (Resources)
Knowledgebase (WP3) (Ontologies)
Smells
Apply Rules
TOSCA Smell Predictor
SODALITE IDE
Application Ops Experts
AADM ID
Smells
AADMs and RMs (Ontology Instances)
Semantic Reasoner (WP3)
Smells
AADMs and RMs
Definition of Smell Detection Rules in SPARQL
Textual Descriptions of Smells

Fig. 4.1: An Overview of our Approach to TOSCA Smell Detection

Figure 4.1 shows the high-level architecture and workflow of our approach to detect the occurrences of smells in deployment model descriptions. More specifically:

- **Population of Knowledgebase.** Resource Experts populate the knowledgebase by creating resource models (ontology instances representing resources/nodes in the infrastructure) using SODALITE IDE. Platform Discovery Service may (semi-)automatically update the knowledge base by creating resources models.
- **Definition of Smells Detection Rules.** We use the semantic rules in SPARQL to detect different smells in deployment models. SODALITE developed rules to detect common security and implementation smells. New, additional rules can be defined to detect new types of smells.

- **Detection of Smells.** Application Ops Experts create the AADM instances for representing the deployment models of the applications. The AADM is automatically translated into the corresponding ontological representation and is saved in the knowledgebase. The smell detection rules are applied over the ontologies in the knowledgebase to detect deployment model-level smells. If a smell is detected, the details of the smell are returned to the Application Ops Experts. The detected smells are shown in the IDE as warnings. The same flow applies to Resource Ops Experts, as they also receive warnings for their resource models.

Table 4.1: Smells, their Descriptions, and the Abstract Detection Rules

Smell	**Smell Description**	**Abstract Detection Rule**
Admin by default	Default users are administrative users.	isUser (x.name) ∧ isAdmin (x.name)
Empty password	A password as a zero-length string.	isPassword(x.name) ∧ (isEmpty(x.value) ∨ isEmpty(x.defaultValue))
Hard-coded secret	Secrets such as usernames and passwords are hardcoded.	(isPassword(x.name) ∨ isUser (x.name) ∨ isSecKey (x.name)) ∨ ~(isEmpty(x.value) ∨ isEmpty(x.defaultValue))
Suspicious comment	A comment includes the information indicating secrets and buggy implementations, etc.	hasComment(x) ∧ isSuspicious(x.comment)
Unrestricted IP address	Using "0.0.0.0" or "::" as binding IP addresses of servers	isIPAddress(x.name) ∧ (isInvalidBind (x.value) ∨ isInvalidBind (x. defaultValue))
Insecure Communication	Using insecure communicate protocols, instead of secure their counterparts	(isURL(x.value) ∧ isInsecure(x.value)) ∨ (isURL(x.defaultvalue) ∧ isInsecure(x.defaultvalue))
Weak Crypto. Algo.	Use of weak cryptography algorithms such as MD5 and SHA1	hasWeakAlgo(x.value) ∨ hasWeakAlgo(x.defaultvalue)
Insufficient Key Size	The key used by an encryption algorithm is less than the recommended key size, e.g., 2048 bits for RSA algorithm.	isCryptoKeySize(x.name) ∧ (hasInsufficientKeySize(x.value) ∨ hasInsufficientKeySize(x.defaultvalue))
Inconsistent naming convention	The conventions used for naming nodes, properties, attributes, etc., are inconsistent.	(case==‘CamelCase’→ isCamelCase(x)) ∨ (case==’SnakeCase’→ isSnakeCase(x)) ∨(case==’DashCase’→ isDashCase(x))
Invalid Port Ranges	TCP port values are not within the range from 0 to 65535.	isTCPPort(x) ∨ (outOfValidRange (x.value) ∧ outOfValidRange (x.defaultvalue))

Table 4.1 shows the (abstract) rules to detect 10 TOSCA smells. The rules are implemented as SPARQL queries for specifying detection rules. Listing 4.6 shows an excerpt from the SPARQL query for detecting Admin by default smell. Line 4 implements the function isUser using a regex matching. Lines 5-9 retrieve the default value for a property of a node. Line 14 realizes the function isAdmin using the IN operator. The SPARQL queries for the other smells are available online in the SODALITE GitHub repository.

4.3.2 A Learning-based Approach for Detecting Linguistic Anti-patterns

We develop a novel approach to detect linguistic anti-patterns in IaC using deep learning and word embeddings [2]. We focus on name-body inconsistencies in IaC

```
select distinct ?property ?propertyDef
where {
    ?property DUL:classifies ?propertyDef.
    FILTER(regex(str(?propertyDef),"user(.+?)|(.+?)?user","i")).
    optional { # node type definitions - tier1
        ?property DUL:hasParameter ?p .
        ?p DUL:classifies tosca:default .
        ?p tosca:hasDataValue ?value.
      }.
    optional { #node template definitions - tier0
        ?property tosca:hasDataValue ?value.
      }
    FILTER (bound(?value)).
    FILTER (str(?value) IN ('admin', 'root'))
}
```

List. 4.6: Part of AdminByDefault SPARQL Query.

code units, for example, tasks in Ansible playbooks or roles. We use the Convolutional Neural Networks(CNN) [5] as the deep learning algorithm, and Word2Vec [4] as the word embedding method. CNNs are neural networks that consist of neurons with learnable weights and biases. Word2vec is a two-layer neural network that processes text by creating vector representations from words.

Figure 4.2 shows the workflow of our approach:

- **Corpus Tokenization.** Given a corpus of Ansible tasks, this phase generates token streams for both task names and bodies. To tokenize a task's body while considering its semantic properties, we build and use its abstract syntax tree.
- **Data Sets Generation.** Finding a sufficient number of real buggy task examples containing inconsistencies is challenging. Therefore, as in [10], we apply simple code transformations to generate buggy examples from likely correct examples. We perform such transformations on the tokenized data set and assume that most corpus tasks do not have inconsistencies.
- **From Datasets to Vectors.** We employ Word2Vec to convert the token sequences into distributed vector representations (code embeddings). We train a deep learning model for each Ansible module type as our experiments showed a single model does not perform well, potentially due to low token granularity. Thus, the tokenized data set is divided into subsets per module, and the code embeddings for each subset are separately generated.
- **Model Training.** This phase feeds the code embeddings to a CNN model and trains the model to distinguish between the tasks having name-body inconsistencies from correct tasks. The trained model is stored in the model repository.
- **Inconsistency Detection.** The trained models (classifiers) from the model repository are employed to predict whether the name and body of a previously unseen Ansible task are consistent or not. Each task is transformed into its corresponding vector representations, which can be consumed by a classifier.

We evaluated our approach with an Ansible dataset systematically collected from open source repositories. Table 4.2 presents the inconsistency detection results for

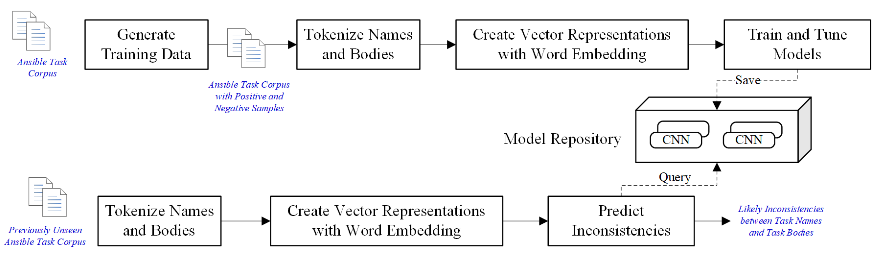

Ansible Task Corpus
Generate Training Data
Ansible Task Corpus with Positive and Negative Samples
Tokenize Names and Bodies
Create Vector Representations with Word Embedding
Train and Tune Models
Save
Model Repository
CNN
CNN
Query
Previously Unseen Ansible Task Corpus
Tokenize Names and Bodies
Create Vector Representations with Word Embedding
Predict Inconsistencies
Likely Inconsistencies between Task Names and Task Bodies

Fig. 4.2: Linguistic Anti-patterns Detection Pipeline

the top 10 Ansible modules in our data set. Overall, our approach yielded an accuracy ranging from 0.785 to 0.915, AUC metric from 0.779 to 0.914, and MCC metric from 0.570 to 0.830. Our approach achieved the highest performance for detecting inconsistency in the file module, where the accuracy was 0.915, the F1 score for the inconsistent class was 0.92, and the F1 score for the consistent class was 0.91.

Table 4.2: Classification results for the top 10 used Ansible modules

Evaluation Metric/Module		**shell**	**command**	**set_fact**	**template**	**file**	**gather_facts**	**copy**	**service**	**debug**	**fail**
Inconsistent	Precision	0.880	0.790	0.770	0.820	0.900	0.900	0.860	0.870	0.870	0.820
	Recall	0.810	0.840	0.900	0.940	0.940	0.830	0.810	0.760	0.770	0.690
	F1 score	0.843	0.814	0.830	0.876	0.920	0.864	0.834	0.811	0.817	0.749
Consistent	Precision	0.810	0.820	0.890	0.930	0.930	0.905	0.82	0.800	0.750	0.760
	Recall	0.890	0.770	0.750	0.800	0.890	0.770	0.870	0.900	0.860	0.870
	F1 score	0.848	0.794	0.814	0.860	0.910	0.870	0.844	0.847	0.801	0.811
	Accuracy	0.847	0.805	0.819	0.868	0.915	0.817	0.838	0.833	0.809	0.785
	MCC	0.697	0.610	0.649	0.744	0.830	0.685	0.678	0.669	0.625	0.570
	AUC	0.848	0.804	0.822	0.868	0.914	0.848	0.838	0.830	0.814	0.779

4.4 Optimizing Containerized Applications

The MODAK (Model Optimized Deployment of Applications in Containers) package, a software-defined optimization framework for containerized MPI and AI applications, is the SODALITE component responsible for enabling the static optimization of applications before deployment. Application optimization is enabled using performance modeling and container technology. Containers provide an optimized runtime for application deployment based on the target hardware and along with any software dependencies and libraries. MODAK aims to manage the optimized application containers for the deployment to infrastructure in a software-defined way.

4.4.1 Architecture

Figure 4.3 gives an overview of the MODAK components. MODAK exposes a high-level application API for the two types of applications supported: AI training and inference and MPI-parallelized applications. We pass this information to MODAK, which matches it with the performance model outputs to produce a job script for the execution submission of the optimized container. MODAK can also auto-tune and auto-scale applications based on user input. MODAK requires the following inputs:

- Job submission options for batch schedulers such as SLURM and TORQUE
- Application configuration such as application name, run and build commands
- Optimization DSL with the specification of the target hardware, software libraries, and optimizations to encode, as well as inputs for auto-tuning and auto-scaling. Examples of the DSL are provided in Section 6.4.4.

After providing the inputs, MODAK produces a job script (for batch submission) and retrieves a pre-built optimized container that can be used for application deployment. An image registry contains MODAK optimized containers, while performance models, optimization rules, and constraints are stored and retrieved from the Model repository. The Singularity container technology was chosen to provide a portable and reproducible runtime for the application deployment, due to better performance and native support for HPC resources than other popular container technologies. In the section 4.4.2 we describe in detail each MODAK component with the related features.

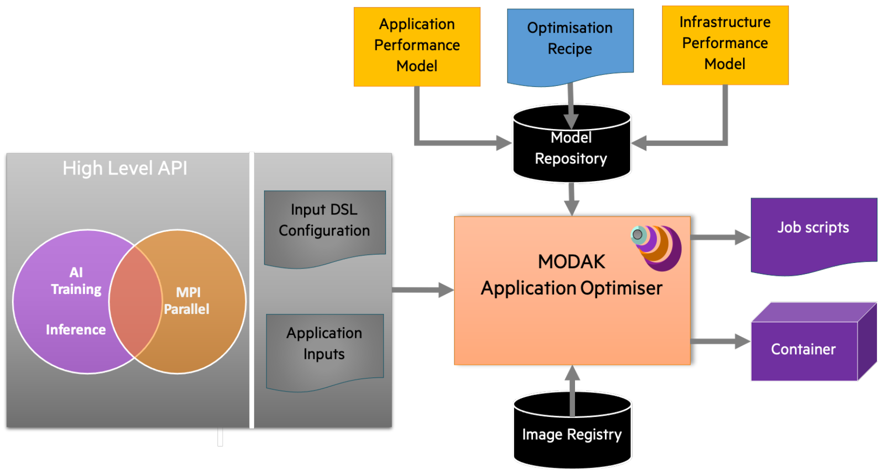


Fig. 4.3: MODAK Architecture.

4.4.2 Features

MODAK automates optimization using four main components, as described below:

- Mapper. The Mapper maps application deployment to an optimized container based on the user-specified input (DSL). While most AI applications are deployed in containers, this is not the default option for MPI parallel applications. Containers should provide an optimized runtime for the application deployment. With diverse hardware and software dependencies and libraries, building or selecting an optimized container for application deployment is crucial. For example, MPI libraries on the host machine and in the container should match when deploying applications on HPC systems in order for the container to use the hardware-optimized version of MPI available on the host. AI training frameworks require target-specific libraries and drivers to be configured. Even though Docker and Singularity support labeling containers, they are seldom used when developing them. To overcome this issue, containers are pre-built for different hardware and MODAK labels them with supported hardware and software information, including any optimizations. An application user uses a similar JSON format to query for an optimized container, and the mapper returns the container type, location, and file name. The user can pull the container from the hub and execute the application with that runtime. Currently, MODAK supports TensorFlow, PyTorch, MXNet, MPICH, OpenMPI, and MVAPICH2 containers for x86 and NVIDIA GPUs. This can be further extended to support specific network interconnects, and storage filesystems like Lustre.
- Enforcer. The optimization process depends not only on application and infrastructure but also on the configuration and data. MODAK allows users to define optimization rules that are enforced for deployment. The Enforcer component returns the optimization script to be used based on the rules and user-selected optimizations in the input DSL. For example, enabling graph compiler-based optimizations in an AI framework requires environment settings to be modified. For MPI-based applications, there are many environment settings that change the way message passing is optimized based on message size and communication pattern. Data-related optimizations may involve the possibility to automatically copy the data to fast disks, if available, to improve I/O bound applications. MODAK can embed the chosen optimizations in the job script submitted to a batch scheduler.
- Autotune. Applications and their dependencies have many configurable parameters which can drastically change performance when altered. Tuning all the parameters is both resource-intensive and time-consuming. Autotuning frameworks help make automated choices regarding application build and deployment, the algorithms they use, and the way the application is launched or changes code.
- Autoscale. Scaling applications to more nodes improves the performance of most MPI parallel applications. The parallel speedup and scaling efficiency is defined as follows

$$\text{Parallel Speedup} = \frac{T_{\text{ref}}}{T_{\text{parallel}}} \tag{4.1}$$

$$\text{Efficiency} = \frac{n_{\text{ref}}\,T_{\text{ref}}}{n\,T_{\text{parallel}}} \tag{4.2}$$

where T_{ref} and T_{parallel} correspond to the runtime on a reference number of nodes n_{ref} (usually a single node), and the runtime on n nodes, respectively. While we aim to achieve higher speedups as we increase nodes, poor efficiency denotes higher overheads and higher costs. Applications are usually scaled until the efficiency drops below a certain percentage. In MODAK, we can predict the efficiency and speedup of an application on n nodes based on the performance prediction model. This allows MODAK to automatically scale applications to a certain number of nodes based on the model prediction. Using the parallel efficiency metric specified by the user, Autoscale aims to predict the scale at which parallel efficiency is achieved, and automatically increase the number of nodes of the deployment.

4.5 Conclusion and Future Work

In this chapter, we have presented the design-time quality assurance and optimization support of the SODALITE framework. To enable the deployment of defect-free IaC scripts, we offer the tools to verify IaC scripts against various constraints, and defect smells and linguistic anti-patterns in them. We use semantic rule-based techniques and deep learning-based techniques, as appropriate. Moreover, to optimize AI or MPI workloads with different configurations and data sets for heterogeneous infrastructure targets, we introduced MODAK, a novel tool that maps optimal application parameters to infrastructure using performance modeling and container technology. MODAK optimized containers were tested on the internal SODALITE HPC Testbed. The test scenarios were taken from the SODALITE use cases compute-intensive tasks. We found that the performance boost of using optimized application containers can reach up to 10x compared wit the unoptimized versions of the application.

As future work, we plan to extend our smell and defect detection support to detect more linguistic inconsistencies and misconfigurations in different IaC languages. We will also extend MODAK to support machine learning applications for the edge.

References

[1] Venera Arnaoudova et al. “A New Family of Software Anti-patterns: Linguistic Anti-patterns”. In: *2013 17th European Conference on Software Maintenance and Reengineering*. 2013, pp. 187–196. DOI: 10.1109/CSMR.2013.28.

[2] Nemania Borovits et al. “DeepIaC: Deep Learning-Based Linguistic Anti-Pattern Detection in IaC”. In: MaLTeSQuE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 7–12.

[3] A. Brogi, A. D. Tommaso, and J. Soldani. “Sommelier: A Tool for Validating TOSCA Application Topologies”. In: *5th International Conference on Model-Driven Engineering and Software Development*. 2017, pp. 1–22.

[4] KENNETH WARD CHURCH. “Word2Vec”. In: *Natural Language Engineering* 23.1 (2017), pp. 155–162. DOI: `10.1017/S1351324916000334`.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[6] Indika Kumara et al. “Quality Assurance of Heterogeneous Applications: The SODALITE Approach”. In: *Advances in Service-Oriented and Cloud Computing*. Ed. by Christian Zirpins et al. Cham: Springer International Publishing, 2021, pp. 173–178. ISBN: 978-3-030-71906-7.

[7] Indika Kumara et al. “The do’s and don’ts of infrastructure code: A systematic gray literature review”. In: *Information and Software Technology* 137 (2021), p. 106593. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2021.106593`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584921000720`.

[8] Indika Kumara et al. “Towards Semantic Detection of Smells in Cloud Infrastructure Code”. In: WIMS 2020. Biarritz, France: Association for Computing Machinery, 2020, pp. 63–67.

[9] Paul Lipton et al. “Tosca simple profile in YAML version 1.3”. In: *OASIS Committee Specification* 1 (2020).

[10] Michael Pradel and Koushik Sen. “DeepBugs: A Learning Approach to Name-Based Bug Detection”. In: *Proc. ACM Program. Lang.* 2 (Oct. 2018). DOI: `10.1145/3276517`.

[11] Tushar Sharma and Diomidis Spinellis. “A survey on software smells”. In: *Journal of Systems and Software* 138 (2018), pp. 158–173. ISSN: 0164-1212.

Chapter 5
The SODALITE Runtime Environment

Indika Kumara, Giovanni Quattrocchi, Dragan Radolović, Kamil Tokmakov, Jesús Ramos Rivas, and Willem-Jan Van Den Heuvel

Abstract Modern applications need to be dynamically orchestrated on heterogeneous infrastructures for reasons such as performance, regulation compliance, or cost. This chapter presents the SODALITE runtime environment that can deploy, monitor, and manage applications on heterogeneous infrastructures consisting of Cloud, HPC, and Edge resources. The SODALITE runtime deploys the applications in the target infrastructures based on the deployment artifacts generated by the SODALITE model-driven approach presented in Chapter 3. It can also monitor the deployed applications and their infrastructure resources, generate alerts, and adapt application deployments.

Indika Kumara
Jheronimus Academy of Data Science, Tilburg University, The Netherlands, e-mail: `i.p.k.weerasinghadewage@tilburguniversity.edu`

Kamil Tokmakov
High Performance Computing Center Stuttgart (HLRS), Germany, e-mail: `kamil.tokmakov@hlrs.de`

Giovanni Quattrocchi
Politecnico di Milano, Italy, e-mail: `giovanni.quattrocchi@polimi.it`

Dragan Radolović
XLAB, Slovenia e-mail: `dragan.radolovic@xlab.si`

Jesús Ramos Rivas
ATOS, Spain e-mail: `jesus.ramos.external@atos.net`

Willem-Jan Van Den Heuvel
Jheronimus Academy of Data Science, Tilburg University, The Netherlands e-mail: `w.j.a.m.v.d.heuvel@jads.nl`

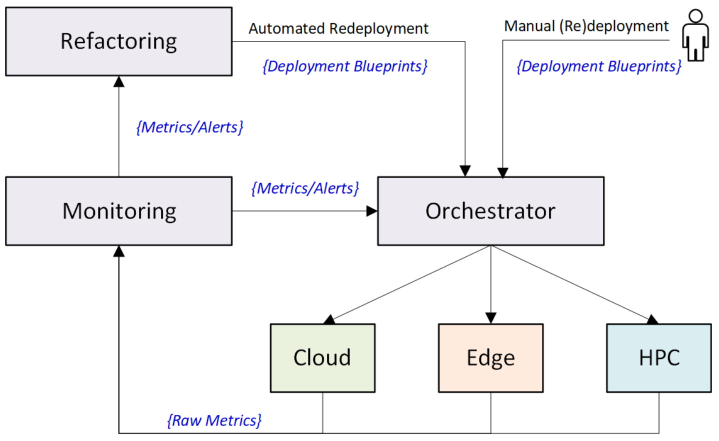


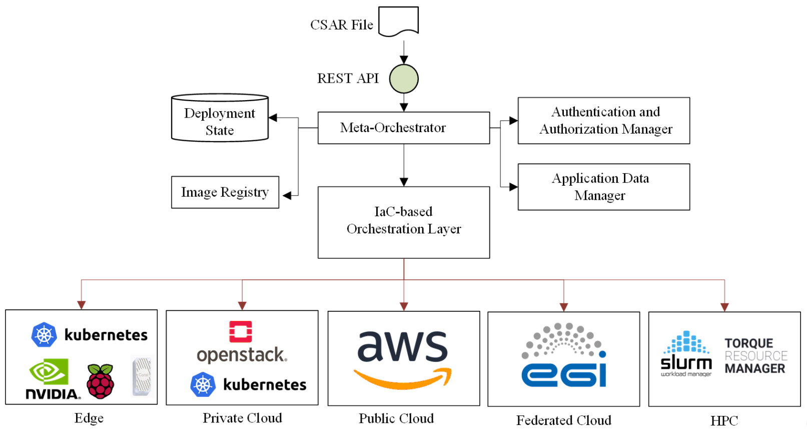
Fig. 5.1: The architecture of the SODALITE runtime environment

5.1 Introduction

Modern computing infrastructures consist of heterogeneous, software-defined, high-performance computing environments and resources, including Cloud servers, Edge accelerators, HPC clusters, and Serverless platforms. Advanced applications require complex and heterogeneous deployments that match their components with the infrastructure that offers the best performance fulfilling their requirements. In this context, SODALITE aims to address this heterogeneity by providing a toolset that enables developers and infrastructure operators to achieve faster development, deployment, and execution of applications on different heterogeneous infrastructures In particular, the runtime layer of SODALITE is responsible for the orchestration, monitoring, and adaptation of applications on these infrastructures.

Figure 5.1 shows the high-level architecture of the *SODALITE* runtime environment, which consists of the components *Orchestrator*, *Monitoring System*, and *Refactoring System*. *Orchestrator* is responsible for (re)deploying a given application on the Cloud-Edge-HPC hybrid infrastructures by executing IaC scripts as necessary. It receives the initial deployment model (from a developer) or a new alternative deployment model (from *Refactoring* System) as a TOSCA model instance. The developers can use the *SODALITE IDE* to create deployment models for applications and trigger their (re)deployment. *Monitoring System* collects different metrics and events from both the application and Cloud-Edge-HPC infrastructure. It can also emit alerts, which are complex events over metrics or simple events. In response to the events from *Monitoring System*, *Refactoring* may decide to modify and reconfigure the current deployment model instance of the application.

In the rest of this chapter, we discuss the *SODALITE* runtime environment in detail. We first present the design and capabilities of the *Orchestrator* (Section 5.2), highlighting its deployment and redeployment operations. Next, we focus on the



The

Fig. 5.2: Architecture of the *Orchestrator*

approach to support data management and data transfer between the cloud and HPC clusters (Section 5.3) and on monitoring applications and infrastructure with the *Monitoring System* (Section 5.4). Finally, our support for the adaptation of the deployment models (Section 5.5) and managing resources at runtime (Section 5.6) is discussed.

5.2 Orchestrating Applications

The *SODALITE Orchestrator* is capable of deploying, undeploying, and redeploying applications over heterogeneous infrastructures. The applications to be deployed are packaged as CSAR (TOSCA Cloud Service Archive) files. Our *Orchestrator*, namely xOpera [1], is a meta-orchestrator that coordinates multiple low-level resource orchestrators. xOpera is compliant with TOSCA YAML v1.3 standard.

5.2.1 Architecture of Orchestrator

Figure 5.2 shows the high-level architecture of the *Orchestrator*, which mainly consists of *Meta-Orchestrator*, *IaC-based Orchestration Layer*, *Image Registry*, *Authentication and Authorization Manager*, and *Application Data Manager*.

- **Meta-Orchestrator** coordinates the low-level resource orchestrators of the execution platforms through the *IaC-based Orchestration Layer* to deploy and manage applications. SODALITE runtime currently supports five key types of execution

[1] https://github.com/xlab-si/xopera-opera

platforms: Edge (Kubernetes), private Cloud (OpenStack and Kubernetes), public Cloud (AWS), federated Cloud (EGI OpenStack), and HPC (TORQUEand SLURM).

- **IaC-based Orchestration Layer** is responsible for acquiring, allocating, and configuring resources from the execution platforms, and deploying and configuring application components using those resources. SODALITE uses Ansible as the IaC tool. Ansible playbooks realize the lifecycle operations for nodes/relationships in a deployment model in TOSCA.
- **Image Registry** stores container images. It can be a private or public repository, for example, Docker Hub or Google Container Registry. The private repositories should provide REST APIs to pull the images through IaC.
- **Authentication and Authorization Manager** handles the user and secrets management across the whole SODALITE stack. It applies role-based access control and token-based authentication. Each TOSCA blueprint and deployment is associated with a project domain with specific roles, an access type to which requires a token with specific JWT (JSON Web Token) claims. Each critical orchestration operation such as deployment, undeployment, deployment updates can only be performed by providing a valid access token. The implementation uses the Keycloak[2] identity and access management solution.
- **Application Data Manager** incorporates various transfer protocols and endpoints to achieve transparent data management across multiple infrastructure providers. Section 5.3 discusses data management capabilities in detail.

5.2.2 Orchestration APIs

The *Orchestrator* exposes its capabilities as RESTFul APIs.

- Blueprint Management. The blueprints (CASR files) can be added, removed, updated, and queried. The blueprints can also be accessed and managed through Git user accounts. Blueprint metadata such as id, version, and name can be obtained.
- Deployment Management. The applications can be deployed, and undeployed based on their blueprints. The deployment status and history can be obtained, and the failed deployments can be resumed. Application redeployment is requested by submitting the new version of the application deployment model. The *Orchestrator* will calculate the difference between the deployed instance and the new blueprint and will (un)deploy it. The new blueprint can be another version of the previously used blueprint or some version of another blueprint.
- Blueprint Inspection. The *Orchestrator* can also validate the syntax of TOSCA blueprints based on the version v1.3 of TOSCA Simple YAML Profile specification. It also supports calculating the differences between the current deployment state and a new blueprint.

[2] https://www.keycloak.org/

5.3 Managing Application Data

The components of heterogeneous applications are deployed across various execution platforms to utilize the capabilities of the different platforms. As such, one component can use HPC resources for better performance of batch computation, while another Cloud resources for better scalability and elasticity. Furthermore, this is also a possibility of processing on Edge devices. Using such a hybrid setup, where dependent components of the applications are deployed across various platforms, might require data transfers from one platform into another, and the orchestration system must support them. In this section, we explore the possibilities of data transfers between application components deployed across multiple infrastructure targets.

The current data management services found in scientific communities (e.g. FTS3[3], Rucio[4], DynaFed[5], OneData[6]) mostly focus on HPC and Cloud storage platforms and do not cover Edge, IoT and serverless platforms. Inversely, Edge and serverless platforms (e.g. MQTT[7], Apache Kafka[8], Fledge[9], Apache NiFi[10] and StreamSets[11]) target stream and Cloud storage platforms, but do not target HPC platforms. Therefore, in order to meet the SODALITE objectives for supporting heterogeneous infrastructures, data management for mentioned platforms shall be provided.

The RADON project [12] has developed a set of standard TOSCA libraries[13] for lifecycle management of data pipelines, which is inline with IaC-based orchestration in SODALITE. The concept of data pipeline allows composition of application components (e.g. microservices, serverless functions or self-contained components) as independently deployable and scalable pipeline tasks with the data movement and possible data transformation between the components [4]. As an underlying technology for data pipelines, Apache NiFi service is used. It exposes a REST API for data flow management between pipeline elements (blocks) and also provides connectors to various platforms and storage systems, such as S3, GCS, Azure, Apache Kafka, HDFS, MQTT, HTTP, (S)FTP, etc. This enables fetching data from one storage provider and pushing data to another provider as a pipeline task. As part of the collaboration with RADON project, we studied the feasibility of using data pipelines as components to unify data management between various heterogeneous platforms. SODALITE extended RADON's TOSCA and IaC libraries for data pipeline man-

[3] https://github.com/cern-fts/fts3

[4] https://github.com/rucio/rucio

[5] https://lcgdm.web.cern.ch/dynafed-dynamic-federation-project

[6] https://github.com/onedata/onedata

[7] https://mqtt.org/

[8] https://github.com/apache/kafka

[9] https://github.com/fledge-iot/fledge

[10] https://github.com/apache/nifi

[11] https://github.com/streamsets/datacollector

[12] https://radon-h2020.eu/

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

agement, which currently target multi-cloud storage and serverless platforms, with GridFTP support – a common file transfer protocol used in HPC. This enables an interoperability between HPC, Cloud storage types and data streams. Extended TOSCA libraries can be found in the joint RADON-SODALITE organization[14].

Figure 5.3 depicts an overview on data pipeline management. The Orchestrator exposes a REST API for deployment of a CSAR, which contains a TOSCA application topology description along with TOSCA node types, IaC and other dependencies. The application topology must specify pipeline blocks and connections between them, as well as specify which NiFi instance to use and whether the orchestrator must create a new instance or use the existing instance of NiFi. On the lower level, to instantiate a pipeline block, the orchestrator uses NiFi REST API to upload a NiFi XML template that describes the pipeline block. NiFi then registers the template and returns the ID of the pipeline, which in turn is used by the Orchestrator to request NiFi for the pipeline execution. Same happens for every pipeline block in the application topology. At this point, the registered pipeline blocks are established and functional, and the Orchestrator relies on NiFi instances to perform data movements between the pipeline blocks or move data to a certain storage system as a pipeline task.

A PipelineBlock [3], depicted in Figure 5.4, is an entity that executes pipeline tasks, such as data processing, API calls invocation, fetching data from or pushing data to remote storage systems or stream platforms, etc. The PipelineBlock may contain input (DataIngestionQueue) and output (DataEmissionQueue) queues for buffering input and resultant data. Using these queues, multiple PipelineBlocks can be connected sequentially, forming a group of PipelineBlocks. Similarly, multiple groups can also be connected using InputPipes - gateways for receiving input data from the previous group or external data source, and OutputPipes - for forwarding resultant data to the next group or external data sink.

The IaC data management features are limited to the capabilities and functionalities offered by Apache NiFi. We mainly focus on utilising NiFi for multi-protocol and multi-platform data movement, abstracted in TOSCA and IaC. Current structure of featured TOSCA node types for data pipeline blocks is presented in Figure 5.5, and they can be categorised into four classes of pipeline blocks and can be extended:

1. Source pipeline blocks - for consuming data from a data endpoint (e.g. HTTP, FTP, S3, GCS, Kafka, MQTT, GridFTP).
2. Destination pipeline blocks - for publishing data to a data endpoint (e.g. HTTP, FTP, S3, GCS, Kafka, MQTT, GridFTP).
3. Midway pipeline blocks - for executing data processing tasks (e.g. local data processing, encryption, invoke serverless FaaS).
4. Standalone pipeline blocks - for performing independent activities (e.g. copy from S3 to S3).

Listing 5.1 depicts an example of two data pipeline blocks that allow the data transfer between GridFTP server and an S3 bucket. *PubsS3Bucket* is a data pipeline

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

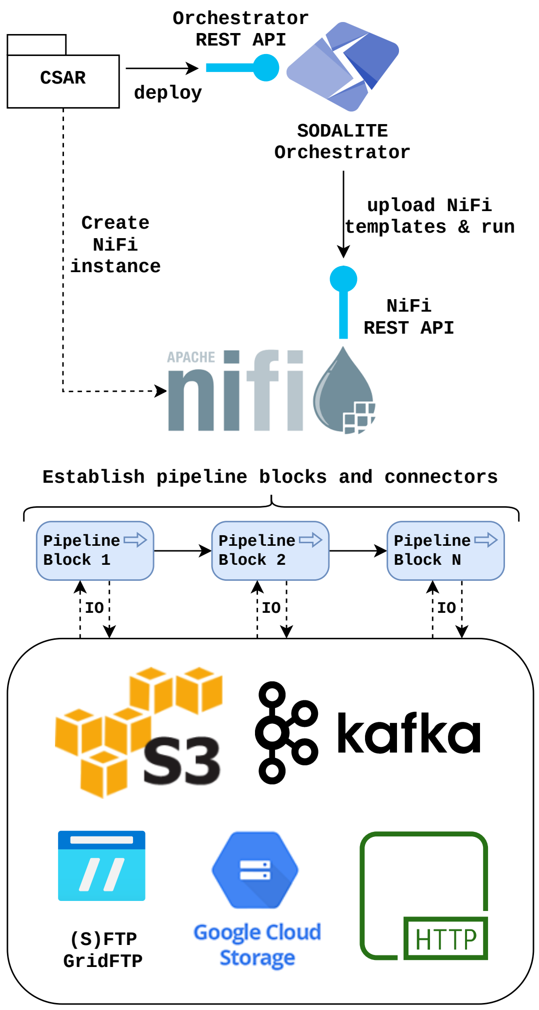


Fig. 5.3: IaC data pipeline management architecture

block that publishes data to an S3 bucket, whereas *ConsumeGridFtp* consumes data from a GridFTP server. The *ConsumeGridFtp* pipeline block has a *connectToPipeline* requirement, which points to the *PubsS3Bucket* pipeline block, therefore connecting these data pipeline blocks and performing data transfers between GridFTP and S3.

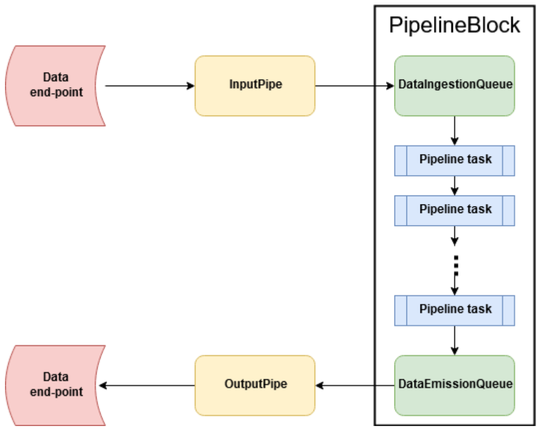


Fig. 5.4: Architecture of data pipeline block [3]

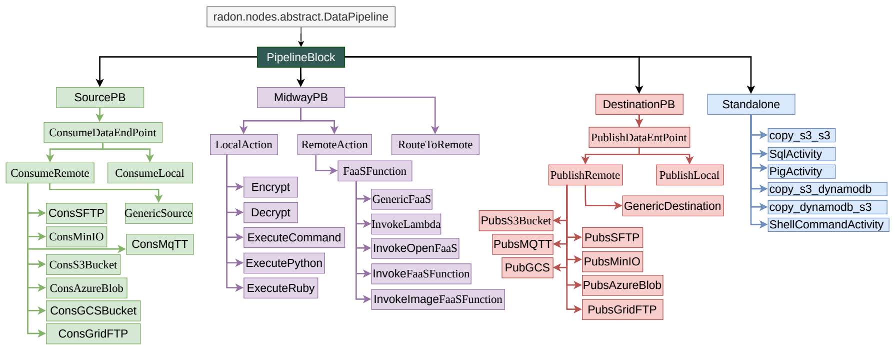


Fig. 5.5: A hierarchy of featured TOSCA node types

```
PubsS3Bucket:
    type: radon.nodes.datapipeline.destination.PubsS3Bucket
    properties:
        BucketName: "gridftp-result-bucket"
        cred_file_path: "/home/user/.aws/nifi_credentials"
        schedulingStrategy: "EVENT_DRIVEN"
        schedulingPeriodCRON: "* * * * * ?"
        name: "sendToS3"
        Region: "eu-central-1"
    requirements:
        - host: NiFi

ConsumeGridFtp:
    type: radon.nodes.datapipeline.source.ConsumeGridFtp
    properties:
        gridftp_port: 2811
        intermediate_folder: "/tmp/nifi_gridftp_subscribe/"
        schedulingStrategy: "EVENT_DRIVEN"
        schedulingPeriodCRON: "* * * * * ?"
        name: "receieveFromGFTP"
        gridftp_user: "user"
        gridftp_host: "gridftp.server.example.com"
        gridftp_cert_path: "/home/user/.globus"
        gridftp_directory: "~/target_dir/"
    requirements:
        - host: NiFi
        - connectToPipeline:
            node: PubsS3Bucket
            relationship: con_ConnectNifiLocal
            capability: ConnectToPipeline
```

List. 5.1: Snippet of TOSCA node template with S3 publisher and GridFTP consumer that allow data transfer from GridFTP server to an S3 bucket

5.4 Monitoring Applications and Infrastructures

The deployed application is continuously monitored, allowing the user to consult the state of the deployment as well as making the data available to other components such as *Deployment Refactorer*. The main requirements the monitoring system must meet are:

1. Dynamic addition and deletion of monitored components
2. Monitoring of different levels (infrastructure, runtime environment and application)
3. Transparency to the user
4. Possibility to add alerting rules for specific components
5. Access to metrics filtered by deployment, so that they are only available to the deployment owner.

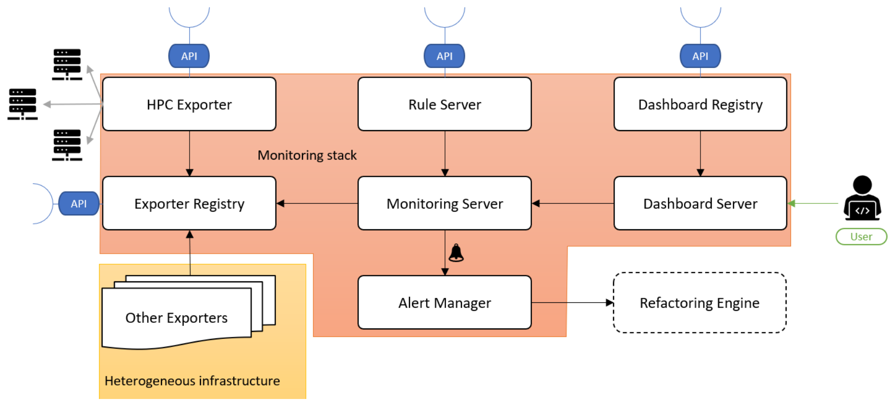


Fig. 5.6: The architecture of the *SODALITE@RT* monitoring system

We designed the monitoring system to meet these requirements (see Figure 5.6). The APIs that appear on the figure are the ones used by components outside of the monitoring system. The system is composed of the following elements:

- Monitoring server: It collects metrics from the exporters and saves them, exposing a service to other components to get monitoring data. It also detects alerting rule violations and triggers alerts.
- Alert manager: When an alert is triggered by the monitoring server, it sends the alert to the subscribed services, like the *Deployment Refactorer*.
- Rule server: Allows the dynamic creation of alerting rules, registering them in the monitoring server.
- Exporter registry: Allows the dynamic registration and deregistration of exporters in the monitoring server.

- Dashboard server: Stores dashboards and makes them available to users. The dashboards aggregate the metrics stored in the monitoring server and present them in a meaningful way. There are different types of dashboards depending on the type of exporter they aggregate the metrics from.
- Dashboard registry: It allows the dynamic creation of dashboards in the dashboard server, also manages the user permissions to access each dashboard so that each user can only view the metrics belonging to their deployments.
- Exporters: They collect the metrics from the monitored resources and expose them to the monitoring server. There are different types depending on which resource they monitor.

The rest of this section will discuss these components and their implementations in detail. carries out, as well as some details on their implementation.

5.4.1 Exporters

The main task of exporters is probing a resource to extract data, treat this data if necessary to convert it to meaningful metrics, and expose these for the monitoring service. There are different types of exporters depending on the resource they can extract metrics from.

Exporters are made up of different collectors, each of them collecting a different type of metrics. For example, to expose a VM's OS metrics the node exporter is used, which, among others, has the CPU, netstat, and file-system collectors to monitor CPU usage, file-system status, and operating system's network statistics respectively.

In SODALITE there are 5 exporter types in use:

- Node exporter: Extracts a machine's OS metrics, such as CPU and RAM usage, context swaps, and file-system stats. One of them is deployed on each virtual machine and edge node.
- Skydive exporter: Exposes the infrastructure's network statistics such as network flow and traffic metrics[15]
- HPC Exporter: To monitor the jobs submitted to an HPC as well as the HPC's infrastructure status (available nodes, queue status, etc).
- Edge exporter: It contains accelerator-specific collectors for any attached heterogeneous accelerators (e.g., Edge TPU and GPU). They provide specific insight into the attached accelerators. This may include aspects such as the number of devices available, the load average, or thermal properties.

The Ansible playbooks that are responsible for setting up nodes also deploy the exporters associated with each node. The configuration parameters for exporters can be provided using TOSCA node properties. Listing 5.2 shows a snippet of an Ansible playbook that installs the EdgeTPU exporters into the edge nodes in a Kubernetes cluster. It uses the Ansible modules for executing the relevant Helm charts. By having

[15] http://skydive.network/

the exporter creation and registration in the exporter registry in the standard Ansible playbooks, these actions are transparent to the user, who only needs to provide high-level settings for the exporters.

```
tasks:
-name: Add Prometheus Community repository
    community.kubernetes.helm_repository:
        name: prometheus-community
        repo_url: https://prometheus-community.github.io/helm-charts
-name: Add Adaptant repository
    community.kubernetes.helm_repository:
        name: adaptant
        repo_url: https://adaptant-labs.github.io/charts/adaptant
-name: Install EdgeTPU exporter
    community.kubernetes.helm:
        name: edgetpu-exporter
        chart_ref: adaptant/edgetpu-exporter
        release_namespace: "{{namespace}}"
```

List. 5.2: Snippet of an Ansible playbook for installing the EdgeTPU exporter.

5.4.1.1 HPC exporter

The HPC exporter is a special case. It connects to an HPC front-end through SSH and runs commands to gather information about the queues, node status, and job statistics. In order to carry out this task it needs to have the user's SSH credentials, it also must be deployed on a different machine than the HPC front-end itself, since it needs to use port 9110 to expose the metrics to the monitoring server, and most HPC's would not allow arbitrary ports to be open for security reasons. In order to solve this issue, there is only a single HPC exporter deployed alongside the rest of the monitoring system's core components, as part of SODALITE's back-end. The exporter, aside from exposing an endpoint for the monitoring server to collect metrics, also exposes 3 more endpoints, which form an API:

- /create: It accepts a JSON object containing the necessary configuration to monitor a given HPC front-end. It creates a collector in the HPC exporter and registers its association with the deployment ID and user that created it. All the metrics exposed by this collector will include labels to identify which deployment it is a part of.
- /delete: Removes the selected collector
- /addJob: It accepts a JSON object which contains a deployment ID and a Job ID, and adds the Job ID to the collector's list of jobs to monitor.

All the calls to this API are secured with the same JWT system used across SODALITE to ensure that only authorized users can create, delete and modify collectors and that users can only delete and modify their own collectors. This system is also used to retrieve the SSH credentials needed to connect to the HPC's front-end from the secrets vault. This way no SSH credentials need to travel unsecured as part of the configuration settings.

5.4.2 Monitoring server

The monitoring server is the central piece of the monitoring system. Its job is to collect the metrics from all the registered exporters and store them in a database, in a time-series format. This component offers an API so that other components can query data. These queries are done in the server's own query language, which allows for data aggregation and filtering. This API is mainly used by the Dashboard server since in SODALITE, users do not have direct access to this monitoring server. This is due to security considerations since users should only have access to metrics coming from their own deployments' components.

The monitoring server also allows the definition of alerting rules, which consist of:

- An condition written in the monitoring server's query language that, when met for a certain amount of time, triggers an alert.
- For how long the condition must be met so that the alert is triggered.
- The severity of the alert
- The contents of the alert that will be sent to the Alert Manager when the alert is triggered. This may include information about the instance that generated this alert, or other context information.

An example of an alert that triggers when a VM's CPU usage has exceeded 75% (within the deployment with monitoring ID `7acf2a5-da51s4da-as44d1c1a8ftr`) is shown in Listing 5.3. When an alert is triggered, the monitoring server sends it to the Alert Manager, which is the component tasked with distributing the alerts to the subscribed services. The main service that consumes the alerts is the Refactoring Engine, which uses the alerts to trigger refactoring actions based on the content of such alerts.

```yaml
alert: HighCPULoad
expr: 1- (avg by(instance,os_id) (irate(node_cpu_seconds_total{mode="idle",
↪  monitoring_id="7acf2a5-da51s4da-as44d1c1a8ftr"}[5m])))> 0.75
for: 5m
labels:
  severity: warning
annotations:
  summary: 'CPU load above 75% load (instance {{ $labels.instance }})'
  description: 'CPU load is > 75%\n  VALUE = {{ $value }}\n  Monitoring ID: {{
  ↪  $labels.monitoring_id }}\n INSTANCE: {{ $labels.instance }}'
```

List. 5.3: An alerting rule for indicating high CPU usage in a node.

Rules cannot be dynamically added to the monitoring server by default. That is why the rule server exists. It consists of an API that allows the registration and removal of alerting rules on the monitoring server. The IDE has a module that allows users to create, edit, and add alerting rules to deployments they own through this component (see section 3.2.5).

5.4.3 Exporter registry

Much like with alerting rules, the monitoring server does not allow for the dynamic registration of exporters. All the endpoints it scrapes metrics from must be known at the time of deployment. In order to allow dynamic creation of resources during the lifetime of the platform, a key aspect of SODALITE, the exporter registry is used.

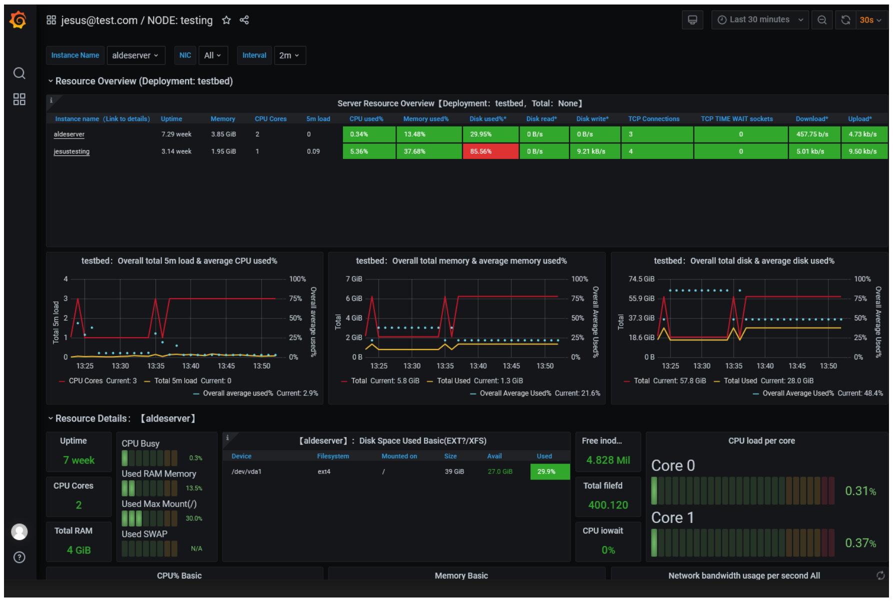
The targets for the monitoring server are actually endpoints offered by the exporter registry. There is an endpoint for each type of exporter. When a new exporter is deployed (for example, when a new VM is created as part of an application deployment), the address of the new exporter is registered with the exporter registry, which ensures that when the monitoring server scrapes metrics, it also scrapes from all its registered exporters.

5.4.4 Dashboard server

The users see all the metrics collected by the monitoring system on a dashboard that has been created for each deployment and exporter type. Each dashboard can be accessed on a web browser and is a set of graphs and indicators that aggregate the metrics from the corresponding Monitoring ID and exporter type. The component that hosts the dashboards is the dashboard server, which also takes care of authenticating users to make sure only users that have access to a Monitoring ID can view its dashboards.

An example of a dashboard for node exporters is shown in Figure 5.7, it includes a list of the VMs that are part of the dashboard's deployment and allows the user to select one of them to view detailed metrics such as CPU, memory, or disk usage graphs, as well as other indicators like the number of context swaps.

To ensure that a dashboard only contains information from a certain deployment, the Monitoring ID the dashboard belongs to is hard-coded when it is created, which means that there must be a way to create dashboards dynamically, a task fulfilled by the dashboard registry.


jesus@test.com / NODE: testing
Last 30 minutes
30s
Resource Overview (Deployment: testbed)
testbed: Overall total 5m load & average CPU used%
testbed: Overall total memory & average memory used%
testbed: Overall total disk & average disk used%
CPU Cores Current: 3
Total 5m load Current: 0
Overall average used% Current: 2.9%
Total Current: 5.8 GiB
Total Used Current: 1.3 GiB
Overall Average Used% Current: 21.6%
Total Current: 57.8 GiB
Total Used Current: 28.0 GiB
Overall Average Used% Current: 48.4%
Resource Details: 【aldeserver】
CPU% Basic
Memory Basic
Network bandwidth usage per second All

Instance Name: aldeserver   NIC: All   Interval: 2m

Server Resource Overview【Deployment: testbed, Total: None】

Instance name (Link to details)	Uptime	Memory	CPU Cores	5m load	CPU used%	Memory used%	Disk used%*	Disk read*	Disk write*	TCP Connections	TCP TIME WAIT sockets	Download*	Upload*
aldeserver	7.29 week	3.85 GiB	2	0	0.34%	13.48%	29.95%	0 B/s	0 B/s	3	0	457.75 b/s	4.73 kb/s
jesustesting	3.14 week	1.95 GiB	1	0.09	5.36%	37.68%	85.56%	0 B/s	9.21 kB/s	4	0	5.01 kb/s	9.50 kb/s

Uptime	7 week
CPU Cores	2
Total RAM	4 GiB
CPU Busy	0.3%
Used RAM Memory	13.5%
Used Max Mount(/)	30.0%
Used SWAP	N/A
Free inod...	4.828 Mil
Total filefd	400.120
CPU iowait	0%

【aldeserver】: Disk Space Used Basic(EXT?/XFS)

Device	Filesystem	Mounted on	Size	Avail	Used
/dev/vda1	ext4	/	39 GiB	27.0 GiB	29.9%

CPU load per core

Core 0	0.31%
Core 1	0.37%

Fig. 5.7: Example of dashboard showing metrics collected by node exporters.

5.4.5 Dashboard registry

The dashboard registry consists of an API that exposes a number of endpoints, all secured by the JWT generated by Keycloak when the user logs in the IDE:

- /dashboard (POST): The registry creates one dashboard per exporter type from a set of templates for the required monitoring ID on the dashboard server. It also sets the dashboard permissions so that only the user that made this call is able to view these dashboards.
- /dashboard (DELETE): Deletes the dashboards that belong to the provided monitoring ID, if the user has the access to them.
- /dashboard/user (GET): Returns the URL of all the user's dashboards
- /dashboard/deployment/<monitoring_id> (GET): Returns the URL of the dashboards that belong to the given monitoring ID, only if the user has access to them.

By using this system we can ensure the security of the monitoring system and at the same time allow for great scalability and flexibility, key values of SODALITE.

5.5 Adapting Application Deployments

In response to the data collected and events received from *Monitoring System*, *Deployment Refactorer* decides and carries out the desired changes to the current deployment of a given application. In this section, we present the architecture and the key capabilities of *Deployment Refactorer*.

5.5.1 Architecture of Deployment Refactorer

Figure 5.8 shows the architecture of the *Deployment Refactorer*. The overall deployment adaptation logic can be codified as an ECA (Event-Condition-Action) policy. Policy Engine can enact and manage such policies. In order to build complex policies, *Deployment Refactorer* provides a set of utilities: *Workload Predictor*, *Performance Predictor*, *Deployment Configuration Selector*, and *Performance Anomaly Detector*. *Workload Predictor* uses linear and polynomial regression models to forecast the workload (the number of requests for the next period). Given the predicted workload and the deployment options used, *Performance Predictor* can predict the performance metrics. If the current deployment model variant cannot meet the performance goals, *Deployment Configuration Selector* can be used to find an alternative deployment model from the allowed set of deployment model variants (expressed in the deployment variability model). *Performance Anomaly Detector* can be used to continuously monitor the current deployment for anomaly behaviors, and generate alerts. The predictive ML models used by each of these components are stored in the *Predictive Model Repository*. The features used by such models are stored in the *Feature Store*. In the rest of this section, the support for the key capabilities of Deployment Refactorer is presented.

5.5.2 Policy-based Deployment Adaptation

To allow a software engineer to define the deployment adaptation decisions, we provide an ECA (event-condition-action) based policy language. Figure 5.9 the key concepts of the policy language. A policy consists of a set of ECA rules.

- **Events and Conditions**. A *condition* of a rule is a logical expression of events. We consider two common types of events pertaining to the deployment model instance of an application: deployment state changes and application and resource metrics. The application and resource metric events include (raw or aggregated) primitive metrics collected from the running deployment, for example, average CPU load, as well as alerts or complex events that represent predicates over primitive metrics, for example, the above-mentioned *HostHighCPULoad* alert.

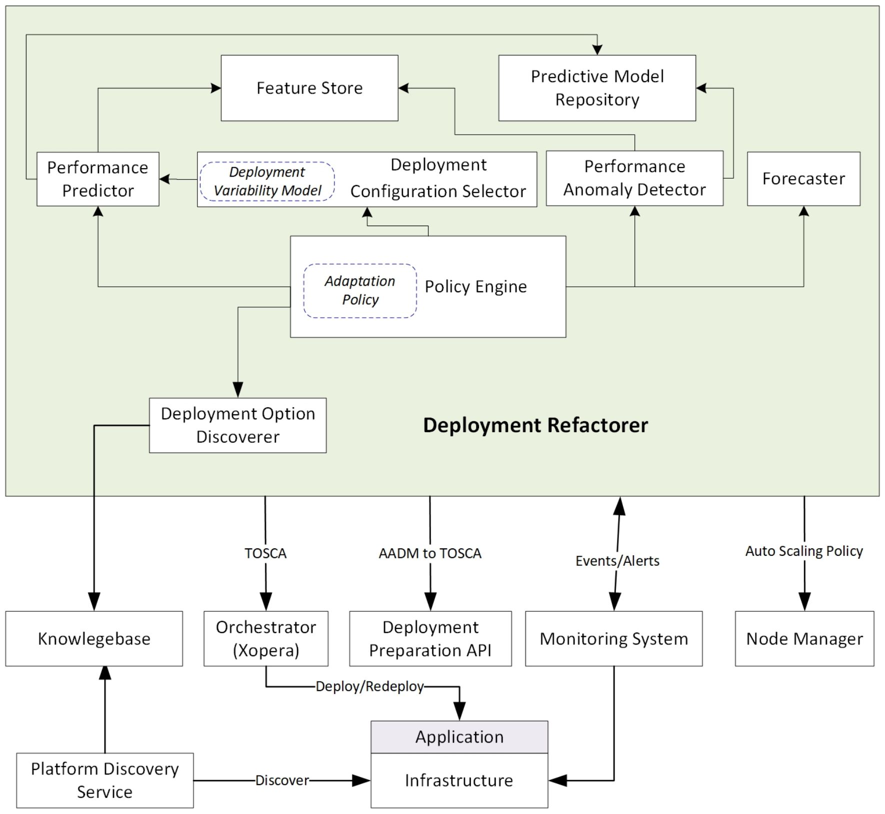


Fig. 5.8: Architecture of Deployment Refactorer

- **Actions**. The actions primarily include the common change operations (*Add*, *Remove*, and *Update*) and the common search operations (*Find* and *EvalPredicate*) on nodes, relations, and their properties. Additionally, the custom actions can be implemented and then used in the deployment adaptation rules, for example, actions for predicting the performance of a particular deployment model instance or predicting workload. To ensure the safe and consistent changes to the deployment model instance, *Deployment Refactorer* makes the change operations to a local representation (a Java Object model) of the deployment model (represented using the concept of models@runtime [2]). Once the adaptation rules in a rule session are executed, *Deployment Refactorer* translates the current local object model to a TOSCA file and calls the update API operation of the Orchestrator with the generated file.

The *Deployment Refactorer* uses a policy engine to enact the deployment adaptation policies. It supports the addition, removal, and update of policies. It can parse given policies, process events, and execute the policies. The policy rules are triggered as their conditions are satisfied, and the desired changes are propagated to the deployment model instance.

Listing 5.1 shows an example of a deployment adaptation rule that reacts to the events *HostLowCpuLoad* and *HostHighCpuLoad* by moving the snow application between a medium VM and a large VM.

5.5.3 Data-driven Deployment Switching

We use a machine learning-based approach to implement deployment switching. In particular, we use a performance model that can predict the performance of a given deployment alternative in terms of deployment options used by the variant. The deployment options represent architectural and resource selection decisions that are made by the experts when creating deployment models, for example, inclusion or exclusion of a web cache, use of a cluster mode, and use of a large VM or small VM. The initial performance models are built offline, and at runtime, based on the monitored data, the models are retrained as necessary, for example, if the model accuracy drops below a predefined threshold. Figure 5.10 shows the design time and runtime workflows of our deployment switching approach.

We first model the allowed set of deployment variants for a given application based on the deployment decisions, their instantiations, and their inter-dependencies. Based on this deployment variability model, we select an initial valid sample of deployment variants and measure the performance of each variant in the sample. We use the measured application performance dataset to train a predictive model and then evaluate its performance. If the model prediction accuracy is unacceptable, the

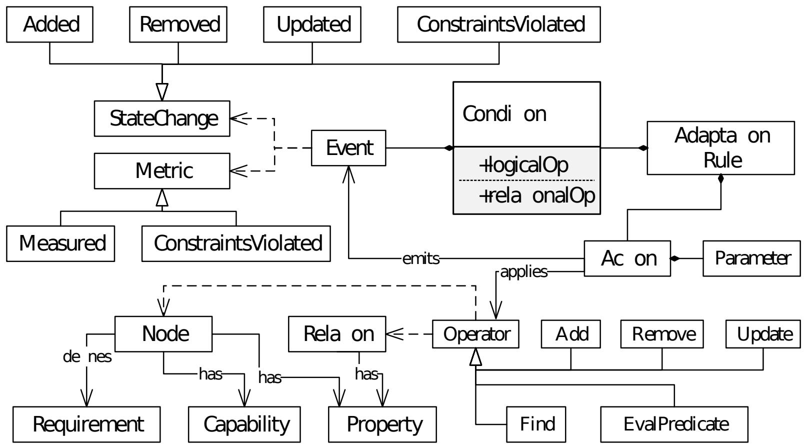


Fig. 5.9: Meta-model of the deployment adaptation policy language

```
  rule "HostHighCpuLoad"
  when
    $f1 : Alert(name == "HostHighCpuLoad")
  then
    Node snowvm2node = refMgt.findMatchingNodeFromRM("( ?name =
        \"snow-vm_new_2\" )");
    AADMModel aadmModel = refMgt.getAadm();
    aadmModel.addNode(snowvm2node);
    List<Node> nodes = refMgt.getNodeMatchingReqFromRM("snow/snow
        -vm-2");
      for (Node node : nodes) {
            aadmModel.addNode(node);
      }
    aadmModel.updateProperty("snow-skyline-extractor", "ports", "
        8080:8080");
    aadmModel.updateRequirement("snow-skyline-extractor", "host",
         "snow-docker-host-2");
    ...
    refMgt.saveAndUpdate();
end

rule "HostLowCpuLoad"
  when
    $f1 : Alert(name == "HostLowCpuLoad")
  then
    AADMModel aadmModel = refMgt.getAadm();
    aadmModel.removeNode("snow-vm-2");
    aadmModel.removeNode("snow-docker-host-2");
    aadmModel.removeNode("snow-docker-registry-certificate-2");
    aadmModel.updateProperty("snow-skyline-extractor", "ports", "
        8082:8080");
    aadmModel.updateRequirement("snow-skyline-extractor", "host",
         "snow-docker-host");
    ...
    refMgt.saveAndUpdate();
end
```

List 5.1: A snippet of a deployment adaptation rule

performance of an additional sample of deployment variants is measured and used to retrain the model.

To model the allowed variations in the deployment topology of an application, we use the feature modeling technique, which is a widely-used variability modeling technique [1], and is also supported by open source and commercial tools. We used FeatureIDE [16], which is an Eclipse plugin that can be installed into the SODALITE IDE. A feature model can represent the commonalities and variations in a family of artifact variants as configuration options and their inter-dependencies and other constraints. An artifact can be a software system, application, design model, and

[16] https://featureide.github.io/

more. By respecting all the constraints defined by the feature model, we can select a subset of configuration options (called a feature configuration), which represents a valid artifact variant or a family member. The feature leaf nodes can represent the component deployment options, which are the unique assignments of application components to VMs. Similarly, the feature group nodes (non-leaf nodes) and their hierarchical organization capture the logical decomposition of the deployment decisions. For example, the web server and the database cache can be deployed together (co-deployment) or separately (separate-deployment), which can be modeled using an XOR feature group.

To sample a variability model (i.e., to select a subset of deployment model variants), there exist many sampling strategies proposed by the research literature in the performance modeling of configurable systems. In our current implementation, we experimented with three sampling techniques: random sampling, T-wise sampling, and dissimilarity sampling.

To collect data for offline training of models, we used the benchmarking approach due to our preference for the accuracy of the performance data. For each deployment variant in the sample, we select the component deployment options, create them in the target environment, subject the application to a range of workloads using a load testing tool, and collect the performance metrics (response time) per workload.

To build the predictive models, the current literature in configurable systems has used many different learning algorithms, including traditional machine learning algorithms as well as deep learning models. In our current implementation, we used the following three models: Decision Tree Regression (DTR), Random Forest Regression (RFR), Multilayer Perceptron Neural Network (MLP). The performance prediction model is used at runtime to predict the performance of a given deployment variant for a given workload. If the current deployment model cannot satisfy the performance goals, then, a deployment model variant that can meet the performance goals is selected. For more information, we refer the readers to the relevant publication at [5].

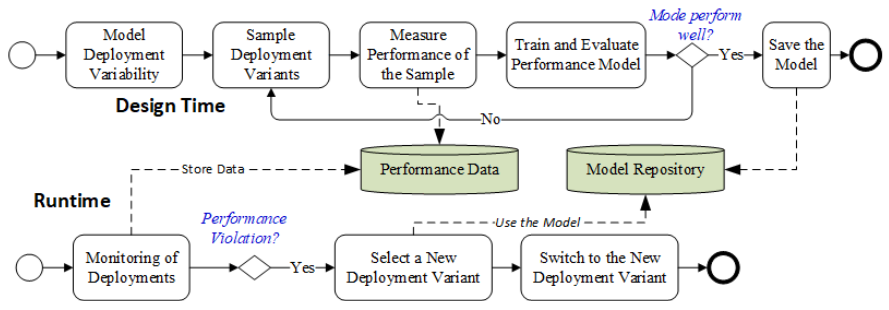

Model Deployment Variability
Sample Deployment Variants
Measure Performance of the Sample
Train and Evaluate Performance Model
Mode perform well?
Yes
Save the Model
Design Time
No
Store Data
Performance Data
Model Repository
Runtime
Performance Violation?
Use the Model
Monitoring of Deployments
Yes
Select a New Deployment Variant
Switch to the New Deployment Variant

Fig. 5.10: Workflows of building and using predictive models for making deployment switching decisions

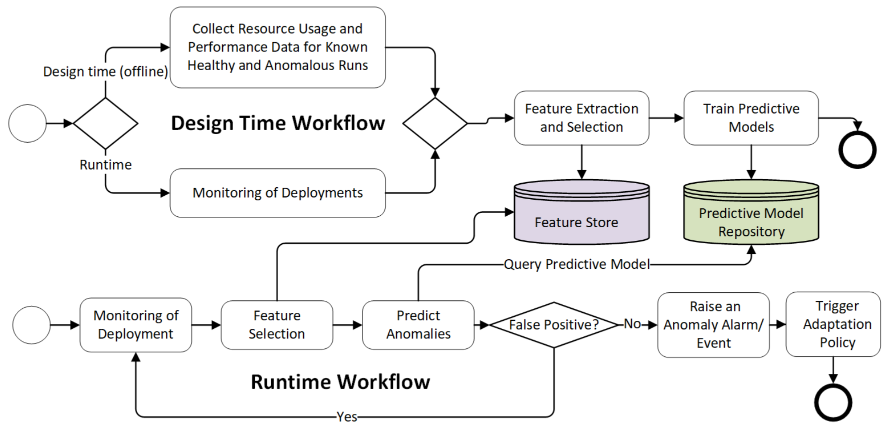

Design time (offline)
Collect Resource Usage and Performance Data for Known Healthy and Anomalous Runs
Design Time Workflow
Runtime
Monitoring of Deployments
Feature Extraction and Selection
Train Predictive Models
Feature Store
Predictive Model Repository
Query Predictive Model
Monitoring of Deployment
Feature Selection
Predict Anomalies
False Positive?
No
Raise an Anomaly Alarm/ Event
Trigger Adaptation Policy
Runtime Workflow
Yes

Fig. 5.11: An Overview of our Anomaly Detection Approach : (a) Training Workflow (Offline and Runtime), (b) Prediction Workflow (Runtime)

5.5.4 Data-driven Anomaly Detection

An anomaly can be defined as a rare event where the system behavior deviates from what is standard, normal, or expected. For example, a service could use an anomalous amount of resources, or the service network exhibits traffic anomalies. The ability to detect anomalies and trigger corrective actions is critical to maintain the quality of service and to prevent runtime service failures and undue usage of resources. In the SODALITE project, we consider the anomalies as Chaos that can occur in a containerized microservice system, for example, *CPU Hog*, *Pod Delete*, and *Pod Network Corruption.*

We aim to detect whether a compute node or a cluster is anomalous and classify the type of the anomaly at runtime, independent of the microservice application that is running on the compute node or the cluster. To detect and classify anomalies, we use a machine learning-based approach (see Figure 5.11). We first build the machine learning models at the design time by utilizing historical resource usage and performance data that are collected from healthy and anomalous situations. Then, at runtime, we apply these models to the monitoring data from the application deployment to detect anomalies. The monitoring data is also used to update and adapt the models. By utilizing various capabilities of SODALITE (e.g., monitoring and alerting, platform and resource discovery, policy-based adaptation, and redeployment), the runtime detection and correction of anomaly behaviors are supported. We used three types of machine learning algorithms to build anomaly predictors: Decision Tree, Random Forest, and AdaBoost. All three models were able to predict anomalies with at least 97% accuracy.

```
select DISTINCT ?node ?description ?nodetype
where {
        ?nodetype rdfs:subClassOf tosca:tosca.nodes.Compute .
        ?node rdf:type ?nodetype .
        OPTIONAL {?node dcterms:description ?description .}
        FILTER (?nodetype != tosca:tosca.nodes.Compute ) .
        FILTER (?node != owl:Nothing) .
        ?node soda:hasContext ?context .
    {
    ?context tosca:properties ?concept .
    OPTIONAL {?concept DUL:classifies snow:flavor .}
    OPTIONAL {?concept tosca:hasDataValue ?flavor .}
    }
    {?context tosca:properties ?concept1 .
        OPTIONAL {?concept1 DUL:classifies snow:image .}
        OPTIONAL {?concept1 tosca:hasDataValue ?image .}
    }
        FILTER ( ( ?flavor = "m1.small" ) && ( ?image = "centos7" ))
}
```

List. 5.4: Snippet of the SPARQL Query Generated for Retrieving Nodes Matching the Constraint flavor = "m1.small" && image = "centos7"

5.5.5 TOSCA Compliant Refactoring Option Discovery

The *Deployment Refactorer* uses refactoring options for adapting a given deployment model. A refactoring option represents one or more nodes in a deployment model. *Deployment Option Discoverer* uses semantic web technologies for discovering TOSCA-compliant resources and deployment model fragments or refactoring options. It considers constraints on node attributes, node requirements, node capabilities, and node policies. The semantic annotation of resource models including the attached policies enables machine reasoning which is then used for both the discovery and the composition of resources.

The *Deployment Option Discoverer* performs matchmaking by executing the SPARQL queries over the ontologies in the knowledgebase. It provides high-level system support to the *Deployment Refactorer* to allow searching for resources, for example, find (a logical expression over node properties). It has the SPARQL query templates for different types of resource matchmaking. The query templates are instantiated with the input data received through the high-level API operations. Listing 5.4 shows a snippet of the SPARQL Query generated for retrieving nodes matching the constraint flavor = "m1.small" && image = "centos7".

5.6 Vertical AutoScaling and Smart Scheduling

Component *Node Manager* aims to deploy and manage applications on existing resources deployed by the SODALITE users or by component *Deployment Refactorer*. In particular, *Node Manager* provides two main features that are complementary to

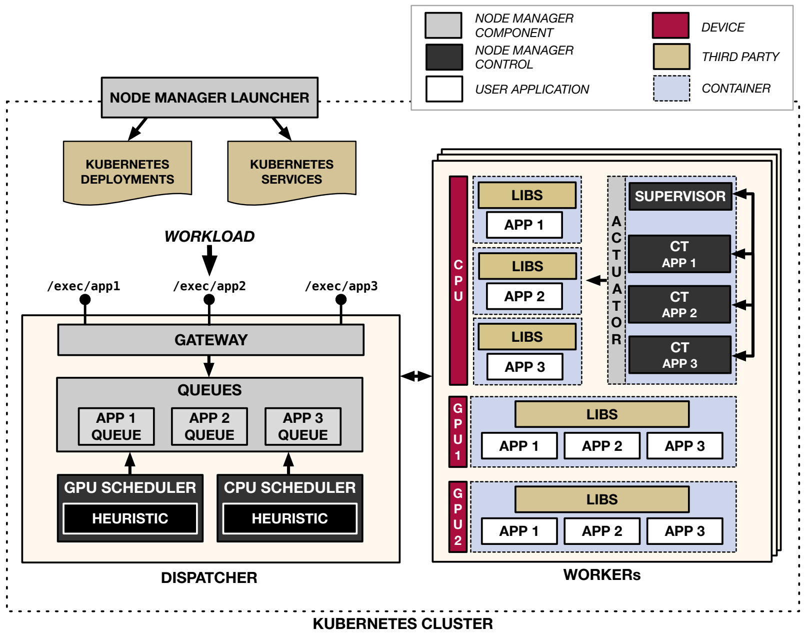


Fig. 5.12: Architecture of the *SODALITE* Node Manager

the ones of the Deployment Refactorer: i) it provides fast vertical scaling of computing and memory resources, and ii) it provides scheduling to select the proper executor (i.e., a CPU or a GPU) for a given request.

While the model of *Node Manager* is general enough to support different platforms, its current implementation is based on TensorFlow[17]. TensorFlow is one of the most used frameworks for developing and executing Machine Learning applications that can be run on both CPUs and GPUs. Specifically, *Node Manager* uses TensorFlow Serving, an image of the container provided by TensorFlow that allows ML applications to be run as Docker[18] containers.

The architecture of *Node Manager* is shown in Figure 5.12. *Node Manager* deploys containerized applications on a Kubernetes[19] cluster and provides two main types of node: a dispatcher and a set of worker nodes (Kubernetes nodes). The dispatcher receives the incoming requests from the users of the different apps and acts as a smart load balancer.

It first stores the requests in a dedicated queue and, according to the requirements (Service Level Agreement) of the applications and their performance, it schedules them for execution on a fast GPU or a CPU. In the workers, dedicated control theoretical planners (CT in the figure) vertically scale the CPU cores allocated to

[17] `https://www.tensorflow.org`
[18] https://www.docker.com
[19] https://kubernetes.io

each containerized application given the monitored performance of the system and the work done by the GPU. On each worker node, a *Supervisor* is also deployed to manage resource contention among the different running applications.

As soon as a user submits a trained model, along with its SLA, *Node Manager Launcher* generates or updates required Kubernetes *deployments* and *services* to let the system deploy and manage the application containers.

To exploit the CPUs and GPUs of a node, each application is bound to a specific *device*. In particular, given *m* applications selected to be deployed onto a worker node, *Node Manager* provisions:

- *m* containers containing one model each, and binds them to the node's CPU(s)
- one container, containing all the apps, for each GPU
- one container that includes the control theoretical planners for all the models, the *Supervisor*, and one actuator implemented as a Kubernetes volume.

Since we assume that the worker depicted in Figure 5.12 comes with two GPUs, and it manages three applications, Node Manager deploys six containers in total.

The deployment and configuration of the *Node Manager* and of users' applications can be done using TOSCA blueprints.

In order to deploy applications, users of *Node Manager* must create an AADM similar to the example provided in Listing 5.5

Users must use a node template of type `sodalite.nodes.nodemanager.deploy` in order to be able to correctly interface with the deployed *Node Manager*. The type requires the definition of the following properties:

- `endpoint`: the endpoint to contact to deploy the applications, by default it is equal to `<Node Manager IP>:5000/deployment`
- `models`: an array that contains essential metadata of each application. In particular each element must specify:
  - the `name` of the application and its `version`.
  - the `sla` (maximum allowed response time for the application) and its nominal response time (`profiled_rt`). This value must be obtained by measuring on average the end-to-end latency of a request with an empty queue.
  - a tuning parameter of control theoretical planner (`alpha`). The higher its value, the faster the controller's responses to sudden changes in the performance. A too high value, could lead to oscillating resource allocation (default value is set to be 0.5).
  - a URL pointing to the TensorFlow model of the application (`tfs_model_url`)
  - the initial number of container replicas for the app (`initial_replicas`).
- `available_gpus`: the amount of gpus available on each machine
- `tfs_image`: the Docker image of the TensorFlow serving container (default is set to `tensorflow/serving:latest`).
- `k8s_api_configuration`: metadata regarding the Kubernetes clusters (clusters, users and contexts) as specified at `https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/`

```
inputs:
    node-manager-ip:
        type: string

 node_templates:
    node-manager-deploy:
      type: sodalite.nodes.nodemanager.deploy
      properties:
        endpoint: get_input: node-manager-ip
        models:
          - name: app1
            version: 1
            sla: 0.4
            alpha: 0.5
            profiled_rt: 0.15
            tfs_model_url: https://app1.com/archive/v1.tar.gz
            initial_replicas: 1
          - name: app2
            version: 1
            sla: 0.3
            alpha: 0.5
            profiled_rt: 0.15
            tfs_model_url: https://app2.com/archive/v1.tar.gz
            initial_replicas: 1
        available_gpus: 1
        tfs_image: tensorflow/serving:latest
        k8s_api_configuration:
          apiVersion: v1
          clusters:
            - cluster:
                certificate-authority: "/home/root/.minikube/ca.crt"
                server: http://32.21.111.161:8080
              name: minikube
          contexts:
            - context:
                cluster: minikube
                namespace: default
                user: minikube
              name: minikube
          current-context: minikube
          kind: Config
          preferences: {}
          users:
            - name: minikube
              user:
                client-certificate: "/home/root/.minikube/profiles/minikube/client.crt"
                client-key: "/home/root/.minikube/profiles/minikube/client.key"
```

List. 5.5: Deploying applications through *Node Manager*

5.7 Conclusion and Future Work

The SODALITE platform enables the deployment of complex applications on heterogeneous Cloud-Edge-HPC infrastructure. It supports the modeling of heterogeneous application deployments using the TOSCA open standard, deploying such applications based on created models, and monitoring and adapting application deployments. SODALITE runtime employs machine learning-based approaches to switching between different deployment variants and detecting performance anomalies. SODALITE runtime includes the distributed control-theoretical planners that can support vertical resource elasticity for containerized application components that use both CPU and GPU resources.

We will be conducting future work in two key directions. On the one hand, we will further develop the SODALITE *runtime* by incorporating new infrastructures such as Open FaaS and Google Cloud, and by completing the integration of the runtime layer within the overall SODALITE stack. On the other hand, the monitoring and deployment adaptation support will be extended for the Edge-to-Cloud continuum.

References

[1] Thorsten Berger et al. “A Survey of Variability Modeling in Industrial Practice”. In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS ’13. Pisa, Italy: Association for Computing Machinery, 2013. ISBN: 9781450315418. DOI: 10.1145/2430502.2430513. URL: https://doi.org/10.1145/2430502.2430513.

[2] Gordon Blair, Nelly Bencomo, and Robert B France. “Models@ run. time”. In: *Computer* 42.10 (2009), pp. 22–27.

[3] Chinmaya Dehury. *Data pipeline orchestration I*. Tech. rep. RADON consortium, RADON publications, 2019.

[4] Chinmaya Dehury et al. “Data Pipeline Architecture for Serverless Platform”. In: *European Conference on Software Architecture*. Springer. 2020, pp. 241–246.

[5] Indika Kumara et al. “FOCloud: Feature Model Guided Performance Prediction and Explanation for Deployment Configurable Cloud Applications”. In: *IEEE Transactions on Services Computing* (2022), pp. 1–1. DOI: 10.1109/TSC.2022.3142853.

Chapter 6
SODALITE in Context

Kamil Tokmakov and Indika Kumara

Abstract This chapter will look at the application and integration of the SODALITE toolkit across various platforms in the Cloud, Edge, and HPC domains, with a specific focus on OpenStack, AWS EC2, Kubernetes, and PBS Torque/Slurm. While Chapter 7 provides a deep dive into the operational environment of the individual use cases, this chapter will focus more on general application and deployment patterns of SODALITE alongside existing deployments, demonstrating ways in which SODALITE can be leveraged more generally by SODALITE users in these respective environments.

6.1 Introduction

This chapter describes a general usage of SODALITE platform and serves as a guide for SODALITE users to create, adapt or extend their respective SODALITE models in order to provide a support for resource provisioning, optimization and deployment of application components in HPC, Cloud and Edge infrastructures. The SODALITE users are categorized into three types: Application Ops Experts (AOE), Resource Experts (RE) and Quality Experts (QE). Each type is performing a certain workflow - a set of common activities associated with the user type.

As such, a RE models available infrastructure resources and provides implementation artifacts that manage a lifecycle of the resources: instantiation and tear down of resources. A QE provides optimization models that are used to select and configure optimal application runtime for specific infrastructure target. An AOE then develops

Kamil Tokmakov
High Performance Computing Center Stuttgart (HLRS), Germany e-mail: `kamil.tokmakov@hlrs.de`

Indika Kumara
Jheronimus Academy of Data Science, Tilburg University, The Netherlands e-mail: `i.p.k.weerasinghadewage@tilburguniversity.edu`

an abstract application deployment model that constitutes application topology and consists of the instances of the node types defined in the resource models, relationships between the instances (e.g. dependency or host relationships) and association of optimization models into the instances. Upon the deployment of the application, an AOE can then monitor its runtime.

For more information on SODALITE users and their workflows, one can refer to Chapter 2. Furthermore, Chapter 3 discusses on application, resource and optimization models, whereas deployment and monitoring are outlined in Chapter 5.

This chapter is organized as follows. Section 6.2 provides details on the actions to be executed by Resource Experts who are in charge to make available to the other roles models of the available resources. Section 6.3 focuses on the activity of Quality Experts. Section 6.4 presents the core work of Application Ops Experts (AOE). Finally, Section 6.5 concludes the chapter.

6.2 Resource Expert

The Resource Expert (RE) follows the workflow depicted in Figure 2.2. He/She develops Resource Models (RM) and implementation artifacts, such as Ansible playbooks, templates or configuration files, that will be performing the lifecycle operations over the infrastructure, middleware and other resources. Alternatively, the RE can use the Platform Discovery to automatically define the resources. The SODALITE IDE repository contains reference examples of resource models[1] and the iac-modules repository[2] contains their TOSCA counterparts and Ansible playbooks.

Typical resources in the cloud environment that the RE models are the following: (1) credentials to access cloud provider; (2) a keypair - an SSH public key that will be embedded into a VM for the remote access; (3) a security group and rules - a list of firewall rules that define ingress ports for accessing the VM or egress ports for the VM to have an access to; (4) a virtual machine (VM), specifying the properties of the VM (flavor, image, network, etc.) and associating keypair and security rules.

Listing 6.1 presents an example of a resource model for OpenStack VM, where properties, such as VM name, image, network, flavor, SSH keypair, security group, OpenStack credentials via environment variables, etc., can be provided to specify the configuration of the VM. The RE can define lifecycle operations in the model and associate implementation artifacts and inputs to these artifacts to a particular operation. Possible operations are: create, configure, start, stop, delete. In the example (lines 14-25), the create and delete operations are defined with the inputs taken from the values of the properties and Ansible playbooks used as implementation artifacts.

In Listing 6.2, an Ansible playbook for an OpenStack VM creation is shown. The inputs that are previously defined in the resource model are passed to the playbook as Ansible variables. OpenStack modules are used to interface with the OpenStack

[1] https://github.com/SODALITE-EU/ide/tree/master/dsl/org.sodalite.dsl.examples

[2] https://github.com/SODALITE-EU/iac-modules

```yaml
sodalite.nodes.OpenStack.VM:
  derived_from: tosca.nodes.Compute
  properties:
    name:
      type: string
    flavor:
      type: string
    image:
      type: string
    os_env:
      type: map
    ...
  attributes: ...
  interfaces:
    Standard:
      type: tosca.interfaces.node.lifecycle.Standard
      operations:
        create:
          inputs: ...
          implementation:
            primary: "openstack_vm_create.yml"
        delete:
          inputs: ...
          implementation:
            primary: "openstack_vm_delete.yml"
```

List. 6.1: Snippet of resource model for instantiation and deletion of an OpenStack VM

```yaml
---
- hosts: all
  tasks:
    - name: Create VM
      os_server:
        state: present
        name: "{{ vm_name }}"
        image: "{{ image }}"
        flavor: "{{ flavor }}"
        ...
      register: server_info
      environment: "{{ os_env }}"
    - name: Set attributes
      ...
```

List. 6.2: Snippet of an Ansible playbook for creating an OpenStack VM

control plane and environment property is used for setting OpenStack environment variables also obtained from the inputs. The playbook may also include tasks with *set_data* module to set the attributes of the resource model, such as public address and ID of the VM, to be later used for VM deletion or as input to other resource models with the hosting dependency.

Unlike the cloud resources that are instantiated dynamically - e.g. a virtual machine, in HPC case, the resources are static and access to an HPC cluster is usually done via a single SSH endpoint of the frontend node. The RE must define the models

```
sodalite.nodes.hpc.WorkloadManager:
  derived_from: tosca.nodes.Compute
  properties:
    scheduler: ...
    username: ...
    ssh-key: ...
  attributes: ...

sodalite.nodes.hpc.Job:
  derived_from: tosca.nodes.SoftwareComponent
  properties:
    name: ...
    script: ...
    nb_nodes: ...
  interfaces:
    Standard:
      type: tosca.interfaces.node.lifecycle.Standard
      inputs: ...
      operations:
        create: ...
        configure: ...
        start: ...
  requirements:
    host:
      capability: tosca.capabilities.Compute
      node: hpc/sodalite.nodes.hpc.WorkloadManager
      relationship: tosca.relationships.HostedOn
```

List. 6.3: Snippet of a resource model for HPC cluster access and job submission

and the implementation artifacts that will access the cluster and interface with the workload managers, such as PBS or Slurm, to submit batch jobs.

Listing 6.3 shows an example of the resource model for defining an access to and job submission in HPC cluster. Scheduler property of the WorkloadManager node type defines the workload manager to interface with, whereas username and SSH key define the credentials needed to access the cluster. Alternatively, the SSH key can be retrieved by the Orchestrator from the secrets vault (see Section **??**) in order to connect to the target cluster. The Job node type defines the properties of the job, e.g. name, number of nodes and walltime, that will be configured at the job submission. It also defines the host requirement that must be assigned to a workload manager instance. With respect to the lifecycle operations, the job creation and configuration are setting the job script in the file system of the cluster, whereas the start operation submits the job via the workload manager's interface, such as qsub and sbatch commands for job submission in PBS and Slurm, respectively.

In the similar way, the RE can provide resource models for Kubernetes. The Kubernetes cluster definition should contain the properties for cluster access, e.g. via kubeconfig file, whereas Kubernetes objects can be defined in node types that require the Kubernetes cluster instance for their creation. Ansible provides Kubernetes module[3] that allows to interface with Kubernetes API to deploy objects, and Ansible playbooks containing this module can operate the lifecycle of the objects.

[3] https://docs.ansible.com/ansible/latest/collections/kubernetes/core/k8s_module.html

```
---
- name: Call HPC Exporter API to create a collector
  uri:
    url: http://{{hpc_exporter_address}}/collector
    method: post
    body:
      scheduler: {{ "pbs" if (scheduler == "torque") else scheduler }}
      host: {{ ssh_host }}
      deployment_label: {{ deployment_label }}
      monitoring_id: {{ monitoring_id }}
      hpc_label: {{ ssh_host if (hpc_label == "") else hpc_label }}
      scrape_interval: {{ scrape_interval }}
    status_code: 200
    body_format: json
    headers:
      authorization: "Bearer {{ jwt }}"
  when: scheduler != "batch"
```

List. 6.4: Ansible playbook to create a collector on the HPC exporter

```
- name: Registering job_id in HPC Exporter
  uri:
    url: http://{{hpc_exporter_address}}/job
    method: POST
    body:
      job_id: {{ job_id }}
      host: {{ ssh_host }}
      monitoring_id: {{ monitoring_id }}
    body_format: json
  when: scheduler != "batch"
```

List. 6.5: Snippet in the job deployment playbook to register a jobid in the HPC Exporter

To enable monitoring, RE should develop the implementation artifacts that will create a node exporter and register it with SODALITE Monitoring System. For example, in order to monitor the jobs deployed to the HPC and the state of the queues/partitions, a collector needs to be registered on the HPC Exporter (see Section 5.4.1.1). To carry out this task a start operation can be added to the life-cycle of the sodalite.nodes.hpc.WorkloadManager node, presented previously, with its corresponding Ansible playbook (see listing 6.4). scrape_interval, which limits how often monitoring calls can be made to the HPC, and hpc_label, which tags the metrics coming from the HPC with a user defined name are optional properties of sodalite.nodes.hpc.WM to configure the collector.

Every time a job is deployed on the HPC a call needs to be made to the HPC Exporter to register the job-id on the corresponding collector, so it can be monitored. This can be achieved by adding an operation to the sodalite.nodes.hpc.Job node's start playbook (see listing 6.5)

In order to monitor a VM, in the Ansible playbook associated to the create node lifecycle operation of this node, an instance of both a node exporter and a Skydive agent can be deployed on the VM. The node exporter can then be registered with Prometheus by making the necessary call to Consul.

Once the resource models and implementation artifacts are developed, the RE saves the models in the Knowledge Base. The Application Ops Expert can then use them to deploy their applications or other REs can extend them to provide additional properties or functionalities.

6.3 Quality Expert

The role of the Quality Expert (QE) is provide optimization models (OM) that will be used as inputs to the Application Optimizer (MODAK, see Section 4.4) to select the optimized container runtime for an application to run on a particular infrastructure target. The workflow of the QE is presented in Figure 2.5, where the QE measures performance characteristics of resources of a particular infrastructure provider and derives its performance model, saves these characteristics and the model into the performance registry of MODAK, and based on the them defines optimization models.

MODAK exposes the following endpoints to populate its performance registry:

1. infrastructures: to define the parameters of a particular infrastructure: such as hardware, storage classes and workload managers. Listing 6.6 shows an example of a cluster definition, where Slurm workload manager is defined with two storage mount points and the *gpu* partition, with 5 compute nodes, each having 32 cores of x86 Zen2 architecture CPU, 2 NVIDIA Tesla P100 GPUs and 256 GiB of memory.
2. scripts: to define scripts that enable certain optimization steps in the job script, such as modules loading, setting environment variables, etc. Listing 6.7 is an example, where for a particular infrastructure, Cray Program Environment (CPE) for GNU should be loaded.
3. models: to define scaling models associated to a particular application. Listing 6.8 shows an example, where parallel efficiency of 0.8 is set to determine the scaling of application following Amdahl's law.

6.4 Application Ops Expert

The Application Ops Expert (AOE) typically follows the design and runtime workflows, depicted in Figures 2.3 and 2.4. He/She firstly creates application container images either manually or via Image Builder and pushes them into the image registry. Then, AOE defines an abstract application deployment model (AADM) - a model that constitutes application topology: it consists of the instances of the node types defined in the resource models by RE, relationships between the instances (e.g.

```
{
  "name": "example-site",
  "description": "possibly named after a mountain",
  "configuration": {
    "scheduler": "slurm",
    "storage": {
      "file:///scratch": { "storage_class": "default-high" },
      "file:///data": { "storage_class": "default-common" }
    },
    "partitions": {
      "gpu": {
        "nnodes": 5,
        "node": {
          "ncpus": 1,
          "cpu": { "arch": "x86_64", "microarch": "zen2", "ncores": 32, "nthreads": 2 },
          "naccel": 2,
          "accel": { "type": "gpu", "model": "P100" },
          "memory": "256GiB"
        }
  ...
}
```

List. 6.6: Snippet of a MODAK request for infrastructure made by QE

```
{
  "description": "enable CPE for GNU for all apps running on site 'example-site'",
  "conditions": {
    "infrastructure": { "name": "example-site" }
  },
  "data": { "stage": "pre", "raw": "module load cpeGNU" }
}
```

List. 6.7: Snippet of a MODAK request for scripts made by QE

```
{
  "opt_dsl_code": "some-hpc-code-v1",
  "model": { "name": "amdahl", "F": 0.8 }
}
```

List. 6.8: Snippet of a MODAK request for scaling models made by QE

dependency and host relationships) and association of optimization models into the instances. Reference examples of abstract application deployment models can be found in the SODALITE IDE repository[4].

When the AADM is developed, AOE deploys it via IDE. Upon the deployment of the application, the AOE can then monitor its runtime using the dashboard of the Monitoring System. During the runtime, the Refactorer can determine whether the application needs additional resources or the resources are overutilized, and the respective scaling can then be performed.

The following subsections will present how AOE can use the resource and optimization models already available in SODALITE for deployment over OpenStack, Slurm and Kubernetes. The resource models for AWS EC2 is similar to the ones

[4] https://github.com/SODALITE-EU/ide/tree/master/dsl/org.sodalite.dsl.examples

for OpenStack, hence they are omitted for brevity, but can be easily adapted to the examples for OpenStack. The same applies to usage of a particular HPC workload manager. The examples for Slurm can also be adapted for PBS case. This section completes with the usage of optimization models and runtime monitoring.

6.4.1 Deployment over OpenStack

A general use case of instantiating a VM in OpenStack involves the following steps: (1) obtaining valid credentials; (2) creation of an SSH keypair; (3) creation of a security group and rules; (4) creation of a VM, specifying the properties of the VM (flavor, image, network, etc.) and associating keypair and security rules.

SODALITE provides *openstack* module that contains resource models (RMs) needed to execute such use case. The OpenStack credentials can be specified as an input to an AADM. These credentials should be specified as OpenStack environment variables, such as *OS_AUTH_URL*, *OS_PROJECT_NAME*, *OS_USERNAME*, *OS_PASSWORD*, via *os_env* parameter. The *env* input can then be retrieved via *get_input* function across the whole AADM, whenever a new OpenStack resource needs to be instantiated.

An SSH keypair can be created either manually in OpenStack dashboard or via *sodalite.nodes.OpenStack.Keypair* node type. For the latter, the name of the keypair and a public key should be specified along with OpenStack credentials. For creation of a security group and rules, it is necessary to specify the name of group as well as a list of firewall rules, each containing the protocol, port range, direction (ingress or egress) and the remote IP prefix that further restricts access to the VM at the network level. A resource model for creation of a VM contains required properties that need to be specified in the AADM, such as the name of a VM, image, flavor, keypair and username used for SSH connection. If default network and security group assigned to the VM by OpenStack shall be altered, the respective properties can be defined.

Overall, Listing 6.9 shows an example of a security group for HTTPS protocol and a VM creation that runs Ubuntu 18.04 LTS image and has the security groups assigned. The *protected_by* requirement represents a dependency relationship between the VM and security group, i.e. the security group shall be created first to be later assigned to the VM.

Once the OpenStack compute resources are specified, it is now possible to define application deployment on top of the resources. The OpenStack VM node type (*sodalite.nodes.OpenStack.VM*) is derived from the standard TOSCA compute node type and provides the *host* capability. Therefore, a software component, which requires a host relationship to be specified, can use OpenStack VM as the host, where the software component will be deployed, e.g. *sodalite-os-vm*. Altogether, SODALITE provides prerequisites for the deployment of an application in OpenStack: the security firewall rules that open ports necessary for the application, the management of keypairs and credentials for management of VMs, application configuration and deployment. A user can define custom node types that require a host,

```
sodalite-security-rules:
    type: openstack/sodalite.nodes.OpenStack.SecurityRules
    properties:
        ports:
            https_port:
                port_range_min: 433
                port_range_max: 433
                protocol: "tcp"
                remote_ip_prefix: "0.0.0.0/0"
                direction: "ingress"
        group_name: "https-access"
        env: get_input: env

sodalite-os-vm:
    type: openstack/sodalite.nodes.OpenStack.VM
    properties:
        key_name: "sodalite-keypair"
        image: "ubuntu-18.04-lts"
        name: "test-vm"
        security_groups: "https-access"
        env: get_input: env
        ...
    requirements:
        protected_by:
            node: sodalite-security-rules
```

List. 6.9: Snippet of OpenStack security group creation for HTTPS and OpenStack VM creation

which then can be instantiated in an AADM, specifying an OpenStack VM in the host requirement.

6.4.2 Deployment over Slurm

An HPC cluster and access to it can be modeled using the *sodalite.nodes.hpc.WM* node type of the *batch* module, as shown in Listing 6.10. In the cluster model, the scheduler type can be specified as e.g *slurm* or *torque*, otherwise the *batch* type is assumed - a generic cluster not managed by any resource manager. *Username* and *ssh-key* are used to connect to the frontend node of the cluster with the hostname specified in the *public_address* attribute. The *ssh-key* property specifies the path to the SSH key in the Orchestrator's file system to use for the connection and is optional when the default SSH key is preferred or SSH client is configured in the *ssh_config* file. Alternatively, the SSH key can be uploaded by the user and retrieved by the Orchestrator from the secrets vault (see Section **??**) in order to connect to the target cluster. The model also contains optional *optimizations* capability, which specifies the target name that serves as an identifier for MODAK to retrieve additional optimizations if MODAK has knowledge about the target cluster.

In order to define or install (as a privileged user) a container runtime used in the cluster, *sodalite.nodes.batch.Container.Runtime* can be used. It provides properties

```
slurm-cluster-A:
  type: batch/sodalite.nodes.hpc.WM
  properties:
    scheduler: "slurm"
    username: "user_123"
    ssh-key: "/path/to/ssh/key"
  attributes:
    public_address: "frontend.cluster-a.com"
  capabilities:
    optimisations:
      properties:
        target: "slurm-cluster-A"
```

List. 6.10: Snippet of an HPC cluster model

```
hpc-app-xthi:
  type: batch/sodalite.nodes.batch.Container.Application
  optimization: cluster-a-mpi
  properties:
    app_tag: "mpi-app-xthi"
    app_type: "hpc"
    executable: "./xthi"
  requirements:
    runtime:
      node: singularity-runtime
    host:
      node: slurm-cluster-A
```

List. 6.11: Snippet of a containerized application definition

to specify which runtime to use, where the images will be stored, and in case of pulling images from the private registry, where the certificates and the registry are located. Currently, only the Singularity runtime is supported for batch applications. The runtime instance should also specify the *host* requirement, e.g. *slurm-cluster-A*.

An example of the definition of a containerized HPC application is presented in Listing 6.11, where the name of the application, type, preferred number of MPI ranks, executable, and arguments can be specified. Additionally, the build parameters can be provided, defining the source, which can also point to a Git repository or tarball, and build command. The *optimization* parameter defines the optimization model associated to the application, such that an optimized container runtime image will be selected. The details of optimization models will be presented later in the section. The *host* and *runtime* requirements define the target cluster where the application will be deployed and which runtime shall be used. The *runtime* requirement also defines the location where the optimized container images should be pulled from.

Batch jobs contain job headers and commands to execute applications, including configurations of the application environments, such as loading of the necessary libraries and drivers, and exporting variables. Job headers specify the parameters of the jobs, such as number of nodes, cores, memory, maximum walltime, queue, etc. The headers format is specific to the resource manager in-use, i.e. the job headers for PBS are different from the ones in Slurm. Therefore, the job parameters were

```
hpc-job-xthi:
  type: batch/sodalite.nodes.batch.Container.JobExecution
  properties:
    job_name: "xthi-job"
    queue: "fast-storage-queue"
    node_count: 2
    process_count_per_node: 40
    wall_time_limit: "1:00:00"
  requirements:
    application:
      node: hpc-app-xthi
    runtime:
      node: singularity-runtime
    host:
      node: slurm-cluster-A
```

List. 6.12: Snippet of a batch job definition

unified in the node type for batch jobs *sodalite.nodes.batch.Container.JobExecution*, and the resource manager is not explicitly specified, but rather derived from the *host* requirement, as shown in Listing 6.12. MODAK then generates a job script with the required job headers for the specified resource manager, as well as commands to prepare the environment and execute the application. The generated job script is then ready for execution on the cluster.

A workflow of batch jobs is an ordered execution of jobs. SODALITE provides node types that start the jobs on the cluster *sodalite.nodes.workflow.Job* and wait for the job results *sodalite.nodes.workflow.Result*. If the job result is successful, the the next job in the workflow is executed; in case of a failure, the workflow terminates. Once the failed job is fixed, the workflow can be resumed from the failed job, so that previously executed jobs are not executed again. The workflow order can be specified using the *dependency* requirement. Independent jobs can be parallelized internally by the SODALITE orchestrator to decrease the deployment time.

6.4.3 Deployment over Kubernetes

SODALITE provides several resource models, such as clusters, nodes, Kubernetes definitions and Helm charts, in the *kube* module that help to develop the deployment model for Kubernetes applications. To define a Kubernetes cluster, *sodalite.nodes.Kubernetes.Cluster* node type is used, as shown in Listing 6.13. *username* property and *public_address* attribute define the target host, where the Kubernetes client is installed. It can be either orchestrator or a remote gateway host, which can access a Kubernetes cluster. Kubeconfig files contain information about clusters, users, namespaces and authentication mechanisms for accessing defined clusters. While *kubeconfig* property defines the path to the kubeconfig file, which already exists on the target host, *kubeconfig_raw* defines the raw values for kubeconfig, which will be created and later deleted, once the deployment finishes.

```
kube-cluster:
  type: sodalite.nodes.Kubernetes.Cluster
  properties:
    kubeconfig: "~/.kube/config"
    kubeconfig_raw: get_input: kubeconfig_raw
    username: "centos"
  attributes:
    public_address: "kube_master.sodalite.eu"

node-xavier-nx:
  type: kube/sodalite.nodes.Kubernetes.Node
  properties:
    name: "xavier-nx"
    gpus: 1
    cpus: 1
    edgetpus: 1
    arm64_cpus: 1
```

List. 6.13: Snippet of a Kubernetes cluster and node definition

The hardware characteristics of nodes of a Kubernetes cluster are defined in the *sodalite.nodes.Kubernetes.Node* node type and can be either populated by the Resource Experts or discovered with Platform Discovery Service. This node type provides properties for describing hardware characteristics, such as number of CPUs, CPU architecture, number of accelerators (GPUs, EdgeTPUs), that serve as a hint to the AOE to select the node, where the pods will be scheduled to based on the hardware requirements. As an example, Listing 6.13 presents a node that contains an EdgeTPU, a GPU and an ARM64 CPU.

In order to deploy an application on Kubernetes, Helm charts can be used. Helm charts can be deployed cluster-wide or targeting a specific node, e.g. with accelerators. In both cases, the properties are common and can define the Helm chart repository, name, version and values, however, node specific deployment must define additional *kube_node* requirement to specify the target node of the Kubernetes cluster. An example of a node specific deployments of MySQL database is outlined in Listing 6.14. The *kube_node* requirement specifies the node with a GPU, as defined earlier in Listing 6.13, hence the chart can utilize the GPU to run its workloads. Without this requirement, the deployment will be cluster-wide.

6.4.4 Optimization models

In Section 6.4.2, the *optimization* parameter was introduced for batch applications to associate the optimizations that will be applied to the application. The SODALITE IDE provides an optimization model editor to create such optimizations, which will then be included in the container runtime images selected by MODAK. An optimization model has the following structure:

```yaml
mysql-helm-cluster-wide:
  type: kube/sodalite.nodes.Kubernetes.Definition.Helm
  properties:
    name: "mysql-release-1-from-helm"
    namespace: "default"
    chart: "stable/mysql"
    chart_version: "latest"
    repo_name: "stable"
    repo_url: "https://charts.helm.sh/stable"
    keep_repo: false
    values:
      replicas: 2
  requirements:
    host:
      node: kube-cluster
    kube_node:
      node: node-xavier-nx
```

List. 6.14: Snippet of a node specific application deployment via Helm chart

- *enable_opt_build*: specifies whether an optimized build specific to a particular hardware target should be included. The list of targets is in *opt_build* parameter.
- *enable_autotuning*: specifies whether autotuning should be included. See Section 4.4.2 for details.
- *app_type*: specifies application type. Currently traditional HPC and AI training are supported.
- *opt_build*: specifies the hardware target (CPU and accelerator types) for the optimized container.
- *autotuning*: specifies optional configuration for the autotuning. Users will provide a script that can be used some application input parameters for the best performance.
- *app_type-hpc* and *app_type-ai_training*: specifies the optimization configuration specific to HPC or AI training application types.

Listing 6.15 shows two optimization models that represent an optimization for HPC with the MPICH v3.1.4 MPI library, and AI training using TensorFlow 2.1 with XLA accelerated with Nvidia GPUs. These optimization models can be associated to the HPC and AI applications in AADM. MODAK will then select optimized container runtime images and generate build and run instructions for executing the applications using these optimized images. The *config* parameter in the optimization configuration for HPC applications specifies which parallelization type should be chosen as an optimization: MPI, OpenACC, OpenCL or OpenMP. For each of the parallelization types, further configurations can be provided, such as a specific MPI library (MPICH, OpenMPI, MVAPICH) and version, the compiler for OpenACC and OpenCL, or affinity for OpenMP. For AI training applications, *config* specifies an AI framework, such as TensorFlow, PyTorch, or Keras. For each framework the version can be selected and further acceleration can be enabled, such as XLA for TensorFlow or Glow for PyTorch.

```
optimization cluster-a-mpi:
  enable_opt_build: true
  enable_autotuning: false
  app_type: hpc
  opt_build:
    acc_type: none
    cpu_type: x86
  app_type-hpc:
    config:
      parallelisation: mpi
    parallelisation-mpi:
      library: mpich
      version: "3.1.4"

optimization cluster-a-gpu:
  enable_opt_build: true
  enable_autotuning: false
  app_type: ai_training
  opt_build:
    acc_type: nvidia
    cpu_type: x86
  app_type-ai_training:
    config:
      ai_framework: tensorflow
    ai_framework-tensorflow:
      version: "2.1"
      xla: true
```

List. 6.15: Snippet of optimization models used for HPC and AI training

6.4.5 Monitoring and Refactoring

Once an application has been deployed, the AOE can monitor in real time the metrics generated by the different exporters on Grafana. In the Governance View of the IDE a link is provided for each of the Dashboards that are created automatically for the deployment. The AOE can log in Grafana through the Keycloak account. Fig. 5.7 shows an example of the node exporter dashboard, containing metrics for the VMs that have been deployed as part of the application.

The IDE provides an editor for alerting rules, the AOE can use it to define rules that, when violated, signal the refactoring engine to redeploy an application modifying the necessary parameters. The rules are written in PromQL and the IDE assists in their definition with contextual suggestions similar to the resource and AADM editors. Listing 6.16 shows an example of an alerting rule that gets triggered when a VM's disk is over 80% full. When the rules are ready they can be registered on the backend by the IDE itself, which sends it to the rule-server, an API used to register the rules on Prometheus dynamically.

In response to the alerts generated by the SODALITE monitoring layer, the application may need to be reconfigured and deployed. The AOE can define such adaptation decisions as set of ECA rules, and configure the deployment refactorer with those rules. Listing 6.17 shows an example of an adaptation rule that reacts to the event *OutOfDiskSpace* by replacing the VM that host the snow application between a new VM having a sufficient disk.

```
groups:
- name: alert.rules
  rules:
  - alert: OutOfDiskSpace
    expr: |
      (node_filesystem_avail_bytes{deployment_id="5sdfsdf121"} * 100) /
      node_filesystem_size_bytes{deployment_id="5sdfsdf121"} < 20 and
      ON (instance, device, mountpoint)
      node_filesystem_readonly{deployment_id="5sdfsdf121"} == 0
    for: 2m
    labels:
      severity: warning
    annotations:
      monitoring_id: '5sdfsdf121'
      instance: '{{ $labels.instance }}'
      summary: Disk space running low (instance {{ $labels.instance }})
      description: |
        'Available disk space is low (< 20% left)
        VALUE = {{ $value }} LABELS: {{ $labels }}'
```

List. 6.16: Disk full alerting rule

```
rule "OutOfDiskSpace"
  when
    $f1 : Alert(name == "OutOfDiskSpace")
  then
    Node newNode = refMgt.findMatchingNodeFromRM("(?disk > 150)");
    AADMModel aadmModel = refMgt.getAadm();
    aadmModel.replaceNode("snow-vm", newNode);
    // Update requirements, etc
    refMgt.saveAndUpdate(); // Update the deployment with the new model
end
```

List. 6.17: A snippet of a deployment adaptation rule

6.5 Conclusion

This chapter provided examples of common usage of SODALITE platform by the users, who are responsible for different cases. While the role of the Resource Expert is to define resource models and implementation artifacts, the Quality Expert defines optimization models that help to statically optimize the applications. Both of them share their respective models with Application Ops Expert, who incorporates the models in the abstract application deployment model, in order to deploy applications into various execution platforms in the Cloud, Edge, and HPC domains.

Chapter 7
SODALITE Use Cases

Joao Pita Costa, Piero Fraternali, Kalman Meth, Paul Mundt, Giovanni Quattrocchi, Ralf Schneider, Kamil Tokmakov, and Rocio Nahime Torres

Abstract This chapter describes the various use cases of the SODALITE project in more detail. Each use case is representative of a unique infrastructure and operational environment supported by SODALITE: Snow (Cloud/OpenStack), Clinical Trials (HPC/Torque), and Vehicle IoT (Cloud + Edge / Kubernetes). Each section includes an overview of the specific challenges faced by the use case, an overview of the use case system architecture, target infrastructure, and operational environment, followed by ways in which the SODALITE approach was successfully applied to address the unique challenges of the use case. Each section concludes with a brief review of the benefits that the use case achieved through the application of SODALITE.

7.1 Introduction

The SODALITE approach has been applied during the development of the project to three different case studies. They have shown to be quite complementary one with respect to the other and have allowed us to experiment with different aspects of the SODALITE platform.

The rest of this chapter is dedicated to the presentation of all the three cases. More specifically, Section 7.2 presents the Snow use cases, which has been already

Paul Mundt
Adaptant Solutions AG, Berlin, Germany e-mail: `paul.mundt@adaptant.io`

Kamil Tokmakov
High Performance Computing Center Stuttgart (HLRS), Germany, e-mail: `kamil.tokmakov@hlrs.de`

Ralf Schneider
High Performance Computing Center Stuttgart (HLRS), Germany, e-mail: `ralf.schneider@hlrs.de`

Joao Pita Costa
XLAB, Slovenia, e-mail: `joao.pitacosta@xlab.si`

introduced in Chapter 5; Section 7.3 presents the in silico clinical trials use case, which shows strict QoS requirements and the need to be executed on HPC; Section 7.4 provides an overview of the Vehicle IoT case, which introduces a new challenging execution environment, that is, the edge. Finally, Section 7.5 concludes the paper.

7.2 Snow Use Case

The Snow use case has been briefly introduced in Chapter 2. This section provides further details on it and on how the usage of the SODALITE platform has been beneficial for modeling the use case deployment and for its execution.

The use case is about collection of images from multiple sources, their analysis and transformations to deduce from them information about the amount of snow available on mountains and, therefore, the available reserve of water. Application components are logically organized in a pipe and filter approach and are mostly executed on cloud resources. In this context, SODALITE offers proper mechanisms to model the deployment of the system and to control its operation and reconfiguration.

The following subsections will elaborate more on the motivation behind the use case, describe the main components of the use case, and provide more details on how SODALITE was used to deploy the use case and on the achieved benefits.

7.2.1 Background and motivation

With climate change impacting the planet, the issue of water scarcity has entered political discourse globally. Even in countries that do not yet suffer water poverty, conservation has become a prominent goal. Making good use of water implies predicting its availability and planning usage in a way that satisfies the potentially conflicting objectives of industry, agriculture, and the common people.

In mountain regions, the water stock is essentially preserved in the form of snow and permafrost. The Alps are the highest and most extensive mountain range system that lies entirely in Europe. The climate changes have a deteriorating impact on the region, negatively affecting the ecosystem as well as a population of 14 million people spread across eight countries. The problem is worsening with increasing recorded annual average temperatures and decreasing precipitation levels. At the same time, ground monitoring infrastructures have not been updated due to high investment requirements. This calls for novel instruments for the low-cost, high resolution, high accuracy monitoring of the dynamics of environmental events and conditions in mountain regions, especially in the Alps.

The Snow Use Case plans to provide a solution by improving the capillarity of mountain environment monitoring. It applies advanced image processing workflows capable of extracting useful environment information from large collections of publicly available mountain related multimedia data.

7.2.2 The snow detection pipeline

The solution pursued in the Snow Use Case is an original method to derive information on mountain snow coverage from a massive amount of public web content (user generated photos and images captured from touristic web cams). The method uses an image processing workflow that aligns the picture taken by the user or crawled from a touristic web cam to a synthesized rendered view of the terrain that should be seen from the camera point of view; the synthetic panorama is generated by a projection applied to a (publicly available) Digital Elevation Model (DEM) of the Earth[1]. This supports decision-making in a snow dominated mountain context, e.g., the prediction of water availability from snow coverage in mountain peaks.

Content acquisition and processing rely on crawling geo-located images from heterogeneous sources at scale. The proposed approach elaborates content by assessing the presence of mountains in each photo, identifying individual peaks, and extracting a snow mask from the portion of the image denoting an identified mountain. Architecturally, the system is distributed between several data centers. The alignment step is computationally intensive and can run on CPU, but is best executed on GPUs. The alignment operation must be executed for every acquired image. In particular, the image analysis workflow extracts the skyline for an image taken in uncontrolled conditions, and thus must implement a multi-stage, GPU-intensive sequence of steps, including image normalization, jitter compensation, occlusion detection, and skyline extraction. Given the massive amount of images to be processed to monitor the entire Alpine region over a long period of time (in the order of millions of images per year), the workflow must be deployed on a massively scalable, GPU-enabled architecture.

Figure 7.1 shows the different components of the pipeline. Two main image sources are used: touristic webcams in the Alpine area and geo-tagged user-generated mountain photos in a 300 x 160 km Alpine region. The subsections below describe the components of the pipeline in details.

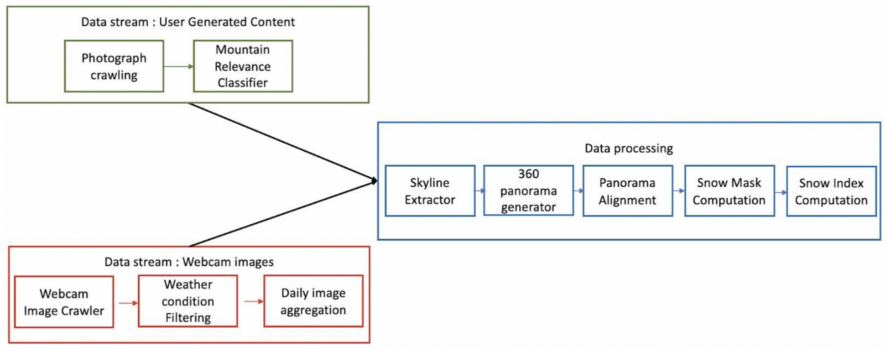


Fig. 7.1: Components of the Snow Use Case pipeline

[1] https://www2.jpl.nasa.gov/srtm/

7.2.2.1 User Generated Image Crawler

This component crawls Web image sharing sites for images representing mountains. It takes as input the coordinates of a rectangular geographical region and mountain-related keywords, then fetches images from the specified area whose metadata contain the keywords. To do so, it opens a connection to the query API of the image sharing site, submits queries formulated with the input keywords and search area, retrieves images that match the query, downloads the images, saves them on disk and stores their metadata in the database. Flickr is exploited as the data source for user-generated photographs, as it contains a large number of publicly available images, many of which have an associated geotag (GPS latitude and longitude position saved in the EXIF (Exchangeable Image File Format) container of the photograph). Figure 7.2 shows two examples of crawled images.

Fig. 7.2: Examples of crawled images

7.2.2.2 Mountain relevance classifier

Images tagged with a location within a mountainous region may not portray mountain landscapes. For this reason, the probability of mountain presence is estimated and the non-relevant photographs are discarded. This component takes as input an image, computes a fixed-dimensional feature vector that summarizes its visual content, and feeds the vector to a multi-layer perceptron classifier that determines whether the image is relevant. A dataset of images annotated with mountain/no-mountain labels is used to train the model. Figure 7.3 exemplifies positively (left) and negatively (right) classified images.

7.2.2.3 Webcam image crawler

Outdoor webcams represent a valuable source of visual content. Their images must be filtered by weather conditions, because these can significantly impact the visibility

Fig. 7.3: Crawled image of a mountain (left) and not a mountain(right)

of mountains. Since snow cover changes slowly over time, multiple webcam images of the same day can be aggregated so as to produce only one representative sample per day. Public webcams expose a URL which returns the most recent available image. The webcam crawler loads the list of all the webcams and starts an asynchronous loop for each of them. At each iteration, it checks the current webcam image and adds it to the dataset if it is changed w.r.t. the previous access. It then idles for 1 second and starts over again. The image check is performed only on the first 5KB of the image, which are hashed and compared to the previous hash of the same webcam: if the hash is different, it is saved as the new hash and the whole image is downloaded. The first image acquired from every webcam is discarded, because it may be stale (some webcams, due to failures, expose the same images for days).

7.2.2.4 Weather condition filter

Clouds, rains and snowfalls may hinder visibility significantly and thus only a fraction of the acquired images can be exploited for estimating snow cover. The weather condition filter is based on the assumption that, if the visibility is sufficiently good, the skyline mountain profile is not occluded. The component takes as input a webcam image and outputs a Boolean value indicating if a mountain skyline is visible or not.

7.2.2.5 Daily median image aggregation

Good weather images might suffer from challenging illumination conditions (such as solar glare and shadows) and moving obstacles (such as clouds and persons in front of the webcam). Yet snow cover changes slowly over time, so one image per day is sufficient. This component aggregates the images collected by a webcam in a day and outputs a single image per webcam obtained by applying the median operator along the temporal dimension. Such a median image also removes temporary lighting effects and occlusions.

7.2.2.6 Skyline extraction

To geo-reference the snow cover, it is necessary to determine which portions of the image represent which mountains and estimate for each pixel its content (terrain or sky) and the corresponding GPS position, altitude, and distance from the observer. Mountain image geo-registration is done by finding the correct overlap between the photograph and a 360-degree virtual panorama. To align the photo and the virtual panorama, the landscape skyline is extracted from the photo, by finding the pixels that represent the boundary between the terrain and the sky with a pixel-level binary classifier. The classifier is trained using a dataset of mountain images, annotated with the skyline boundary. The component takes as input an image, the camera field of view and shooting position, and outputs a skyline mask. Figure 7.4 shows a sample image (left) and the extracted skyline represented as a binary mask (right).

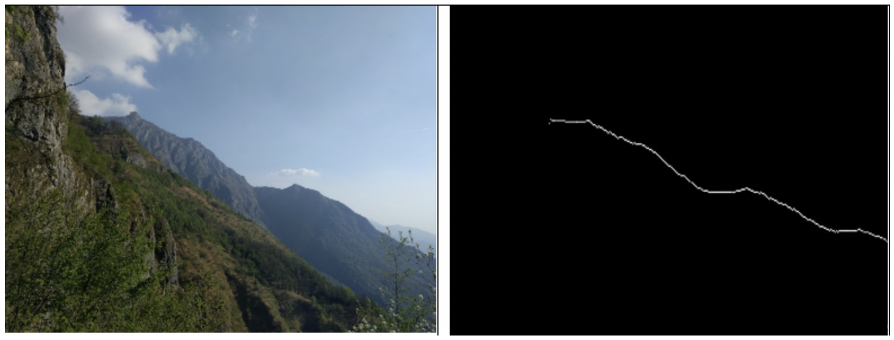


Fig. 7.4: Sample image (left) and extracted skyline represented as a binary mask (right)

7.2.2.7 360 Panorama generation

The virtual panorama is a synthetic image of the visible mountain skyline generated by projecting the Digital Elevation Model (DEM) of the Earth provided by the NASA SRTM data set from the camera shooting position. The component is deployed as a service that takes as input the coordinates of a location and generates an image corresponding to the 360-degree panorama visible from that point. Figure 7.5 shows an example of a 360-degree virtual panorama with the skyline visible from a given location.

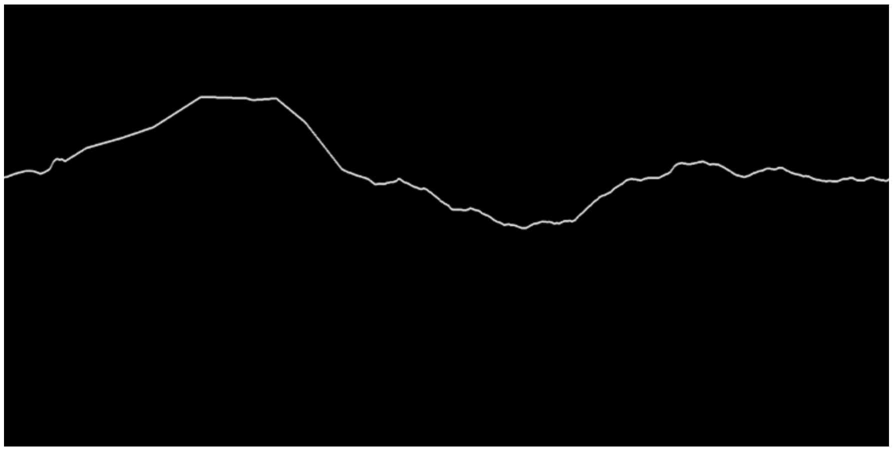


Fig. 7.5: Example of 360-degree virtual panorama with the skyline visible from a given location

7.2.2.8 Panorama alignment

The alignment between the skyline of the real image and of the virtual panorama can be seen as the search for the correct overlap between two cylinders: one containing the 360-degree virtual panorama and the other one containing the skyline extracted from the photo. The component takes as input the real and virtual panorama images and returns the real image annotated with a mask that represents the portion of the image that contains the mountain registered with the virtual panorama. Figure 7.6 shows an example of alignment between the 360-degree virtual panorama and the skyline extracted from the image.

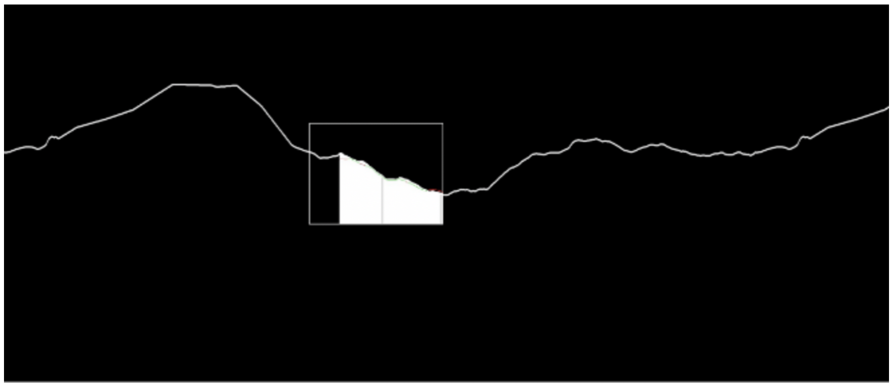


Fig. 7.6: Example of alignment between the 360-degree virtual panorama and the skyline extracted from the image.

7.2.2.9 Snow mask computation

A snow mask is the output of a pixel-level binary classifier that, given an image and a mask M that represents the mountain area as input, produces a mask S that assigns each pixel of the mountain area a binary label denoting the presence of snow. Snow masks are computed using the Random Forest supervised learning classifier with spatio-temporal median smoothing of the output. The classifier is trained with images annotated at the pixel level indicating if the pixel corresponds to the snow area. Figure 7.7 shows an example of an image and its snow mask.

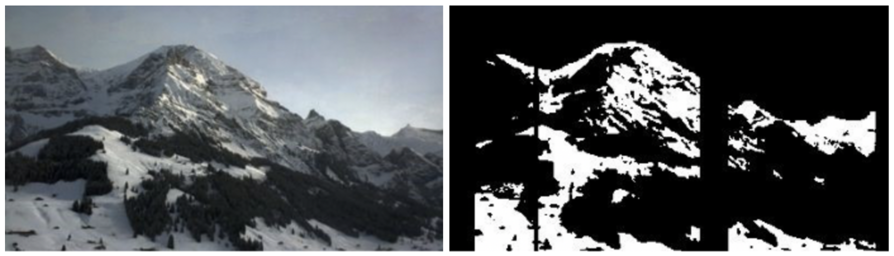


Fig. 7.7: Example of an image and of the snow mask generated from it

7.2.2.10 Snow index computation

The pipeline produces a pixel-wise snow cover estimation from images, along with a GPS position, camera orientation, and mountain peak alignment. Thanks to the image geo-registration with the DEM data it is possible to estimate physical and geographical properties of every pixel, including its type (snow/no snow) and altitude. Consequently, it is possible to compute the snow line altitude (the point above which snow and ice cover the ground) expressed in meters.

7.2.3 How SODALITE is used

The Snow Use Case takes advantage from SODALITE in multiple ways. More specifically:

- to enable training of the skyline extractor, we exploit SODALITE ability to automate the deployment of application components on HPC resources and to optimize their execution;
- to manage the allocation of heterogeneous resources (i.e., CPUs and GPUs) at runtime, we used component Node Manager that allows to efficiently provision resources with the goal of minimizing SLA violations

- to automate the deployment of the whole pipeline on a cloud infrastructure, we rely on the definition of an articulated Abstract Application Deployment Model that containerizes application components and distributes them on different virtual machines;
- to inject images from data sources, e.g. webcam, we have used the integration between NiFi and SODALITE.

More details on these aspects are described in the following subsections.

7.2.3.1 Deployment on HPC

The training of the skyline extraction component has been optimized for the execution on HPC resources. The goal of this component is to obtain the landscape skyline of a photograph via a DL classification method run in TensorFlow. The dataset used for the training consists of 8,856 images with skyline annotations, from which 80% is used for training and validation and the remaining 20% for testing. The component was initially trained using TensorFlow 1.11. The training was performed with a baseline container taken from DockerHub (tensorflow/tensorflow:1.11.0-gpu-py3) and converged within approximately 7.2 hours on one GPU node of the HPC testbed (using single core execution). The training executed until convergence was achieved and early stopping initiated at epoch 20.

The first step of the optimization process was the porting of the Python training code for TensorFlow 2.2, as it has been optimized by the developers to outperform the outdated TensorFlow 1.11. Therefore, we built an optimized Singularity container with TensorFlow 2.2. As a sanity check, we performed a run until convergence which finished within 8056s (approximately 2.3 hours) and 20 epochs. As training times converge across epochs within 2-3 epochs, we trained the skyline extractor for 5 epochs across every further optimization we considered. For the initial Singularity container with TensorFlow 2.2, that took 3473s, of which 872s constitute training time, while the rest includes data batching time. This is a well-known bottleneck for DL applications involving massive datasets. To account for this, we optimized the Python code to perform batch dataset prefetching to the GPU via the TensorFlow Data API. This shortens the execution pipeline by performing training and data input concurrently. The training time thus improved to 2181s, of which 514s constitute training time. As a final optimization, we optimized the data movement by staging the dataset on an SSD attached to the GPU node. The dataset was moved to the SSD, and the dataset directory passed to Singularity via file binding. This optimization improved the training time to 424s, of which 236s constitute training time. This yields an 8.2x speedup improvement over the initial TensorFlow 2.2 run. We tested additional optimizations such as using XLA and various combinations of SSD, GPU prefetching, and XLA, but these did not yield significant improvements. Finally, we executed the optimized container up to convergence. It takes 2042s with 21 epochs. Overall, this is a 12.7x speedup.

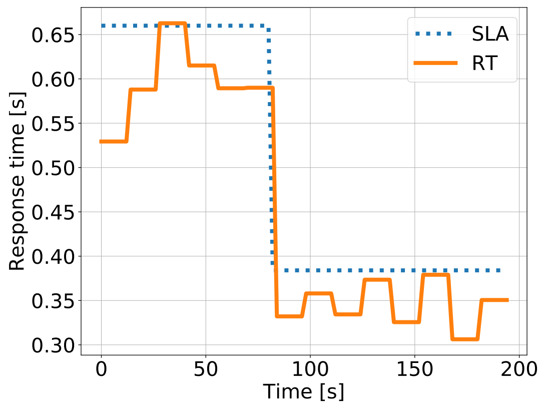


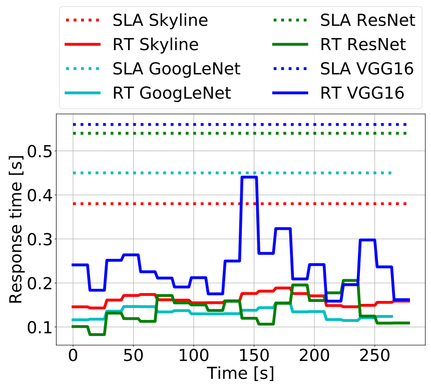
Fig. 7.8: Skyline extractor controlled by Node Manager.

MODAK (see section 4.4) can be used to automate the process of choosing an optimal container, thus returning the best possible container, in this case one that stages the dataset to an SSD.

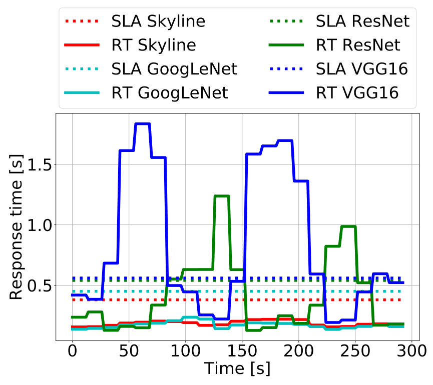
7.2.3.2 Resource Management

As shown in Section 5.6 component Node Manager is able to manage a set of applications that share a cluster of heterogeneous resources. Users set SLAs for each applications and the Node Manager i) deploys containers for the applications using Kubernetes, ii) schedules incoming requests on either GPUs or CPUs according to application needs, iii) dynamically allocates CPU cores to avoid SLA violations. In the context of Snow UC we configured Node Manager to control application skyline extractor along with other third party apps (ResNet, GoogLeNet, and VGG16) on a shared cluster deployed on the Azure public cloud. The cluster was composed of three virtual machines: one VM of type HB60rs with a CPU with 60 cores and 240GB of memory, and two VMs of type NV 6 equipped with a NVIDIA Tesla M60 GPU and a CPU with 6 cores and 56GB of memory. An additional instance of type HB60rs was used for generating the client workload. Different shaped, highly varying synthetic workloads were tested in all the experiments run and the different applications were run in random combinations concurrently on the servers.

The first type of experiment conducted was about varying either the input workload or the set-point of the system to test the ability of Node Manager to rapidly adapt the resource allocation to the new state. Results showed that the Node Manager is able to efficiently adapt to different unforeseen conditions. Figure 7.8 shows how Node Manager quickly reconfigures the resources when the SLA is changed (around second 80) for application skyline extractor to avoid SLA violations.



(a) Node Manager



(b) Rule-based approach

Fig. 7.9: Comparison.

Node Manager was compared with a rule-based approach that schedules incoming requests using a round-robin approach on available CPUs and GPUs and dynamically scales the resources using a rule-base engine. Different synthetic workloads were tested and the Node Manager outperformed the baseline in all the experiments obtaining overall 96% fewer SLA violations while using 15% fewer resources. Figure 7.9 shows the different behavior of the systems (with the same workload when all the four applications were running concurrently. While Node Manager can quickly react to changes, rule-based approach often violates the SLAs for applications ResNet and VGG16.

By adopting the Node Manager, use case owners can deploy components that exploit heterogeneous resources, set constraints on their response times and have the platform automatically managed for optimizing resource allocation and fulfill the desired goal. Node Manager is able to control multiple applications at the same time and to govern potential resource contention scenarios among concurrent applications. As clearly shown in Figure7.9, Node Manager outperforms a rule-based approach by order of magnitude. The SLA violations are minimized (96% improvement) and the resources are precisely allocated to the different containers (15% improvement).

```
module: snow
import: docker
import: openstack

inputs:
  ssh-key-name:
    type: string
  image-name:
    type: string
  ...

node_templates:
  snow-security-rules:
    type: openstack/sodalite.nodes.OpenStack.SecurityRules
    properties:
      ports:
        component_ports:
          port_range_max: 8084
          remote_ip_prefix: "0.0.0.0/0"
          port_range_min: 8080
          protocol: "tcp"
    ...

  snow-vm:
    type: openstack/sodalite.nodes.OpenStack.VM
    properties:
      key_name:  get_input: ssh-key-name
      image: get_input: image-name
      name:  "snow-vm_new_1"
      network: get_input: openstack-network-name
      security_groups:  get_input: security-groups
      flavor:  get_input: flavor-name
      username: get_input: username
      env: get_input: env
    requirements:
      protected_by:
        node: snow/snow-security-rules

  snow-vm-2:
    type: openstack/sodalite.nodes.OpenStack.VM
    ...

  snow-docker-host:
    type: docker/sodalite.nodes.DockerHost
    ...

  snow-docker-registry:
    type: docker/sodalite.nodes.DockerRegistry
    ...

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

  snow-webcam-crawler:
    type: docker/sodalite.nodes.DockerizedComponent
    properties:
      docker_network_name:  get_property:
                              entity: SELF
                              property: snow/snow-docker-network.name
                              req_cap: snow/snow-webcam-crawler.network
      volumes: [...]
      image_name:  "snow-webcam-crawler:latest"
      alias: "snow-webcam-crawler"
      command: "34,40,50,62,608,666,822,852,943,1307,6666"
      registry_url: get_input: docker-registry-url
    requirements:
      host:
        node: snow/snow-docker-host
      dependency:
        node: snow/snow-mysql
      network:
        node: snow/snow-docker-network
      registry:
        node: snow/snow-docker-registry
  ...
```

List. 7.1: Snippet of the Snow UC AADM

7.2.3.3 Deployment on the Cloud

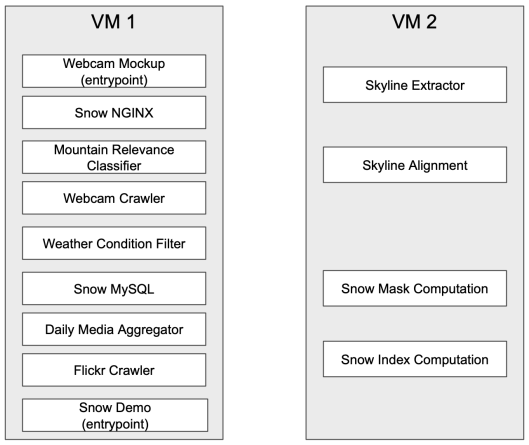


Fig. 7.10: Deployment of the Snow UC on SODALITE testbed.

The Snow Use Case was modeled using SODALITE AADM to facilitate its deployment on the SODALITE testbed. Figure 7.10 shows how components were deployed onto two VMs provisioned using OpenStack. The first VM contains more components but they are generally less demanding of resources than the four deployed on the second virtual machine. Listing 7.1 reports an extract of the AADM. Lines 1 – 3 declare the *snow* module and import the necessary dependencies where Docker and OpenStack types are declared. Lines 5 – 10 define the inputs (partially omitted) of the AADM. These values must be prompted by the user before starting a deployment. Lines 13 – 22 show how the security rules for the OpenStack VMs are defined. In particular ports between 8080 and 8084 are opened so that Snow components can be used by clients after the deployment. Lines 24 – 41 declare the two VMs. These nodes use some of the inputs declared above (e.g., *ssh-key-name*) and are protected by the defined security rules (e.g., lines 36 – 37). Lines 43 – 53 define Docker related nodes, namely a Docker host, registry and network. Finally lines 55 – 82 report the definition of component *snow-webcam-crawler*. The node is declared as a container with an image that is assumed to be stored in the Docker registry defined at lines 47 – 49. The component has some key requirements: a docker host (lines 68 – 69), a MySQL database (lines 70 – 71) which definition is omitted herein, a Docker network (lines 72 – 73) and registry (lines 74 – 75).

7.2.3.4 Data Management

One of the tasks of Snow UC component - WebCam Crawler - is to pull images from public webcams and store the images in either local file system or in a MinIO (S3 compatible) object storage. Data Management component is an alternative to WebCam Crawler with the benefits for scalable data movement and support for various heterogeneous data sources. Data Management component creates a data pipeline, consisting of HTTP consumer and S3 Publisher - shown in Figure 7.11, that allows to periodically get the images from the public webcam and store them in S3 storage.

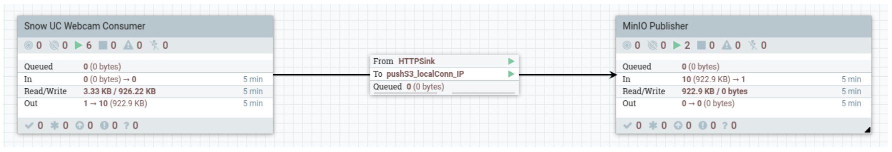


Fig. 7.11: Data pipeline, connecting HTTP Consumer and S3 Publisher

Listing 7.2 shows an AADM for such data pipeline. The *webcam-consumer* component is a subpipeline that receives the JSON list of objects (*List*), containing URL of the webcams and any arbitrary data that can be used e.g. to name the images in a specific format in *Filename*, and downloads the images. These arbitrary data can be extracted through *AttributeMap*. The *ListHTTPSchedulingPeriodTimer* property sets the scheduling of pulling the images, e.g. once a minute. The *connectToPipeline* requirement connects the subpipeline to another subpipeline - S3 Publisher, which saves the downloaded images into the specific S3 bucket. Additional properties of the S3 Publisher include access and secret keys and the endpoint to the S3 storage.

7.2.4 Benefits of using SODALITE

The Snow Use Case represents a typical scenario in which a complex data processing pipeline comprising components with very different non-functional requirements and resource usage profiles should be deployed and administered over a long period of time. The application targets public administrations, such as environment protection agencies and water bodies regulators, which makes simplicity of maintenance and cost effectiveness prominent requirements.

In such a scenario, the first benefit of SODALITE is the advantage of a model-driven approach. In fact, the definition of the Abstract Application Deployment Model proven to be an effective tool to raise to the conceptual level of architecture design the problem of defining the details related to the pipeline deployment, so as to enable an effective communication between the operation manager and the application owners. More specifically, an analysis of the pipeline from the deployment

```
webcam-consumer:
  type: datapipeline/sodalite.nodes.datapipeline.Consumer.HTTP.JsonList
  properties:
    name: "Snow UC Webcam Consumer"
    canvas_layout:
      origin_X: 600.0
      origin_Y: 300.0
    List: get_input: webcam_object_list
    AttributeMap:
      id: "$.id"
      url: "$.url"
    Filename: "${id:append('.jpg')}"
    ListHTTPSchedulingPeriodTimer: "1 min"
  requirements:
    connectToPipeline:
      node: s3-publisher

s3-publisher:
  type: datapipeline/sodalite.nodes.datapipeline.Publisher.S3Bucket
  properties:
    name: "MinIO Publisher"
    canvas_layout:
      origin_X: 1500.0
      origin_Y: 300.0
    BucketName: "wc-shots"
    Region: "eu-central-1"
    AccessKey: get_input: minio_access_key
    SecretKey: get_input: minio_secret_key
    EndpointOverrideURL: get_input: minio_endpoint
```

List. 7.2: Snippet of the AADM for data management to periodically transfer webcam images to S3 storage

and operation perspective has highlighted a number of design and implementation issues concerning, among the others, the approach adopted to coordinate the pipeline components, the way to manage data transfer between components, the importance of clearly identifying all software layers to be wrapped in each component container, as well as the need to parametrize the application software for what concerns information such as IP addresses and user credentials. The importance of the AADM definition increases even further when considering the case in which the Snow system is hosted by a third party, a frequent case in PAs. In this case, the explicit definition of the AADM simplifies the migration from one provider to the other compared to the case when the deployment and configuration of the system is manually handled.

A second benefit is the monitoring of complex runtime conditions possibly spanning multiple components and measured by means of declarative rules. Such a capability goes well beyond the alarm triggers of commercial cloud platforms and is essential in such an articulated and heterogeneous pipeline as the one of the Snow Use case, where many things can get critical at the same time.

Other benefits, not even foreseen in advance, concern the possibility to take advantage not only of the cloud but also of HPC clusters for specific operations (the skyline extraction in our case), to exploit the Node Manager for increasing the efficiency of execution and to exploit SODALITE-ready mechanisms for data transfer.

7.3 In-silico Clinical Trials Use Case

This section describes the in-silico clinical trials use case and how the usage of the SODALITE platform is beneficial for the use case execution. The use case reproduces real clinical trials in biomecanics by means of simulation to determine optimal bone implant systems for patients with spinal conditions. The workflow of the use case was originally executed on a specific HPC infrastructure, however, with SODALITE the hybrid execution of the workflow was achieved, additionally targeting cloud environments. Moreover, SODALITE provided a user interface, which assists in the development of deployment model for the use case workflow, and helped to optimise the runtime execution of certain components of the use case.

The following subsections will elaborate more on the motivation behind the use case and describe the simulation chain of the use case, as well as provide more details on how SODALITE was used to deploy the use case and what were the benefits of SODALITE usage for the use case.

7.3.1 Background and Motivation

The in-silico clinical trials for spinal operations use case targets the development of a simulation process chain supporting in-silico clinical trials of bone-implant-systems in Neurosurgery, Orthopedics, and Osteosynthesis. It deals with the analysis and assessment of screw-rod fixation systems for instrumented mono- and bi-segmental fusion of the lumbar spine by means of continuum mechanical simulation methods. As a novelty, we consider the uncertainty inherent in the computation by means of probabilistic programming. The simulation chain consists of a number of steps that need to be fulfilled in order and can be thought of as a pipeline. The output of each step serves as input to the next step.

The use case addresses one of the most prevalent health problems experienced by the populations of developed nations resulting in enormous losses of productivity and costs for ongoing medical care. The simulation process developed within this use case will optimize the screw-rod fixation systems based on clinical imaging data recorded during standard examinations and consequently target the lowering of the reported rates of screw loosening and revisions, enhance safety, expand the knowledge of the internal mechanics of screw-rod fixation systems applied to the lumbar spine and finally reveal optimization potential in terms of device application and design.

7.3.2 The Simulation Chain of Clinical Trials Use Case

The individual steps for the simulation chain can be seen in Figure 7.12 below. First, the Image Processing and Filtering component receives three data sets reconstructed in different image planes and generates a high resolution image dataset. The Extraction component then takes this enhanced imaging dataset and extracts a geometry for the vertebral bodies. The de-facto standard for doing this is the marching cubes algorithm, for which many implementations exist.

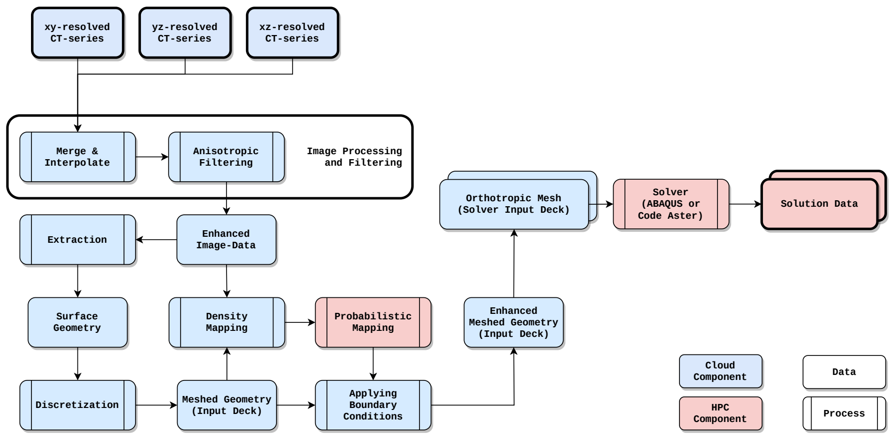


Fig. 7.12: The simulation chain of the Virtual Clinical Trial use case

Next, the Discretization component generates a volume-mesh inside the surface geometry. This enables one to treat the mesh as a set of finite elements and to use the existing finite elements methodology. The Density Mapping component takes the original image data and maps it onto the volume-mesh. In doing this, each element inside the mesh is assigned a density value.

This enhanced meshed geometry is then fed into the Probabilistic Mapping component. Here the values for density are transformed into values for elasticity, as this is what is actually needed for the simulation. Because of the uncertainty that is inherent in this transformation, we use a probabilistic programming approach. Eventually, boundaries for the 95% highest density interval as well as the mode are computed.

These data are used in the input decks for the last step, the Solver component. Here, the finite element method is actually used in computing a solution that describes the structural mechanics inside the vertebral bodies.

7.3.2.1 Image Processing and Filtering

The image processing and filtering component is composed out of several processing steps, integrated into a pipeline via the Visualization Toolkit[2] (VTK), which provides all steps as ready-to-use algorithms with corresponding python bindings. As input

[2] http://www.vtk.org

the implemented pipeline takes three image series in VTK file format[3], which are reconstructed in different image planes i.e. whose resolution differs along the three coordinate axes as described in the Extraction section below. As output it delivers an integrated dataset in VTK file format.

Since the data are represented as image data, i.e. as a rectilinear grid, each dataset's reconstruction plane is originally aligned along the xy-coordinate plane. Due to that the first step is to rotate them back into their original scanner coordinate system as specified in the header of the original DICOM image series. Subsequently all three datasets are resampled and interpolated to a rectilinear grid which provides high resolution in each of the three coordinate planes.

7.3.2.2 Extraction

In the original pipeline implementation the Extraction step was directly based on the clinical imaging datasets, which were first analyzed with respect to their content as well as their quality using 3D Slicer.

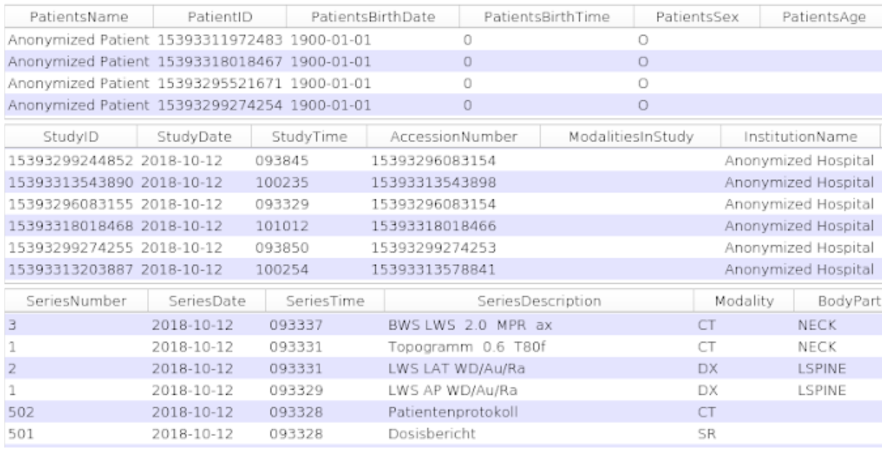
The datasets used for the developments were provided by the clinic for neurosurgery at the university medical center Knappschaftskrankenhaus Bochum, Germany as DICOM (Digital Imaging and Communications in Medicine) datasets. While there is no standard on how DICOM datasets have to be stored physically, they are organized logically by header information stored within each image. In Figure 7.13 it can be seen that the datasets are logically organized firstly by patient and secondly by so called studies, which in turn can contain several series.

Even though the datasets contain several imaging modalities like full body X-ray images, dual energy X-ray absorptiometry (DXA) scans, magnetic resonance imaging (MRI) data, as well as computer tomography (CT) images, we currently concentrate on the CT-images as they are the basis for the three dimensional (3D) reconstruction of the spine's bone structures.

The analysis of the datasets revealed that, for each patient, three studies contain 5-8 series with CT-data, which are recorded pre-operatively (without the implant), directly after surgery (with the implant), and after several weeks during clinical control examination. Further on, it was found that out of the 5-8 series, each study contains at least three CT-series each with a different reconstruction plane. The remaining CT-series contained different information like dose reports, or in some cases additional CT-series treated with different smoothing kernels.

The left subfigures of Figure 7.14 show an isosurface along with two cutting planes and resulting contour lines. As a basis for this feature extraction we took the CT-series from the pre-operative study, reconstructed in the x-y-plane. In the bottom-right subfigure the contour line of a vertebra in the x-y-plane is shown in detail. It can be seen that the vertebra's contour was extracted smoothly. In the top-right subfigure the contour line of the same vertebra, taken on the y-z cutting plane is shown. Here

[3] https://lorensen.github.io/VTKExamples/site/VTKFileFormats/



PatientsName	PatientID	PatientsBirthDate	PatientsBirthTime	PatientsSex	PatientsAge
Anonymized Patient	15393311972483	1900-01-01	0	O	
Anonymized Patient	15393318018467	1900-01-01	0	O	
Anonymized Patient	15393295521671	1900-01-01	0	O	
Anonymized Patient	15393299274254	1900-01-01	0	O	

StudyID	StudyDate	StudyTime	AccessionNumber	ModalitiesInStudy	InstitutionName
15393299244852	2018-10-12	093845	15393296083154		Anonymized Hospital
15393313543890	2018-10-12	100235	15393313543898		Anonymized Hospital
15393296083155	2018-10-12	093329	15393296083154		Anonymized Hospital
15393318018468	2018-10-12	101012	15393318018466		Anonymized Hospital
15393299274255	2018-10-12	093850	15393299274253		Anonymized Hospital
15393313203887	2018-10-12	100254	15393313578841		Anonymized Hospital

SeriesNumber	SeriesDate	SeriesTime	SeriesDescription	Modality	BodyPart
3	2018-10-12	093337	BWS LWS 2.0 MPR ax	CT	NECK
1	2018-10-12	093331	Topogramm 0.6 T80f	CT	NECK
2	2018-10-12	093331	LWS LAT WD/Au/Ra	DX	LSPINE
1	2018-10-12	093329	LWS AP WD/Au/Ra	DX	LSPINE
502	2018-10-12	093328	Patientenprotokoll	CT	
501	2018-10-12	093328	Dosisbericht	SR	

Fig. 7.13: Logical organization of patient datasets

it can be seen that the vertebra's contour could not be extracted smoothly due to the low resolution of this CT-series in the y-z- direction.

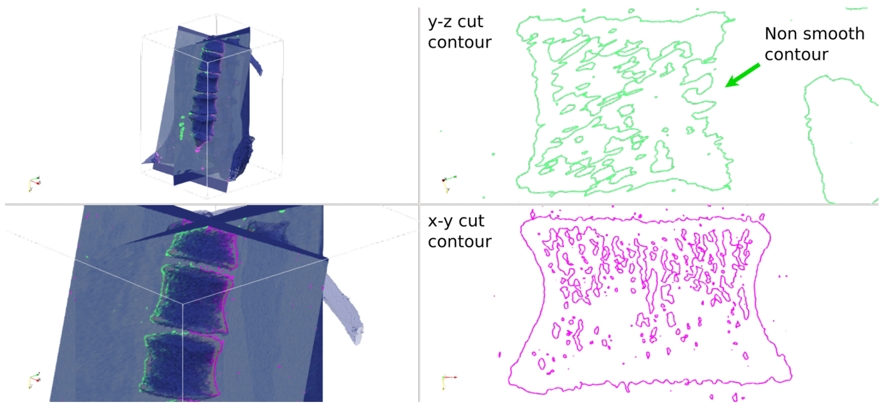


Fig. 7.14: y-z cut contour line and x-y cut contour line - x-y reconstructed dataset.

For the results shown in Figure 7.15 the same feature extraction as shown in Figure 7.14 was applied to the CT-series from the pre-operative study reconstructed in the y-z plane. In contrast to Figure 7.14, in the top right subfigure of Figure 7.15 it can be seen that the contour line on the y-z cutting plane in that case reproduces the vertebra's contour smoothly. In the lower right subfigure of Figure 7.15, however it can be seen that the contour in the x-y plane is no longer reproduced smoothly once the feature extraction is based on the CT-series reconstructed in the y-z plane.

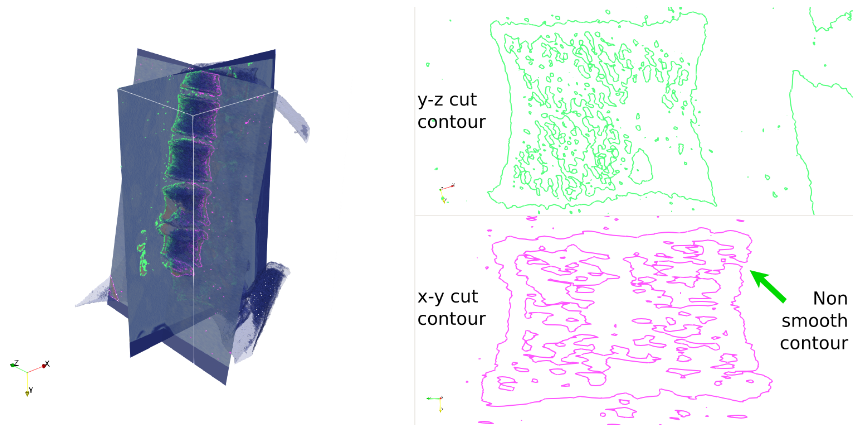


Fig. 7.15: y-z cut contour line and x-y cut contour line - y-z reconstructed dataset.

From these analyses it was concluded that exploiting only one CT-series for geometry extraction is not feasible and additional image processing steps have to be introduced as explained in the previous subsection.

7.3.2.3 Discretization

Based on the three surface meshes volume meshing of the bone geometries is performed. Targeted libraries for 3D volume mesh generation are currently Netgen or NGSolve. Additionally the boundary conditions i.e. the supports and the loadings have to be discretized and structural model features like muscle strands, tendons and cartilage have to be attached to the modeled bone geometries. As a result of this step the completed models will be written out as so called solver input decks still with a homogeneous material distribution.

7.3.2.4 Density Mapping

In this step, the three input decks as well as the three CT data sets are taken as input. By means of direct geometrical mapping, the grayscale distribution of the improved imaging data produced by the Image Processing and Filtering step is mapped onto the volume mesh provided by the discretization. After the mapping, each element in the volume mesh holds a density value. The Density Mapping step is written in

Fortran 2003. Its initial implementation was started in 2007 and is documented by Schneider [1]. The algorithmic principles are published by Schneider et al. [2].

Since the original implementation relied on an internal data format and was not able to load DICOM datasets with the given complex structure, the data input part had to be modified. 3D Slicer, which was used for the analysis of the datasets, has a very advanced DICOM reader implemented and is also able to export the loaded data as VTK STRUCTURED_POINTS[4]. Because of these features and since the Image Processing and Filtering stepis also based on VTK, it was decided to add the capability to load VTK STRUCTURED_POINTS data in VTK XML format to the Density Mapping component. By this extension, the Density Mapping step is now able to map the density distribution from the CT-data on the Meshed Geometry. This results basically in one density value per element.

To give an impression of the resulting density distribution, the final mapping result is visualized side by side with the original data in Figure 7.16. Since the probabilistic mapping step acts locally on each element, no topological information has to be passed on between the density mapping component and the probabilistic mapping component. Everything that has to be passed is a list that contains one integer value per element representing its averaged density value. These values have to be passed as a file containing binary 64-bit integer values.

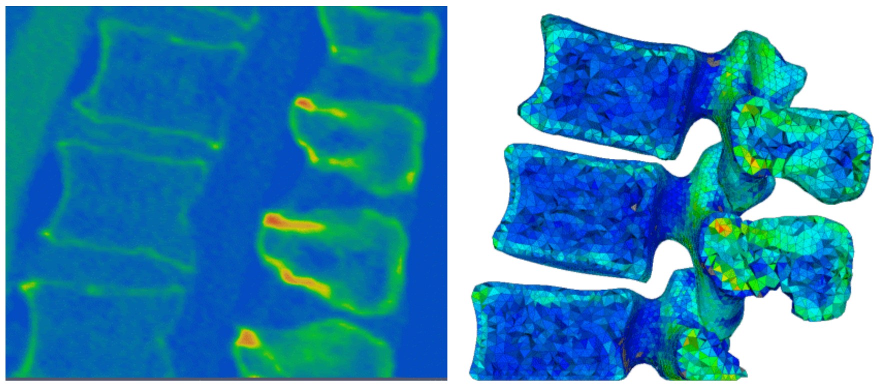


Fig. 7.16: Density Mapping component - Left: Input data - Right: Mapping result

7.3.2.5 Probabilistic Mapping

Based on the input from the Density Mapping component, the Probabilistic Mapping component produces three probability distributions per element, one for each of the

[4] https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf

three elastic moduli of an orthotropic stiffness matrix. This is done by means of the transfer functions between density and orthotropic elastic moduli given in Schneider et al. [2]. The low and high bounds and the mode of 95% confidence interval are computed from the aforementioned probability distributions. This finally results in the output of three files where each file contains three elastic moduli for each element.

Since these outputs have to be extended by shear moduli and poisson ratios to form complete, orthotropic material distributions and then become integrated with the Meshed Geometry to form a complete Input Deck, i.e. the Enhanced Meshed Geometry, the data path in the simulation process chain was changed from the original layout. Instead of treating the Enhanced Meshed Geometry directly with the Probabilistic Mapping component and passing its output directly to the Solver component, its output is now sent to the Applying Boundary Conditions component. This change in the data flow was decided during the parallel development of the Probabilistic Mapping component and the prototype of the Code_Aster[5] model. During the development, it was recognized that it would amount to less effort to add the integration of the results from the Probabilistic Mapping component to the Applying Boundary Conditions component than to implement the treatment of the Enhanced Meshed Geometry by the Probabilistic Mapping component. This was due to the fact that in the Density Mapping component, from which we derived the Applying Boundary Conditions component, the algorithmic part was already implemented and only the output part had to be changed.

7.3.2.6 Applying Boundary Conditions

The Applying Boundary Conditions component is derived from the original implementation of the Density Mapping component. The component integrates each of the three output files of the Probabilistic Mapping component with the Meshed Geometry in Code_Aster med file format. It generates three Enhanced Meshed Geometries in Code_Aster's med file format which include inhomogeneously distributed material Information needed by Code_Aster.

7.3.2.7 Solver

In the Solver component, a reference model prototype had been developed for ABAQUS and transferred to Code_Aster format. Initial results of this development are displayed in Figure 7.17.

Additionally, the coupling of the Applying Boundary Conditions component to the Solver component was implemented. The implementation of the coupling required basically the implementation of the mesh output in Code_Aster's HDF5-based MED-format and the implementation of the output of Code_Aster's input parameter file to the Applying Boundary Conditions component.

[5] https://www.code-aster.org/

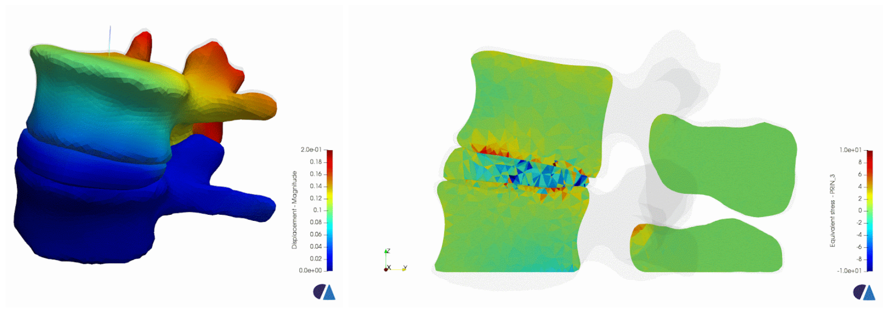


Fig. 7.17: Density Mapping component - Left: Input data - Right: Mapping result

7.3.3 How SODALITE is Used

The in-silico Clinical Trials use case reproduces real clinical trials in biomecanics by means of simulation to determine an optimal fixation and function of bone implant systems for patients with spinal conditions (e.g. disk displacement or prolapse). The use case is being developed by HLRS and was originally strictly HPC driven, i.e. it was realised as a workflow executed on a particular HPC infrastructure provider. This limits the adoption of the developed methodology in biomechanical clinical trials for medical device manufacturers or medical research institutes due to the specifics of the target execution environment of the use case.

Therefore, Clinical UC expects SODALITE to be beneficial in moving the process towards production-like environments with the following improvements:

1. Increase the effectiveness and productivity of component deployment.
2. Ease the adaptation to different IT-infrastructures (supercomputers, Clouds, on-premise) and different hardware.
3. Lower the efforts for component integration.
4. Lower the efforts for data management.

The rest of the subsection presents how SODALITE platform was used in the scope of Clinical UC.

7.3.3.1 Clinical UC Workflow Orchestration

SODALITE produced a set of reference TOSCA libraries and Ansible playbooks[6] for provisioning of VMs in Cloud (AWS EC2 and OpenStack) as well as execution of batch jobs on HPC clusters, managed by resource manager, such as PBS Torque and Slurm. Clinical UC validated these libraries by executing the workflow on the single and multiple HPC and Cloud targets using the SODALITE orchestrator. Additionally,

[6] https://github.com/SODALITE-EU/iac-modules

SODALITE developed TOSCA libraries for data movement with GridFTP, a data transfer protocol widely offered by HPC infrastructure providers, thus enabling multitarget workflow execution of Clinical UC components with data dependencies.

Clinical UC benefited from this development as it allowed the distribution of the workflow execution from a single infrastructure target into multiple targets, utilising the capabilities offered by various providers. For example, less capable but more available virtual resources can be used for UC components that do not demand a lot of computation resources, while bare-metal and more capable resources can be used for more compute-demanding tasks. Possible setups for the workflow execution of the Clinical UC are presented later in subsection 7.3.4.

7.3.3.2 Clinical UC Workflow Optimization

Together with optimization experts (Quality Experts), we focused on optimization of the Code_Aster Solver component, which is contributing the most to the total execution of the workflow. For Code_Aster, we compared the performance of the MODAK optimised container with that of the official Code_Aster container[7]. The experiment was performed on the MC partition of the Cray XC50 "Piz Daint" supercomputer at the Swiss National Supercomputing Centre (CSCS). Each node of the system is equipped with a dual-socket CPU Intel Xeon E5-2695 v4 @2.10GHz (18 cores/socket, 64/128 GB RAM). On a single thread, the MODAK containerised application had an average execution time of 725 seconds, while the official Code_Aster container executed for 745 seconds. This is a 3% speed-up.

Furthermore, with the build of parallel Code_Aster, additional optimised container images are being prepared for parallel execution of the Solver, which is proven to reduce the execution time significantly. The runtime tests were carried out on the HLRS system Vulcan using computing nodes with Intel Xeon Gold 6138 @2.0GHz Skylake processors and 192 GB of main memory, and different underlying solver libraries (MUMPS[8] and PETSc[9]) with and without METIS[10] partitioner were applied. The results of the runtime tests are given in Table 7.1.

Solver	**Partitioner**	**#-MPI-Procs**	**Walltime [s]**
MUMPS	-	4	654.52
MUMPS	-	8	442.66
MUMPS	METIS	8	226.87
PETSc	METIS	8	67.43

Table 7.1: Results of bare metal runtime tests of parallel Code_Aster

[7] https://github.com/codeaster/container

[8] http://mumps.enseeiht.fr/

[9] https://www.mcs.anl.gov/petsc/

[10] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

7.3.3.3 Clinical UC Workflow Modeling

SODALITE IDE offers a context-assistance and models validation that was useful during the modeling of the Clinical UC deployment. Moreover, SODALITE IDE offers optimization specification, which allows to create optimization recipes (with the help of Quality Experts) and apply optimizations to a particular application component, which in turn is executed using optimized container runtime.

Using SODALITE IDE, we modeled and deployed the single and multi-targets use case workflow executions on HLRS HPC testbed with the optimization applied to the Probabilistic Mapper for parallel execution, as described in subsection 7.3.4. Resource Experts have provided HPC resource models and models for workflow execution. Clinical UC developers acted as Application Ops Experts and created an AADM (Abstract Application Deployment Model) for the use case deployment, utilizing provided resource models. Quality Experts helped to develop optimization recipes to enable MPI pluralization for Probabilistic Mapper, during runtime of which, it was built and executed within optimized container runtime.

During the development of AADM for Clinical UC, SODALITE IDE assisted with modeling, e.g. offering available node types and resolving requirements for node templates. Inconsistencies in deployment models (e.g. mismatch of node types in requirements) were also checked and reported back instantly at the development time. Since initially we started deployment modeling using TOSCA with a simple YAML editor, we found the usage of the SODALITE IDE extremely convenient, error reducing and effort saving for failure resolution and component integration.

7.3.4 Deployment Architecture

The components of the Clinical Trials UC can be deployed with SODALITE in various Cloud platforms and HPC systems. We validated the following deployment cases:

1. Single target HPC system managed by either PBS Torque or Slurm
2. Multiple HPC targets with data transfers using GridFTP
3. Cloud and HPC targets with data transfers using GridFTP

The PBS Torque target is a bare-metal cluster and has MPICH as an MPI implementation, whereas the Slurm target is virtual (deployed via EGI EC3[11]) and uses OpenMPI as an MPI implementation. As Cloud target, OpenStack and AWS EC2 is used. Both HPC and Cloud targets are modelled in an AADM that is reused across different cases.

To ensure portability of application components deployed over these targets, Singularity container runtime was used. In case of HPC, the compatibility should also be respected at the level of communication between processes with MPI. As

[11] https://docs.egi.eu/users/cloud-compute/ec3/

such, the version and implementation variant of MPI (e.g. OpenMPI and MPICH) included in the container image must match with the MPI setup of the host. In the Clinical UC workflow, the Probabilistic Mapper component can be run in parallel with MPI. Optimization model is associated with this component, and this helps MODAK to select a container compatible with the MPI of the target host, i.e. either OpenMPI or MPICH will be selected, depending on what the target host supports.

In the first case, the partial workflow of Clinical UC is executed in a single HPC target, as depicted in Figure 7.18. The AADM for this case consists of definition of workflow components, executed in the specific order defined with *dependency* relationship between the components. To define where the component should be deployed, the *host* relationship between the component and the target is used. In this case, we specify either PBS Torque or Slurm cluster for all the components.

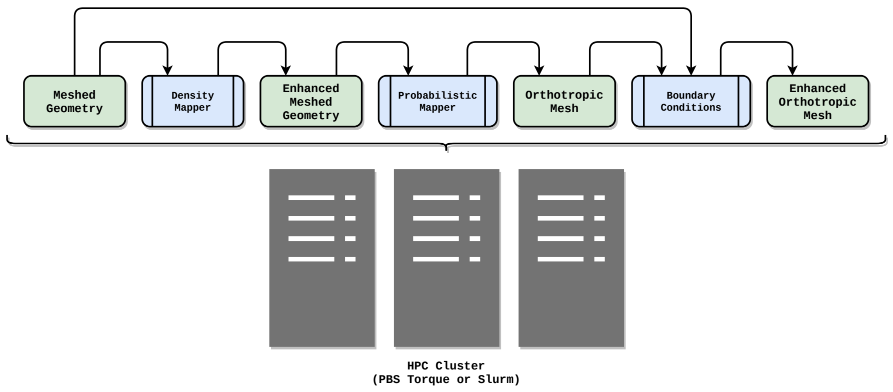


Fig. 7.18: Deployment of Clinical UC over a single HPC target, either PBS Torque (used in HLRS HPC Testbed) or Slurm (used in EGI EC3 Virtual Cluster)

In the second case, the partial workflow of Clinical UC is distributed across multiple HPC targets, as depicted in Figure 7.19. Density Mapper and Boundary Condition components are hosted in the Slurm cluster, whereas the Probabilistic Mapper is hosted in the PBS Torque cluster. Additionally, a GridFTP client is deployed in OpenStack, such that a 3rd party data transfer between the clusters is performed, thus allowing to transfer intermediate results of the workflow. Compared to the first case, the workflow order was extended by adding the data transfer steps between the component execution, to ensure that a dependent component will first receive the needed data from the previous step before its execution.

The third case, depicted in Figure 7.20 is similar to the second case with the difference that Density Mapper and Boundary Condition components are hosted in a VM (either AWS or OpenStack). The GridFTP client is co-located in the VM and moves the data to and from the HPC cluster after the respective component execution is finished.

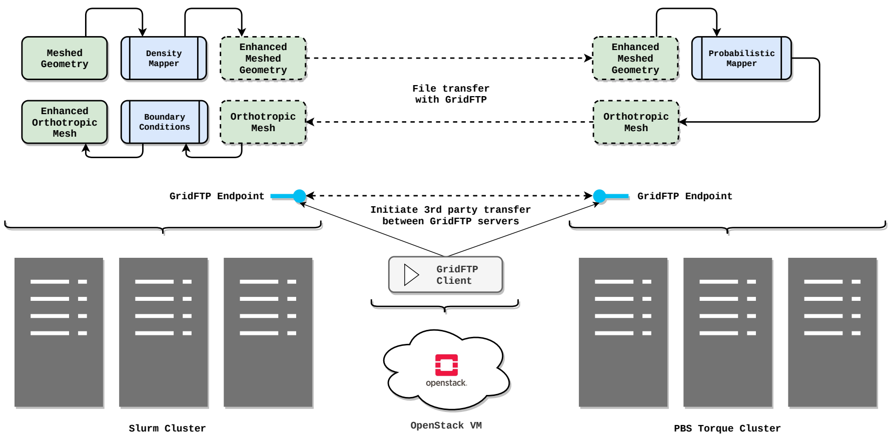


Fig. 7.19: Deployment of Clinical UC over multiple HPC targets with the third-party data transfers between the GridFTP servers, initiated by GridFTP client

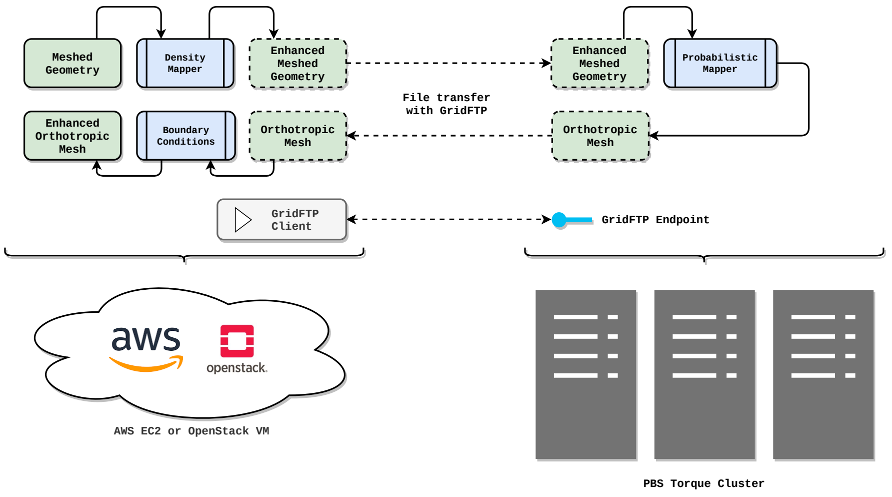


Fig. 7.20: Deployment of Clinical UC over Cloud and HPC targets with the data transfers between the GridFTP client and server

7.3.5 Benefits of Using SODALITE

The workflow of Clinical UC was executed with the SODALITE platform. Modelling, optimisation and orchestration aspects of the platform were validated and evaluated in the context of the Clinical UC, showing the use case improvements in the following:

1. Originally, Clinical UC workflow was HPC-driven and executed in a single HPC cluster. With SODALITE, the workflow execution is distributed across multiple infrastructure targets.
2. The execution of the Solver (Code_Aster) component of Clinical UC in a container was optimised. Furthermore, parallel build and execution became possible, promising further reduced execution time.
3. The workflow was modelled in SODALITE IDE, which reduced the effort for development of the deployment code, compared to TOSCA, as it became easier to integrate new components and avoid possible deployment errors. Additionally, optimised container runtime can be automatically provided based on the optimisation options specified in SODALITE IDE.

7.4 Vehicle IoT Use Case

This section describes the Vehicle IoT use case and how the usage of the SODALITE platform is beneficial for the use case execution. The use case is centered around the development of services useful to support the management of car fleets. As discussed in the following subsections, the use case is composed of a significant number of components and services that run on different operational environments. End-user apps are typically executed on mobile devices of various kinds. Special services with strict latency requirements are executed at the edge, either on special-purpose devices installed on the cars or at the periphery of the system. Specific tasks that require a significant amount of resources, such as those related to model training activities, are executed on HPC resources or on the cloud. In this context, SODALITE offers proper mechanisms to control the deployment, operation and reconfiguration of the whole Vehicle IoT system.

The following subsections elaborate more on the motivation behind the use case, describe the main components of the use case, and provide more details on how SODALITE was used to deploy the use case and on the achieved benefits.

7.4.1 Background and motivation

Through the combination of vehicle telemetry, instrumentation, and behavioural data, insurance companies are able to shape a more holistic view of an individual driver's overall risk profile based on empirical analysis of driving data (referred to as usage based-insurance, or UBI) - areas that have traditionally relied upon static data points over which the individual has little control, and which have been more focused on risk probability than empirical analysis (these factors include, e.g. age, gender, marital status, make/model of vehicle, etc.). While UBI models have been successfully engaged in markets with a more relaxed and homogeneous regulatory environment, European industry (and citizens) have been hesitant to pursue this model without adequate safeguards for personal data protection and privacy rights - a situation remedied in part by the coming into force of the GDPR (General Data Protection Regulation).

The growth of Connected Car data and concerns over data usage are further compounded by: (1) Individual expectation of contextualised service offerings that respect personal preferences and privacy expectations; (2) Service providers aiming to deploy service offerings across an increasingly dynamic environment; and (3) growing trend of drivers seeking to analyse and benefit from their own driving data directly.

These growing expectations, both from individuals and businesses, lead to an enormous increase in the volume and rate of the sensor data, its aggregation, and its analysis, at various hierarchical levels. This data, in turn, must be processed in line with the relevant privacy constraints and regulatory restrictions it is subject to - aspects subject to dynamic change, while also being highly latency-sensitive.

This leads to two key architectural demands: (1) an increasing amount of in-vehicle data processing and intelligence at the network edge, and (2) increased computational capacity to process large amounts of data in a timely manner - at varying levels of granularity (e.g. device-local, vehicle-local, fleet-wide) - including both fleet-wide big data analytics, as well as periodic online retraining of machine learning models that support the deployment.

Thanks to this case study, it is possible to demonstrate the ability of SODALITE to deploy applications through the whole continuum from the cloud to the edge, to refactor the deployment depending on the system configuration at runtime and to ensure privacy preservation.

7.4.2 The Vehicle IoT case architecture

Figure 7.21 presents the structure of the VehicleIoT system. It includes end-user apps running on IoT devices such as smartphones and watches, a special-purpose device installed on cars and a number of components at the edge that enables the connection with specific microservices, including License Plate Recognition, Reverse Geocoding, and Drowsiness Detection. Many of these microservices, in turn, leverage trained machine learning models, and are able to quickly provide results with minimal computational overhead, providing the opportunity to re-deploy and run these services at different hierarchical levels (backend, in-vehicle edge gateway, smartphone, etc.).

The edge part of this case study is managed through Kubernetes and is assumed to be located partly on each car and partly on the premises of some fleet management company, as show in Figure 7.22.

The following subsections present in detail the most relevant features offered by the system.

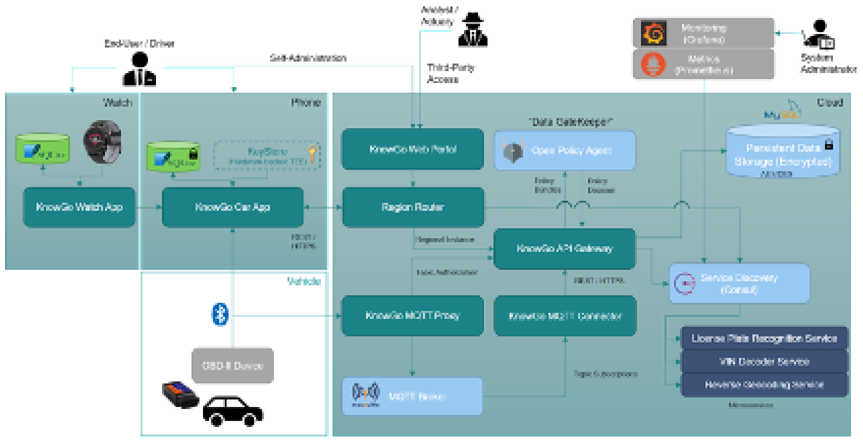


Fig. 7.21: Overview of the VehicleIoT architecture.

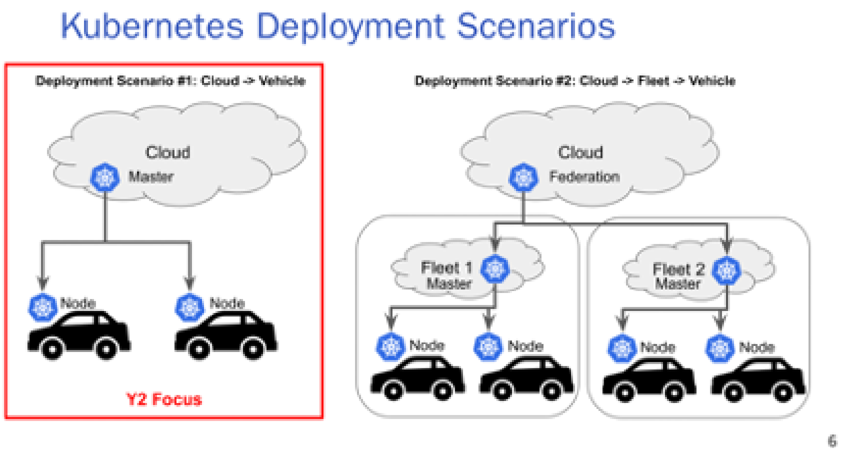

Kubernetes Deployment Scenarios
Deployment Scenario #1: Cloud -> Vehicle
Cloud
Master
Node
Node
Y2 Focus
Deployment Scenario #2: Cloud -> Fleet -> Vehicle
Cloud
Federation
Fleet 1 Master
Fleet 2 Master
Node
Node
Node
Node
6

Fig. 7.22: The VehicleIoT edge part configuration.

7.4.2.1 License Plate Detection

Within the Vehicle IoT Use Case, individuals may, at various times, submit license plate images for recognition. This can be done for various purposes, including the initial registration of the vehicle with the mobile app (as one possible registration mechanism - of particular interest in countries which provide open access to their vehicle registration databases), evidence to support claims preparation (in the case of a collision), etc. In order to benefit from improved plate recognition, the license plate recognition service relies on a machine learning model that is trained by leveraging appropriate resources (Cloud or HPC). This is envisioned across a number of steps:

- Inclusion of user-generated images in the training data set;
- Plate extraction from uncropped training data (Bulk processing);
- Re-training model on suitable backend resource (e.g. GPU cluster);
- Validating control set against the new model (regression detection);
- Re-deployment / update of plate recognition microservice backed by the new model.

This process is summarised in Figure 7.23.

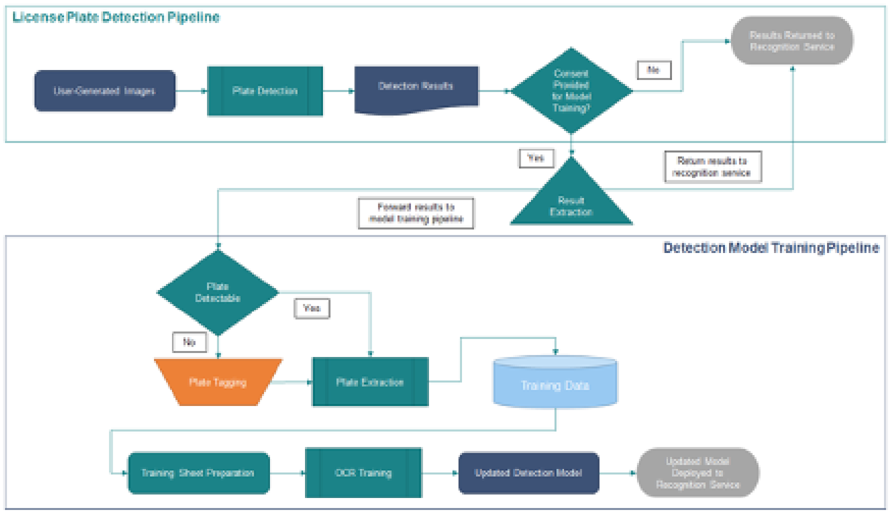

License Plate Detection Pipeline
User-Generated Images
Plate Detection
Detection Results
Consent Provided for Model Training?
No
Results Returned to Recognition Service
Yes
Return results to recognition service
Result Extraction
Forward results to model training pipeline
Detection Model Training Pipeline
Plate Detectable
Yes
No
Plate Tagging
Plate Extraction
Training Data
Training Sheet Preparation
OCR Training
Updated Detection Model
Updated Model Deployed to Recognition Service

Fig. 7.23: The License Plate Recognition Pipeline.

7.4.2.2 Drowsiness Detection

Drowsiness Detection aims to determine when a Driver is at risk of falling asleep at the wheel and taking evasive actions (e.g., playing a loud noise, triggering a vibration, etc.) in order to alert the driver to the problem before a more serious incident occurs. Drowsiness detection is typically carried out using a couple of different methods, with differing levels of accuracy and invasiveness. While the gold standard (and most accurate method) for drowsiness detection remains ECG monitoring, ECG measurement itself is invasive and requires active participation by the individual under monitoring, making it a poor fit for passive observation of a driver. The most common non-invasive methods, on the other hand, are PERCLOS (Percentage of eyelid closure) - measuring the proportion of time that the eyelids are between 80-100% closed, and blink detection (Blink detection methods further being split between blink frequency and duration detection). While the PERCLOS method is fairly well established, it has also been found to generate false positives in scenarios that are compatible with the driving activity. For this reason, blink detection is investigated in this case.

It is important to note that drowsiness detection is highly latency-sensitive and must be done in real-time in order to be as accurate as possible and to alert the driver at the time they need to be alerted. Blink duration can be summarized as awake (< 400ms blink duration), drowsy (400-800ms blink duration) and sleepy (blink duration > 800ms).

With current wireless technologies demonstrating round-trip latencies near 50-200ms (for 4G) and 500ms (for 3G) with good connectivity, a backend-deployed monitoring service cannot be expected to reliably identify and respond to drowsiness events in time - necessitating a push-down of the service delivery to the Vehicle itself.

7.4.2.3 Intrusion and Theft Detection (Face Recognition)

Intrusion detection builds on the face detection model developed in the drowsiness detector and defines a face recognition model capable of identifying the authorized driver's face. In the case where someone other than the designated driver is found to be driving the vehicle, further actions can be taken by the system. These may include notifying the authorized driver and seeking confirmation of a driver switch, notifying a fleet manager, streaming vehicle telemetry to a third party, etc.).

In contrast with drowsiness detection, intrusion and theft detection is not directly latency-sensitive, and as it does not require real-time access to the driver's video stream, is suitable for backend deployment as a long-lived microservice. While the authorized driver may indeed wish to know if someone is stealing their vehicle as quickly as possible, the added round-trip latency associated with mobile communications is unlikely to have a measurable impact on any asynchronous notifications that may result from the analysis.

A unique characteristic of this service is that custom classifiers must be modelled and trained in order to provide value for the Driver (that is, SVC models capable of identifying the authorized Driver's face - which the driver may take with them). This may involve dynamic training of vehicle-restricted classification models or may be open for sharing across a fleet of vehicles, or any other vehicle the end-user may use, dependent upon their individual privacy preferences and sharing settings.

Notably, the infrequent nature of the invocation also makes this an ideal candidate for serverless deployments.

7.4.3 How SODALITE is Used

The vehicle IoT use case makes effective use of the SODALITE platform.

- SODALITE IDE and deployment workflow is used to automate the deployment of the application over the cloud and edge infrastructure.
- Kubernetes support is used to model the Edge cluster and to deploy the services on the Edge nodes.
- Platform discovery service is used to discover the changes to the Edge cluster, for example, appearing and disappearing of Edge nodes (CPU and GPU), and to atomically update the knowledgebase to reflect the current state of the Edge testbed.
- Image Builder is employed to create the container images of the service variants optimized for specific Edge accelerator hardware devices.
- The monitoring and refactoring components are employed to collect various metrics from Edge devices, to generate alerts based on the collected metrics, and to perform the necessary changes to the application deployment at runtime.

7.4.3.1 Deployment Architecture

Figure 7.24 shows the deployment of the vehicle IoT case study in the SODALITE cloud and Edge testbeds. Some services (e.g., drowsiness detector, the MySQL storage, and the reverse geocoder) are deployed on edge nodes, while other services (e.g., region router and echo services) are deployed on cloud VMs. Three edge devices are Raspberry Pi 4, Google Coral AI Dev Board, and NVIDIA Jetson Xavier NX. Their accelerators are NCS2 (Neural Compute Stick 2), EdgeTPU, and NVDLA x2.

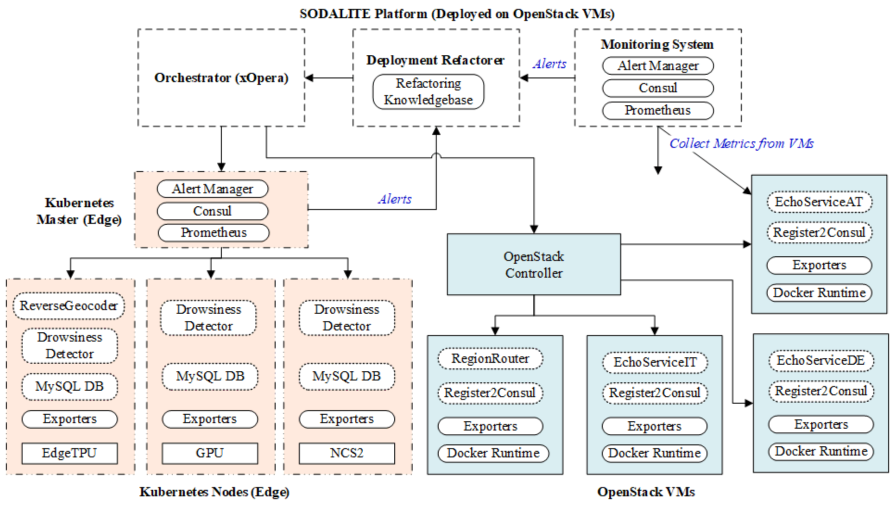


Fig. 7.24: Vehicle IoT case study deployment architecture

```
rule "de_to_it"
  when
    $f1 : LocationChangedEvent(preLoc == "de", cLoc == "it") and
    $f2 : DeploymentChanged()
  then
    Node cloudVM = refMgt.findNode("( ?location = "it" )");
    AADMModel aadmModel = refMgt.getAadm();
    aadmModel.replaceNode("vehicle_iot_vm", cloudVM);
    refMgt.saveAndUpdate();
end

rule "edgetpu_temp_critical"
  when
    $f1 : Alert(name == "TempCritical")
  then
    Node edgeNode = refMgt.findNode(
                          "(?edgetpus = 1 ) && (?variant = "std")");
    AADMModel aadmModel = refMgt.getAadm();
    aadmModel.replaceNode("vehicle_iot_edge_node", edgeNode);
    refMgt.saveAndUpdate();
end
```

List 7.3: A snippet of the deployment adaptation rules used by the vehicle IoT case study

7.4.3.2 Monitoring and Refactoring

The vehicle IoT use case employs the SODALITE runtime layer to implement two deployment adaptation scenarios: location-aware redeployment and alert-driven redeployment. In the first scenario, the application deployment is adapted in response to changes in legal jurisdiction, helping the application to maintain both service continuity and meet its compliance requirements as vehicles travel between countries. The second scenario uses edge-based monitoring and alerting to throttle an application deployment that has exceeded thermal tolerances to mitigate the risks of rising temperature inducing inference failure. The EdgeTPU run-time libraries are provided in *-max* and *-std* versions. Using the image builder, the two different variants of the inference application containers, each linked against one version of the run-time library, using an appropriate accelerator-specific base container were created. The deployment refactorer can switch between these two variants as thermal trip points of the devices change.

List 7.3 shows two adaptation rules used by the case study. The first rule reacts to the event *LocationChangedEvent* by replacing the hosting VM and the second rule reacts to the alert *TempCritical* by replacing the hosting edge node. The predicates over the TOSCA node properties, for example, *location* and *variant* are used to find the correct TOSCA node templates.

7.4.4 Benefits of using SODALITE

The SODALITE platform improved the vehicle IoT use case in the following aspects:

- The ability of the SODALITE platform to seamlessly orchestrate applications over hybrid heterogeneous infrastructure eased the deployment of the vehicle IoT application. The SODALITE IDE and the guidance provided by the SODALITE Knowledge Base simplified the development of the deployment model for the application.
- The image builder and the application optimization components helped to utilize heterogeneous edge devices by enabling creating device-specific variants of application services.
- The SODALITE runtime improved the responsiveness of the vehicle IoT application by enabling monitoring of the application and the hosting infrastructure resources and codifying and enforcing of deployment adaptation decisions.

7.5 Conclusion

This chapter demonstrated the usage and benefits of SODALITE in the context of the three case studies, each of which required a specific execution and operational environment in Cloud, HPC and Edge platforms.

As such, using SODALITE platform, Snow use case has successfully managed the lifecycle of its execution pipeline in private Cloud, trained its deep learning component in HPC resources using optimized container runtime and utilized runtime optimization of resource allocation. Clinical use case managed to extend its workflow execution from a single HPC infrastructure to multiple HPC and Cloud platforms and utilize optimized container runtimes. Vehicle IoT use case leveraged SODALITE to deploy its application components in Cloud and Edge infrastructures, to build container image variants optimized for specific Edge accelerator hardware devices and to handle dynamics of the Edge infrastructure, such as appearance of new hardware in edge nodes.

The experiments performed by the case studies demonstrate that SODALITE is able to handle complex use cases in various domains and execution platforms and achieves its objectives to enable simpler and faster development of IaC and deployment and execution of heterogeneous apps in HPC, Cloud and other software defined computing environments.

References

[1] R Schneider. “Modellierung des inhomogenen orthotropen Materialverhaltens der kortikalen Femurstruktur auf der Basis klinischer CT-bzw. Dichte-Daten”.

PhD thesis. Thesis, Institute of Mechanics, Structural Analysis, and Dynamics of Aerospace Structures (ISD), University of Stuttgart, 2007.

[2] Ralf Schneider et al. “Inhomogeneous, orthotropic material model for the cortical structure of long bones modelled on the basis of clinical CT or density data”. In: *Computer Methods in Applied Mechanics and Engineering* 198.27-29 (2009), pp. 2167–2174.

Chapter 8
Toward Impact Generation and Future Research

Joao Pita Costa, Elisabetta Di Nitto and Indika Kumara

Abstract This book has presented the main results developed by the SODALITE project, some examples of models an end-user can develop for the purpose of exploiting the SODALITE deployment and operation features, and the three specific case studies that have been used as first experimental playground for the approach. In this chapter we wrap up by first providing an overview of how the SODALITE results are being packaged in ready-to-use tools and then discussing about future research challenges.

8.1 Impact Generation: the SODALITE stack

In order to determine the scope of the SODALITE technology from the perspective of the usefulness of its inventions, i.e., its innovation, we organized the SODALITE results as a stack of five innovation layers that highlight the multidimensional capability of this software toolset (see Figure 8.1 for illustration and [1] for the formal definition of software stack that we follow). All layers shown in the figure can either be used in combination with each other or they can be adopted in isolation and exploited in any TOSCA-based context.

The first innovation layer includes all elements that constitute the `SMART IDE` (see Chapter 3) and allow end users to define Abstract Application Deployment Models and to transform them into executable IaC.

Joao Pita Costa
XLAB, Ljubljana, Slovenia e-mail: `joao.pitacosta@xlab.si`

Elisabetta Di Nitto
Politecnico di Milano, Italy, e-mail: `elisabetta.dinitto@polimi.it`

Indika Kumara
Jheronimus Academy of Data Science, Tilburg University, The Netherlands e-mail: `i.p.k.weerasinghadewage@tilburguniversity.edu`

The second innovation layer, `FindIaCBug` offers the verification and bug prediction features (see Chapter 4, Sections 4.2 and 4.3).

The third and fourth innovation layers, `MOORING` and `REFIT`, focus, respectively, on orchestration of deployment (see Chapter 5.2) and monitoring and reconfiguration (see the rest of Chapter 5).

Finally, the last innovation layer includes `POET`, which encompasses the performance optimization mechanisms available at design time (Chapter 4, Section 4.4) and at runtime (Chapter 5, Section 5.6).

All above layers are sustained and supported by four pillars that constitute the main structure of the whole SODALITE solution. These concern the usage of semantic and AI technologies (the semantic intelligence pillar), the attention to Quality of Service, the focus on enabling the usage of heterogeneous infrastructures and, finally, the emphasis on security and privacy. For space reason, this last point has not been described in detail in this book. On the one side, it concerns the adoption of state of the art approaches to access and secret management, security vulnerability scanning, and code inspection. On the other side, it includes the attention to the quality and security issues in IaC scripts generated from the deployment models created by the users through the Smart IDE (see Chapter 4).

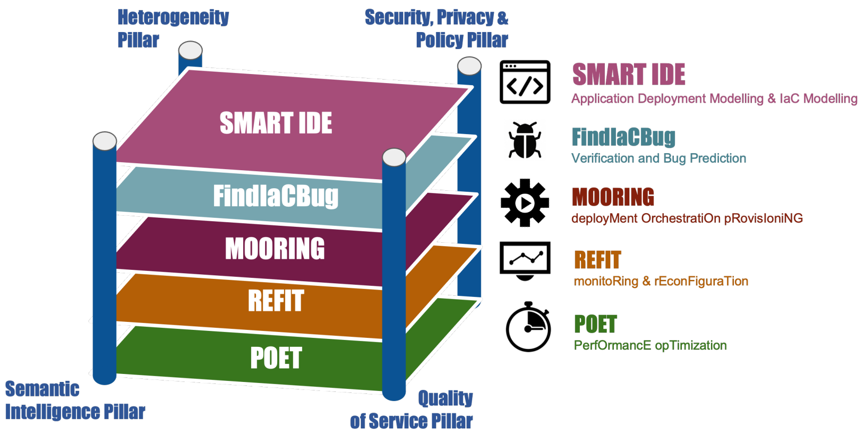

Heterogeneity Pillar
Security, Privacy & Policy Pillar
SMART IDE
FindIaCBug
MOORING
REFIT
POET
Semantic Intelligence Pillar
Quality of Service Pillar
SMART IDE
Application Deployment Modelling & IaC Modelling
FindIaCBug
Verification and Bug Prediction
MOORING
deployMent OrchestratiOn pRovisIoniNG
REFIT
monitoRing & rEconFiguraTion
POET
PerfOrmancE opTimization

Fig. 8.1: Layered presentation of the SODALITE stack

8.2 Future Research

SODALITE represents one of the first attempts to tackle the problem of simplifying the work of those who need to automate the deployment and operation of complex software. In particular, SODALITE has tried to address the problem from multiple perspectives, from the introduction of smart editing features to the usage of ontologies and machine learning to infer possible bugs as well as possible code completion. Problems such as the automated generation of execution containers and the identification of the optimal ones given a specific execution environment have been addressed as well, together with the automated generation of models for the discovered resources and the automated identification of possible refactoring options.

The solutions we have developed show that all above problems can be addressed. There are though a number of challenges that are still to be addressed. Among the others, we highlight the following ones.

- *Single modelling approach and compatibility with multiple IaC languages*: even if SODALITE offers modelling support to the definition of application deployment models, still the user does not face a single modelling paradigm, but he/she should be able to handle the usage of multiple Domain Specific Languages that are heterogeneous in terms of their characteristics. A single paradigm for specifying application deployment would certainly help the users and facilitate their work. If the result of the modelling phase could be translated into multiple languages, the approach could be adopted by users of different orchestration frameworks (e.g., TOSCA and Terraform).
- *Deployment and operation frameworks for the cloud continuum*: SODALITE has considered HPC, Cloud and edge resources. A natural continuation of the project would be to manage heterogeneous resources as a seamless continuum where components could be dynamically deployed on the most suitable resource types and could be moved in the continuum if the situation in terms of resource consumption and application-level requirements evolves.

Both aforementioned areas are becoming the focus of new projects and initiatives (see, for instance, the H2020 PIACERE project[1]). We do hope that the findings, experience, and concrete results we have achieved in SODALITE will be beneficial for them and, possibly, reused and extended.

References

[1] J. J. Hack N. Heinonen and M. E. Papka. “Software Stack in a Snapshot”. In: *Comput. Sci. Eng.* 198.21 (2019), pp. 114–116.

[1] https://www.piacere-project.eu/