Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

IaC Management - initial version

D4.1

XLAB

31.7.2020

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

Deliverable data			
Deliverable	IaC management - Initial version		
Authors	Dragan Radolović (XLAB) Nejc Bat (XLAB) Elisabetta Di Nitto (POLIMI) Mehrnoosh Askarpour (POLIMI) Karthee Sivalingam (CRAY) Indika Kumara (JADS/UVT) Panagiotis Mhtzias, Georgios Meditskos (CERTH) Kalman Meth (IBM)		
Reviewers	Mario Martínez Requena (ATOS) Anastasios Karakostas (CERTH)		
Dissemination level	Public		
History of changes	Dragan Radolović (XLAB)	Outline created	12.9.2019
	All	Partner contributions	30.12.2019
	All	Additional partner contributions	9.1.2020
	All	Reactions to comments of first review	22.1.2020
	All	Corrections made according to review suggestions and resubmission into internal review	24.1.2020
	Nejc Bat (XLAB)	Final version and preparation for submission	29.1.2020
	All	Revised based on the feedback: - Literature review added - Architecture section restructured - Components matched with background technologies and the	10.7.2020

		UCs that they support - Motivation for development of individual component added - Document structure reordered - Overall text adjustments	
	Nejc Bat (XLAB)	Prepared for internal QA	13.7.2020
	Nejc Bat (XLAB)	Finalised and passed through internal QA check, prepared for re-submission	31.7.2020

Acknowledgement

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

Table of Contents

List of figures

Executive Summary

This deliverable presents the status of development of the Infrastructure as Code (IaC) layer within the SODALITE platform, as well as the integration with other components and tools in SODALITE platform at the end of the first year of the project.

The main focus is to cover the background, research and development progress of the SODALITE project in the field of IaC management. The document first describes the underlying technologies used in the production of the SODALITE system. It then continues to describe the Infrastructure Management Support that takes care of the building of the application images. Later, the Deployment Preparation and Optimization procedures are presented. The document concludes with the presentation of the Analytic and Semantic Decision support system that is responsible for the bug prediction and refinement of the deployed images. Overall, the report aims at describing the deployment preparation process and performance optimisation tasks with a preliminary definition of predictive and corrective analysis of the quality of infrastructure-as-code produced before deployment.

Three iterations of this deliverable are planned, one at the end of each year of the project. By the end of Year 1, an initial implementation of the basic components making the SODALITE platform has been set up. During Year 2 progress in integration of the components are expected, more advanced features, and initial evaluation of the improvement provided by the SODALITE platform. In Year 3 the consortium expects to have a framework of tools able to model, provision, optimise and deploy applications on heterogeneous environments as the final result of the development of the SODALITE platform.

Glossary

Acronym	Explanation
AI	Artificial Intelligence
AOE	Application Ops Expert The equivalent process from the ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes is Operation processes and maintenance processes
API	Application Program Interface
CPU	Central Processing Unit
CRESTA	Collaborative Research into Exascale Systemware, Tools & Applications
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
ETL	Extract, Transform, Load
GPU	Graphical Processing Unit
FPGA	Field-Programmable Gate Array
HPC	High Performance Computing
IaC	Infrastructure as Code
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MAMBA	Managed Abstract Memory Arrays
M2T	Model-to-Text
OASIS	Organization for the Advancement of Structured Information Standards
QE	Quality Expert The equivalent process from ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes: Infrastructure management and Configuration management processes
QoS	Quality of Service
RDF	Resource Description Framework
RE	Resource Expert The equivalent process from ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes is Quality Management and Quality assurance processes
REST	Representational State Transfer
SHACL	Shapes Constraint Language

SSH	Secure Shell
TOSCA	Topology and Orchestration Specification for Cloud Applications
TLS	Transport Layer Security
UDJ	Universal Data Junction
UML	Unified Modeling Language

1 Introduction

While more and more organizations and institutions embrace the cloud for infrastructure provisioning and deployment of applications, the need to manage such complex structures through different services represent a real challenge for developers and IT operations teams.

The idea to manage these services through Infrastructure as Code (IaC) approach is effective and already acknowledged by an ever-growing community of users and contributors. Since the field of managing infrastructure through IaC is in an early growth phase there are several different languages usually bound to service implementation. SODALITE implements and uses IaC relying on existing standards and best practices.

The objective of this deliverable is to present the plan of the Consortium regarding the development of the SODALITE platform with focus on the Infrastructure as Code (IaC) management, that will serve as the foundation for the development and implementation of the three SODALITE demonstrating use cases. To this end, this document provides a description of the resources needed to achieve the components' functionality that will be developed within SODALITE and of the platform as a whole, as well as a report on the plans and status of each component coupled with information about their practical implementation.

1.1 Structure of the document

This deliverable is structured as follows:

- The remainder of the Introductory section explains the positioning of IaC management in the SODALITE project and reviews the envisioned components for IaC management and their relations to other parts of SODALITE component models. This material adds a level of detail in the functional description that highlights parts of IaC Management from previous deliverables D2.1 "Requirements, KPIs, evaluation plan and architecture - First version".
- Section 2 provides a description of the existing technologies that are used for the development of the platform's components regarding the IaC management. It also depicts the relationships with other components from the SODALITE platform which are used or use IaC management components in their workflows. This section also provides a short introduction to sections 3, 4, 5 and 6 which are concrete descriptions of tasks to be resolved within Work package 4 - IaC Management.
- Section 3 presents the *IaC Management and support,* which is basically the cornerstone for building and assembling application deployment in work package building blocks.
- Section 4 features the methods and tools used to parse the abstract model definition, build the IaC blueprint and the actuation scripts which are going to be deployed through the orchestrator.
- Section 5 describes and tools used to optimize, test and model the runtime image for performance assuring that the deployed runtime provides the optimal execution.
- Section 6 introduces the approach and presents the tools for semantic validation of IaC elements used to prepare the deployment.
- Finally, Section 7 provides concluding remarks and a wrap-up of the document.

2 Description of Technology Stack

As explained in the Introduction section, the currently envisaged components that make up the first iteration of the SODALITE platform are summarized above and described in detail in deliverable D2.1 under the Architecture section. This section describes the technologies that are used to implement some of the components. Some of the technologies will be further improved with the features that might be found necessary for the implementation of the SODALITE platform and tools. It should be noted that these technologies were selected based on the consortium partners' expertise, as well as the potential to further uptake the work in several tools/technologies that were developed as part of past European projects or initiatives, in which the consortium partners have been involved.

2.1 WP4 baseline technology stack

WP4 covers the aspects of IaC (Infrastructure as Code) within the SODALITE project. Abstracting models and representing them in a simple and understandable way has been a long pursued task for different fields of computing and Infrastructure and Application deployment modelling are the one targeted by the SODALITE project.

2.1.1 TOSCA

TOSCA[1] standard was created with the purpose to support Cloud information models, enabling extensions of the concepts defined by node types or inheritance and thus providing an extensive level of abstraction. The sheer variety of Cloud services, their resource definitions and management are challenging for cloud actors since various and non-standard interfaces are provided to deal with these components.

Additionally, the provided descriptions for cloud resources and services may be ambiguous, mainly due to allowing different semantics to address the same concepts. In this context, conflicts and incompatibilities between cloud services take place, increasing the need for human interactions. Moreover, the lack of semantics represents a key issue contributing to the emergence of such challenges. TOSCA resolves these issues by adding common and formal descriptions of cloud services and resources, thus providing the necessary levels of abstractions and interoperability, enabling the development of intelligent discovery, matchmaking and composition algorithms to ease the development of software components, and their testing, deployment and management.

SODALITE uses these extendible TOSCA concepts to describe the infrastructure resources, application deployment topology definitions as nodes, relationships between those nodes, policies etc.

Since TOSCA is implementation agnostic, meaning that node lifecycle operation implementation can be done using one of the low-level or high-level programming languages from bash scripts, python to infrastructure management tools.

2.1.2 Ansible Actuation

As infrastructure provisioning, application deployment and configuration is usually the cornerstone of DevOps operations, the selection of a high level infrastructure management tool already known and adopted by the DevOps community, like Puppet[2], Chef [3]or Ansible[4], seems a natural choice for having the maximum impact on these communities.

Both Chef and Puppet are open-source, mostly designed to help DevOps configure and manage the infrastructure. While they both have a large number of already tested and verified code repositories, both Chef and Puppet have been architectured as an Agent-Master solution and thus need agents installed on each node for configuration, which complicates the deployment scenarios. The DSL is Ruby-like which is also usually considered more difficult to learn.

Ansible is a provisioning, configuration and application management tool open-sourced by RedHat. The highlight points are a vast community support and probably the largest set of cloud infrastructure libraries support (Ansible Galaxy). A simple and clean, declarative YAML DSL, widely accepted as easy to learn and adopt. Ansible's inherently simple agentless approach to remote infrastructure management is implemented through the standard python paramiko SSH library enabling the DevOps to manage any infrastructure accessible through SSH. Ansible is sequential in its execution making it a bit slow on scaling, but SODALITE tries to overcome this with the implementation of parallelization of TOSCA deployments

Figure 1 shows a comparison of interest and questions posted to the known Stack Overflow developer open community platform regarding the main contendants in the field of Automation and configuration IaC for cloud.

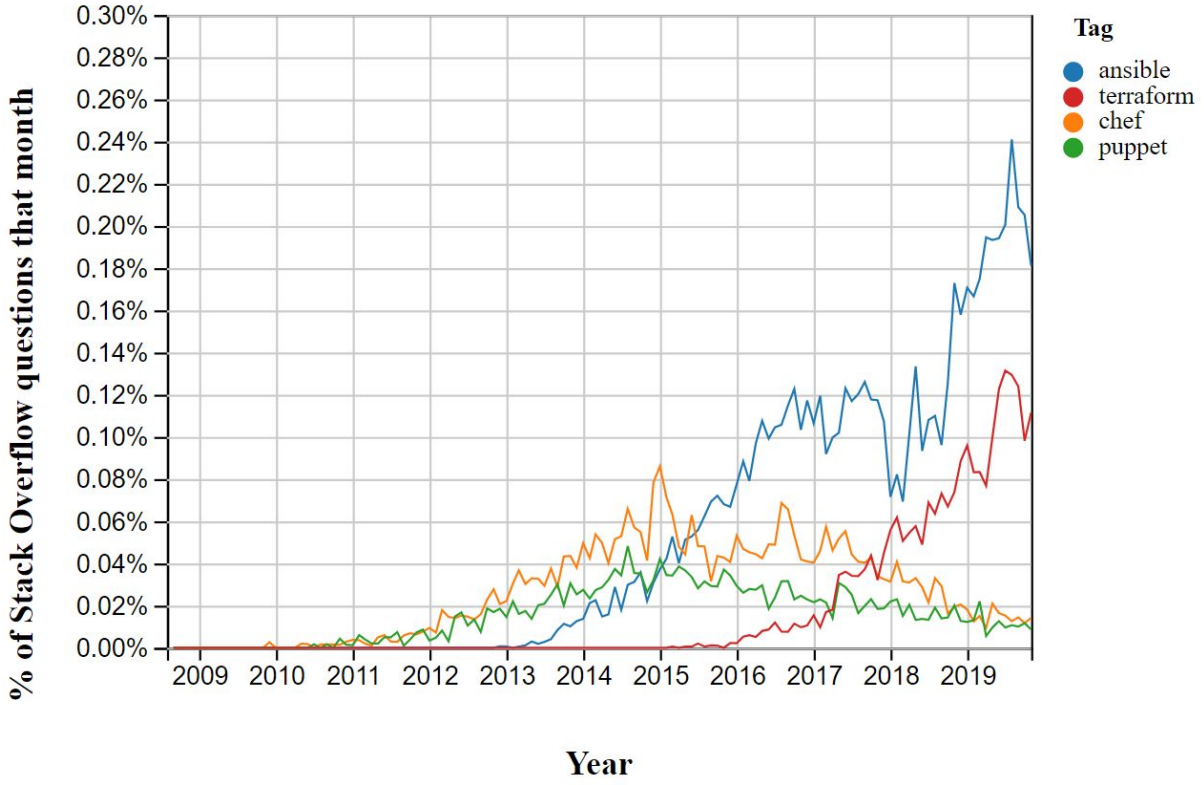

Figure 1 - Source Stack Overflow Trends: Comparison between main cloud management Automation & Configuration contendants

SODALITE provides initial node modelling through TOSCA and configurable Ansible roles and playbooks as part of the *Infrastructure Management Support*, thus creating a repository of predefined actuation scripts used by the orchestrator to deploy, start and monitor application artifacts. A decision has to be made about whether Chef is to be used as well. This technology may be used for creating the deployment artifact images by the SODALITE *Deployment Preparation* package and is used by the SODALITE *Orchestrator* as a deployment actuation tool. SODALITE will provide Ansible collections to support different platforms and execution environments with parameterization and optimization in mind.

2.2 Overview of SODALITE architecture

Here we present a short synopsis of the SODALITE architecture. This has already been described in deliverable D2.1 in the architecture section (Section 3). For full details, check the functional description, inputs, outputs and dependencies of each component.

SODALITE aims to provide developers and infrastructure operators with tools that abstract their application and infrastructure requirements to enable simpler and faster development, deployment, operation and execution of heterogeneous applications (e.g. containers, microservices and HPC jobs running in conjunction) on heterogeneous, software-defined, high-performance, cloud infrastructures. To this end, SODALITE aims to produce:

- a pattern-based abstraction library that includes application, infrastructure and performance abstractions,
- a design and programming model for both full-stack applications and infrastructures based on the abstraction library,
- a deployment framework that enables the static optimization of abstracted applications onto specific infrastructure,
- automated run-time optimization and management of applications.

The SODALITE platform is divided into three main layers, each implemented in a separate work package. These layers are the Modelling layer (WP3), the Infrastructure as Code layer (WP4), and the Runtime layer (WP5). Figure 2 below shows these layers together with their relationships.

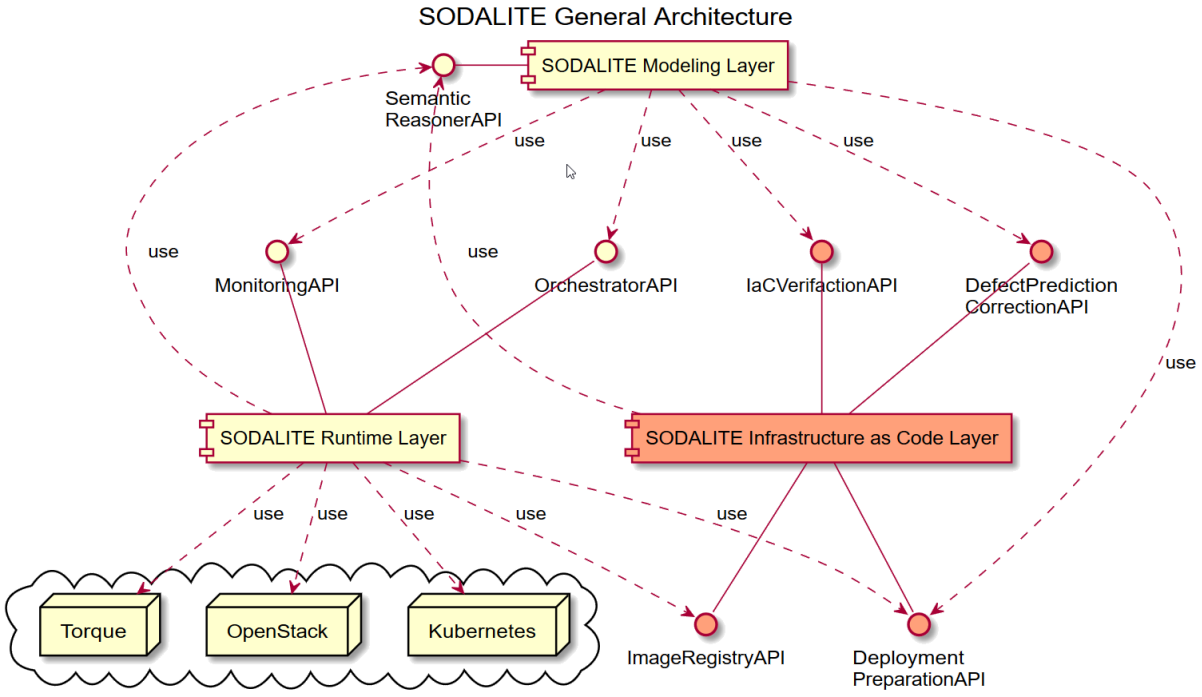

Figure 2 - SODALITE overall Architecture

2.3 WP4 layer description

The main task of the IaC layer is to take the modelling information provided by the SODALITE IDE (WP3) and produce a working error free IaC blueprint. Deployment Preparation involves a number of operations to build an IaC blueprint. These operations are handled by sub-components depicted in Figure 3 which were introduced in the deliverable D2.1 and described further in this deliverable.

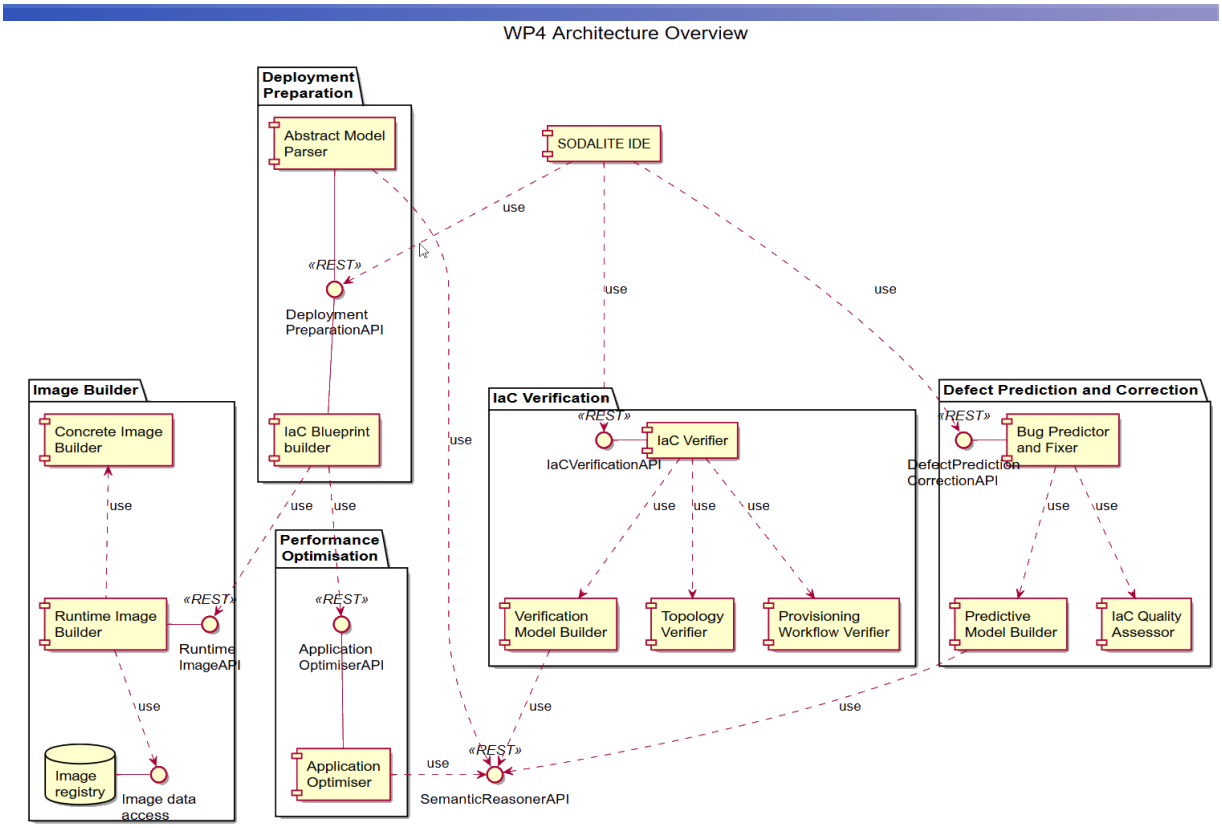

Figure 3 - SODALITE infrastructure as code layer components (WP4)

2.3.1 Components

Infrastructure Management Support components play a key utility role in the SODALITE platform by connecting the modelling and deployment layers. As SODALITE is about the IaC, special care is given to apply the IaC development approach when building and deploying the components. This means using IaC as much as possible for deployments of SODALITE components, in the integration pipeline and even using IaC when developing SODALITE components to further parameterize the usage of a component.

The components developed and deployed are part of different UML use cases defined within SODALITE such as UC16 (Build Runtime images), UC15 (Statically Optimize Application and Deployment) and mostly UC3 (Generate IaC code) found in Figure 6.

A big part of this work package is also contributing to the definition of the models used for modelling the infrastructure and application deployment plans. The following subsections describe the current status of the components and the layout of the development plan. During the development process, a part of the architecture for the Infrastructure as a Code layer was redesigned. It was decided not to have two separate independent repositories for IaC but rather build this as a single source of information from the SODALITE *Knowledge Base,* which is accessible through the *SemanticReasonerAPI*. Such a solution provides a more robust implementation and eliminates the issues of synchronization between different sources.

Additional components are envisioned to verify the correctness of the provided model, to predict possible bugs in the provided model and to optimise the application for a given target execution platform.

A simple overview of the functionalities and interaction between the components is presented in Figure 4. The *Runtime image builder*, builds the runtime images according to the Target architecture and artifact definition and build context. The *Image registry* stores the executable runtime image of the artifact defined in the application design process and built in the SODALITE deployment preparation process while the *IaC Blueprint Builder* is the central component for the preparation of the deployable *IaC blueprint* and related actuation scripts.

The *IaC Blueprint Builder* internally produces the IaC blueprint based on the input provided in the abstract application deployment model (AADM) passed to the *Abstract Model Parser*. IaC Blueprint Builder will pass the data as needed to optimise an application if such a step is required by the user.

An important part of the IaC Layer is *Application optimiser (MODAK)* which if selected may apply static optimization to the deployed application. Based on benchmarking MODAK sets optimal compiler directives when building the executables, select optimal container image for specific ML training applications or set optimal MPI parameters for typical HPC long running jobs. The Application optimizer gets called from the *IaC Blueprint Builder* to create an optimized job script which is inserted in TOSCA and later deployed to target architecture.

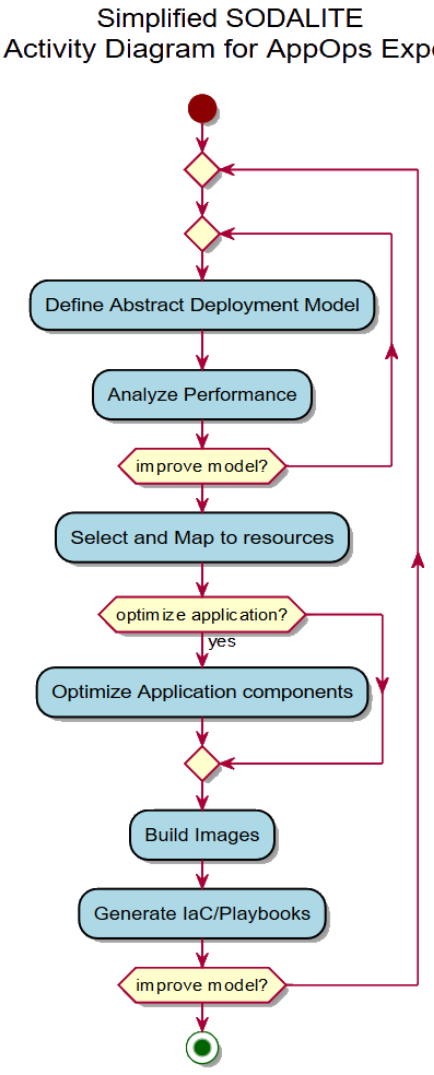

Figure 4 - WP4 pipeline schematic

The *IaC Model repository* is part of the knowledge base and contains data about the infrastructure, benchmarks, mappings, etc.

Special care is taken with regards to the preparation of a flawless and error prone IaC deployment blueprint. Several components are built to provide an error detection and validation framework which enables the automation of the deployment pipeline. The *Bug Predictor and Fixer* detects the smells in TOSCA and Ansible Artifacts and suggests corrections or fixes for each smell. The *Predictive Model Builder* builds the models that can find the smells in TOSCA and Ansible artifacts.

The *IaC Quality Assessor* can calculate different software quality metrics for TOSCA and Ansible artifacts. Similarly the *IaC Verifier* acts as a facade to the *Topology Verifier* and *Provisioning Workflow Verifier*, and coordinates the processes of verification of the application deployment topology and provisioning workflow.

Provisioning *Workflow Verifier* verifies the constraints over the deployment (provisioning) workflow of the application, then the *Verification Model Builder* builds the models required to verify the deployment model and its provisioning workflow. The *Semantic search and Reuse* (which is in fact part of the Topology Verifier) is responsible for implementing semantic search and reuse capabilities on top of the SODALITE knowledge graph.

2.4 Relationships with other SODALITE layers

WP3 is concerned with the semantic abstractions and the relevant design and modelling of applications and cloud infrastructures along with their performance characteristics and deployments. The main software components to support these are:

- the *Semantic Knowledge Base* - A semantic repository to accommodate SODALITE's knowledge in the domains of applications, infrastructure, performance optimisations, deployment and lifecycle, and more. This knowledge is generated by multiple stakeholders (e.g. resource experts) and represented into RDF-based knowledge graphs (ontologies),
- the *Semantic Reasoner* - A dedicated middleware to interact with the *Semantic Knowledge Base* by importing/retrieving data, and applying complex, rule-based semantic reasoning. Thus, the *Semantic Reasoner* exposes an API accessible by other system components,
- the SODALITE IDE - A software component to provide complete support for the authoring of abstract application deployment models with the use of the SODALITE DSL. It also enables separate views for the monitoring of each deployment's lifecycle, applied optimisations, etc.

The following technologies are used for the WP3 developments.

2.4.1 IDE

An Eclipse[5]-based framework for specifying DSL (Domain-Specific Language) metamodels and textual edition of conforming model instances. It includes several components, namely a parser, linker, typechecker, compiler, as well as a textual editor for Eclipse. It is also compatible with any editor that supports the Language Server Protocol and your favorite web browser. DSL metamodels/models are EMF (Eclipse Modeling Framework)/Ecore-based. Therefore, it is compatible with EMF-based M2T (Model-to-Text) transformations tools, such as Xpand[6] or Acceleo[7] for DSL conversion (to SPARQL queries, for example).

Xtext also provides support for Web DSL edition, leveraging existing Web editors such as Orion[8], Ace[9] or CodeMirror[10].

Around Xtext there are some related technologies. Concretely, DSLForge[11] provides an integrated Web IDE (Integrated Development Environment) Workbench for Xtext DSL editors, with a Project Explorer view and model persistence. Sirius[12] and Graphiti[13] provide a graphical DSL modeling framework for Eclipse. SODALITE IDE uses these technologies which enable users to define their DSL using graphical notation in contrast to the textual notation available in XText.

2.4.2 API Interfaces

A number of REST API interfaces serve as entry points for modelling and gathering abstract models from the *Knowledge Based* ontologies (ie. semantic graph database that acts as a SPARQL-served endpoint for ontologies) and some more entry points are still being developed. SODALITE's ontologies, created within Protégé[14], are hosted by a GraphDB[15] deployment, which supports the population of system data and the execution of rule-based semantic reasoning. This GraphDB deployment acts as SODALITE's *Semantic Knowledge Base* - repository. The input/output entry points are implemented as REST API interfaces to be used and integrated in different parts of the SODALITE stack.

3 Literature review

Virtualization of the infrastructure came to life through the widespread implementation of the Infrastructure as a Service (IaaS) paradigm giving birth to different cloud platforms and a plethora of cloud service based operators (PaaS Platform as a Service models, SaaS Software as a Service etc.).

Among others the incentives behind the uprisal of "virtualisation" are clearly:

- provisioning and usage on a time basis is much more cost effective,
- no need for complicated large scale hardware management and upgrades,
- high availability and disaster recovery defined through quality of service (QoS),
- more flexibility in license management.

The issues that such virtualisation and expansion of service based approach introduces are 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 management.

The Infrastructure as Code is a relatively novel way of interaction and management of resources on many different levels. IaC provides effective usage of resources, enables applying security concepts, versioning and central management and introduces immutability of the execution.

The usage of IaC in the DevOps toolchain:

- 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, which applies changes by rebuilding resources instead of modifying the existing resources,
- minimizes the risk of possible flaws induced by manual configuration management or user interface interactions,
- possibility of applying traceability, validation and testing helps reduce the number of errors which helps in mitigating risks and leads to robust setup for built in security.

3.1 Deployment preparation and Infrastructure management

Nowadays deployment and infrastructure management is centered around the adoption of Infrastructure as Code (IaC)[16]. This is an approach to automation of provisioning, configuration and deployment of infrastructure resources that is based on machine-readable files. These files are usually configuration files given as input to some software agent that processes them and executes specific tasks that aim to provision, configure and deploy the user-defined infrastructure. The configuration files that drive infrastructure automation can be considered as infrastructural software in all respects.

Within the context of IaC, a large number of languages and approaches have been developed. They support coding and automated execution of the phases highlighted in Figure 5[17], where configuration management is related to the deploying and managing at runtime all required underlying software stack, service provisioning concerns the acquisition of VMs or containers for executing application-level components, application deployment concerns the installation of the application on the provisioned resources, monitoring and self-adaptation have to do with the execution of the application on top of the provisioned and configured resources. As the figure suggests, some IaC languages are focused on the configuration aspects, among these, the most prominent ones are Chef, Ansible and Puppet. Others are more focused on the provisioning and

orchestration. Terraform and TOSCA (as a standard) are the two most notable representatives of this category.

While Guerriero et al[18] focus on identifying the most used approaches based on a survey answered by practitioners, in the following of this section we will focus on available research and industrial approaches that aim at covering all or some of the aspects highlighted in Figure 5. Most of the papers we have found in the literature focus on TOSCA. Some of them present a case study of the use of TOSCA. In some others, TOSCA is not the main focus, but only a means to achieve some goal. Some further papers discuss integrating TOSCA with other concepts to combine their benefits.

Wettinger et al. show the use of TOSCA in the DevOps area[19] [20]. A core principle of DevOps is to automate the deployment process to enable continuous software delivery. The authors address the challenge of combining different DevOps artifacts. For this purpose, a framework is presented to search public repositories for such artifacts. These artifacts are converted into TOSCA format to ensure a uniform representation and to enable their combination. Specifically, the transformation of Chef cookbooks and Juju charms is described. In another paper, Wettinger et al. integrate Configuration Management with Model-Driven Cloud Management in the context of DevOps[21]. Model-driven cloud management provides an overview of the structure of a complex application and supports the developer in handling necessary infrastructure changes, abstracting from lower-level actions (like installation or configuration of components), which in turn are provided by Configuration Management. The DICER approach follows a similar line, with an emphasis on modeling and deployment with TOSCA for Data Intensive Applications[22].

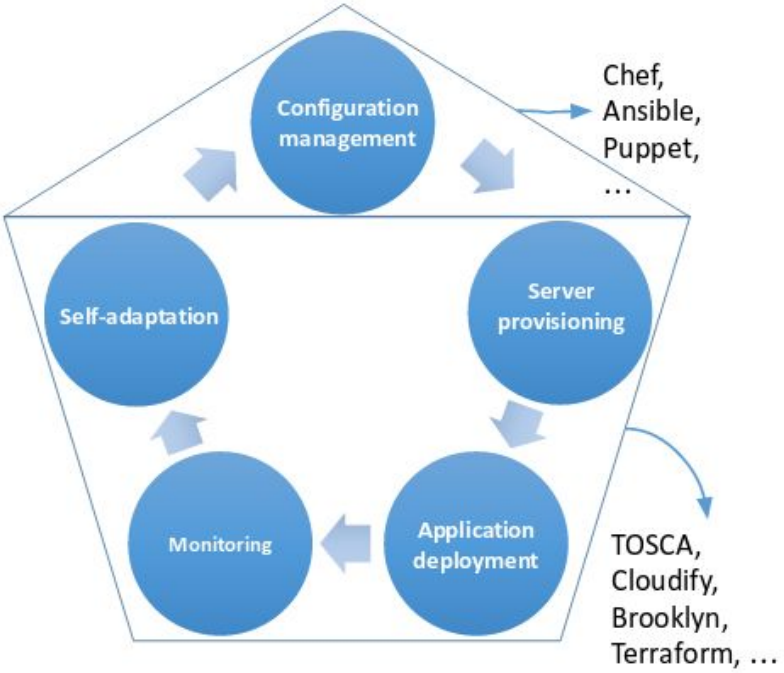

Figure 5 - Typical Ops activities covered by IaC

Brogi and Soldani focus on processing topologies and describe an approach that involves matching between individual Node Types and Service Templates[23] [24] [25]. This matching allows sets of Node Types to be grouped together in a topology to reduce its complexity. In addition, proven combinations of Node Types can be reused in new application topologies. Soldani et al. present TOSCAMart, an approach to reuse proven topologies in new environments[26]. TOSCAMart is based on a repository of various existing topologies provided to an application developer for the development of a new composite application. The developer defines a node in the new topology that describes the requirements for the fragment being inserted. TOSCAMart then selects a suitable solution for these requirements from the repository.

A different domain for using TOSCA is the specification of Internet of Things (IoT) applications. This area is addressed by various authors. Li et al. show how TOSCA can be used for an IoT application, namely an Air Handling Unit (AHU) that controls the condition and circulation of air in modern buildings[27]. The authors give examples of how to define the components of the application and describe their experience using TOSCA for this use case. In another paper, Da Silva et al. address the multitude of sensor data produced in IoT scenarios[28]. The authors describe how Complex Event Processing Systems can be deployed using TOSCA to process the incoming data and efficiently use network resources.

Palesandro et.al[29] explore an aspect-oriented approach to IaC deployment and management. They propose Mantus, a IaC-based multi-cloud builder composed of an aspect oriented Domain-Specific Language called TML, or TOSCA Manipulation Language, and a corresponding aspect weaver to inject flexibly non-functional services in TOSCA infrastructure templates. They show the practical feasibility of our approach, with also good results in terms of performance and scalability. Chirivella-Perez et. al.[30] address the challenge of fast deployment of 5G infrastructures by designing and prototyping a novel 5G service deployment orchestration architecture that is capable of automating and coordinating a series of complicated operations across physical infrastructure, virtual infrastructure, and service layers over a distributed mobile edge computing paradigm, in an integrated manner.

From the analysis of the available literature, it emerges that there are multiple approaches focusing on enabling the orchestration of IaC with the purpose of automating some phases of the Ops cycle. Compared to the works available in the literature, our goal is to enable the usage, in the same application, of heterogeneous execution environments ranging from edge devices to classical clouds and HPC clusters. Moreover, our deployment preparation and infrastructure management framework, framed within the larger context of the SODALITE project, poses particular attention to extend the Ops cycle in Figure 5 with novel steps focusing on the optimization of execution containers and on the verification and improvement of IaC.

3.2 Performance optimization

Application performance and scalability are important for HPC users. The optimisation process generally involves manual profiling and tuning of application parameters to suit target hardware. Furthermore, it is not portable and needs to be repeated when moving to other HPC systems due to the diversity of hardware in HPC systems.

The automation of application optimisation on both HPC and cloud systems requires models that can be used for performance prediction and to study how different hardware components affect performance, a task made more complex by the wide variety of cloud offerings available with a wide variety of hardware. Application profiling and historical data gathered on HPC and cloud systems were used by[31] to create a performance model. ParaOpt[32], a tool that autotunes application configurations for different instance types based on runtime and cost, was evaluated for genomics, molecular dynamics, and machine learning applications on multiple public clouds.

A number of works explored the performance of cloud environments. Exabyte compared cloud targets using the Linpack benchmark[33], and developed a software tool for the continuous evaluation of various cloud environments[34]. EPCC directly compared the performance of HPC on-premise systems and the Oracle cloud cluster using the DiRAC application benchmarks[35], discovering issues in the usability and scalability of cloud based clusters.

Many tools are developed to optimise application deployments that are packaged as containers. ConfAdvisor[36] is a tuning framework for containers on Kubernetes. AWS compute optimiser[37] optimises workloads for both cost and performance based on historical utilization metrics.

Google[38] similarly offers optimised containers for AI application deployments on the Google Cloud Platform. HPAI project[39] studied the feasibility of deploying AI workloads in HPC systems using Charliecloud[40].

3.3 Semantic decision support

In this section, we review the existing research studies related to the three key use cases or tasks of the semantic decision support of SODALITE: Iac smell and bug taxonomy, IaC verification, and IaC smell and bug prediction and correction.

3.3.1 IaC Smell and Bug Taxonomy

There are several recent works on bug and smell catalogues for IaC. Sharma et al.[41] developed a catalogue of design and implementation smells for Puppet. They defined smells violations of Puppet best practices, which were identified by analyzing the official documentation of Puppet and the validation rules of Puppet-Lint. Similarly, Schwarz et al.[42] compiled a catalogue of smells for Chef, containing violations against the best practices for Chef, extracted from the official documentation of Chef. For Puppet, Rahman et al.[43] [44] identified seven security smells by analyzing Puppet scripts in open-source repositories, and eight defect types by analyzing defect-related commits.

There are several limitations on the existing smell and bug catalogues for IaC. There is no such catalog for Ansible, which is the most popular IaC language, according to our survey with the practitioners[45]. Most catalogs focus on a subset of bug types or smell types, for example, security smells, and coding smells for a subset of constructs. They also lack the details of a bug/smell, such as consequences, resolutions, and inter-dependencies. Moreover, the smell taxonomies use the best and bad practices that were extracted from one or a few sources. Finally, there is no unified taxonomy across different (widely-used) IaC languages. Thus, in SODALITE, we first develop a catalog of best and bad practices in three IaC industrial languages (Ansible, Puppet, and Chef) and TOSCA, based on an analysis and synthesis of themulti-vocal literature. Based on this best/bad practices catalog, we develop an unified smell catalog for IaC. In addition, we create a bug and resolution catalog and dataset for Ansible, based on a qualitative analysis of bugfix related commits collected from open source software repositories.

3.3.2 IaC Verification

The key IaC languages used by SODALITE are TOSCA and Ansible. As observed by the recent surveys on IaC and TOSCA[46] [47], there is a lack of research on verification and testing of IaC. There was on studies related to Ansible. Among the studies on TOSCA verification, Brogi et. al.[48] developed a formal approach to verify the constraints on inter-component relationships in an application topology, enabling to detect the mismatches or inconsistencies between the requirements and capabilities of participant nodes. In related works[49], they also used a Petri nets base model to verify the control flow of a TOSCA management plan, for its reachability. Vetter[50] proposed a monitoring framework to detect operator errors, including configuration errors, by utilizing complex event processing and TOSCA management plans. Calcaterra et al.[51] proposed a fault aware provisioning framework based on explicit modeling of the possible types of provisioning errors as well as the counteractions to recover from them, all in the management plan of the application. The orchestrator can detect faults while enacting the provisioning workflow, and automatically to recover from detected faults.

The existing literature lacks a comprehensive taxonomy of infrastructure management errors (such as provisioning errors, deployment errors, configuration errors). Thus, SODALITE develops such error and resolution taxonomy for cloud and HPC applications based on a systematic literature review. In SODALITE, we employ semantic technologies for verifying structural constraints and semantics of IaC, and also support explaining errors and their causes, recommending the resolutions, (semi) automating the correction of erroneous IaC code through model-to-model transformations. To verify the TOSCA management plans/workflows and imperative IaC workflows (e.g., Ansible roles and plays), we use the workflow verification techniques that can verify both control flow and data flow of a workflow[52]. As the deployment of an application uses TOSCA

management plans and Ansible workflows, the verification of both artifacts needs to be coordinated and unified.

3.3.3 IaC Smell and Bug Prediction

In software engineering literature[53], data-driven models (e.g., machine learning) and rule-based models have been used to detect smells and bugs in the source code of different programming languages. Recently, the software engineering community has paid attention to bug and smell detection in IaC[54] [55] [56] [57] [58]. The rule-based techniques have been also applied to detect defects in infrastructural code scripts such as Puppet and Chef Scripts, e.g., security smells in Puppet[59], implementation and design smells in Puppet[60] and implementation and design smells in Chef[61]. Most industrial IaC smell detectors (i.e., so-called Linter tools), for example, Ansible Lint, Puppet Lint, Foodcritic (Chef), also use a rule-based approach.

Several studies have applied semantic technologies for definition and detection of patterns and antipatterns[62] [63] [64]. Settas et al.[65] modeled the antipatterns in software projects with ontologies, and used a production rule engine to implement detection rules. Inspired by that study, Brabraet al.[66] employed similar semantic technologies to detect anti patterns in cloud service APIs, and to recommend resolutions. Rekiket et al.[67] developed an ontology to represent cloud service offerings, and used common patterns and antipatterns to validate the proposed ontology. They also defined cloud service antipatterns such as invalid VM types and invalid service provider descriptions. Their antipattern detection algorithms employ SPARQL queries.

Compared with the existing studies, SODALITE proposes a semantic rule-based approach to detect the smells and antipatterns in IaC, for example, smells in TOSCA blueprints and Ansible scripts. Compared to existing approaches, our framework facilitates the generation of knowledge graphs to capture TOSCA-based deployment models following the conceptual model of DnS. The aim is to map TOSCA to self-contained, independent and reusable knowledge components, amenable to analysis and validation using Semantic Web standards, such as SPARQL. We have recently published the current results of our smell predictor[68]. To explain detected smells and recommend fixes, the initial semantic models are being extended to specify smells, their causes, and their fixes. The rule-base is being refined to and extended to cover all smells in our smell catalogue. We plan to build a unified framework to detect smells across heterogeneous deployment and infrastructure code specifications by utilizing model-driven engineering and semantic Web techniques such as ontology alignment and query rewriting.

There is an emerging trend in software defect prediction for using deep learning and natural language processing, in particular, code embeddings (code vectors)[69]. However, there are no similar studies on IaC defect prediction. Thus, SODALITE also develops deep learning and natural language processing based techniques for detecting linguistic anti-patterns and misconfiguration errors. Moreover, the software metrics have been used to predict the defect proneness of software artifacts[70]. With the collaboration with the RADON H2020 project, we develop a comprehensive set of software metrics for IaC[71] [72] [73] that can be used to predict the defect proneness of IaC artifacts. Within the SODALITE project, we further extend these IaC ametrics with the workflow metrics, which are relevant for imperative IaC languages such as Ansible.

4 Image Builder

The IaC languages usually adopt a declarative way of defining the expected status of the infrastructure and processes. In contrast with imperative methods, which give specific commands on how to implement the process, the declarative one usually gives a definition of the expected results and state of the resources after the execution.

SODALITE treats TOSCA and corresponding implementation actuation scripts (for example Ansible playbooks) as an intermediate code language which is exploited in the process of resource provisioning and management of infrastructural resources and application deployment.

One of the main aspects of the Infrastructure Management Support layer is actually defining the infrastructure and its nature in a structured way using IaC.

Since TOSCA and actuation scripts are needed by the orchestrator to put into life an application deployment, TOSCA IaC node descriptions are set up and Ansible actuation playbooks are using them as templates for building and preparation of larger application deployment plans. The results of this task are saved in the SODALITE Knowledge Base thus creating a unique source of modelling infrastructural and deployment patterns used through the SemanticReasonerAPI calls. The components being developed within Infrastructure Management Support usually represent a utility structure or only a small but crucial part of the SODALITE platform. The developed components are mostly used and accessed by SODALITE Infrastructure as Code Layer and SODALITE Runtime Layer components.

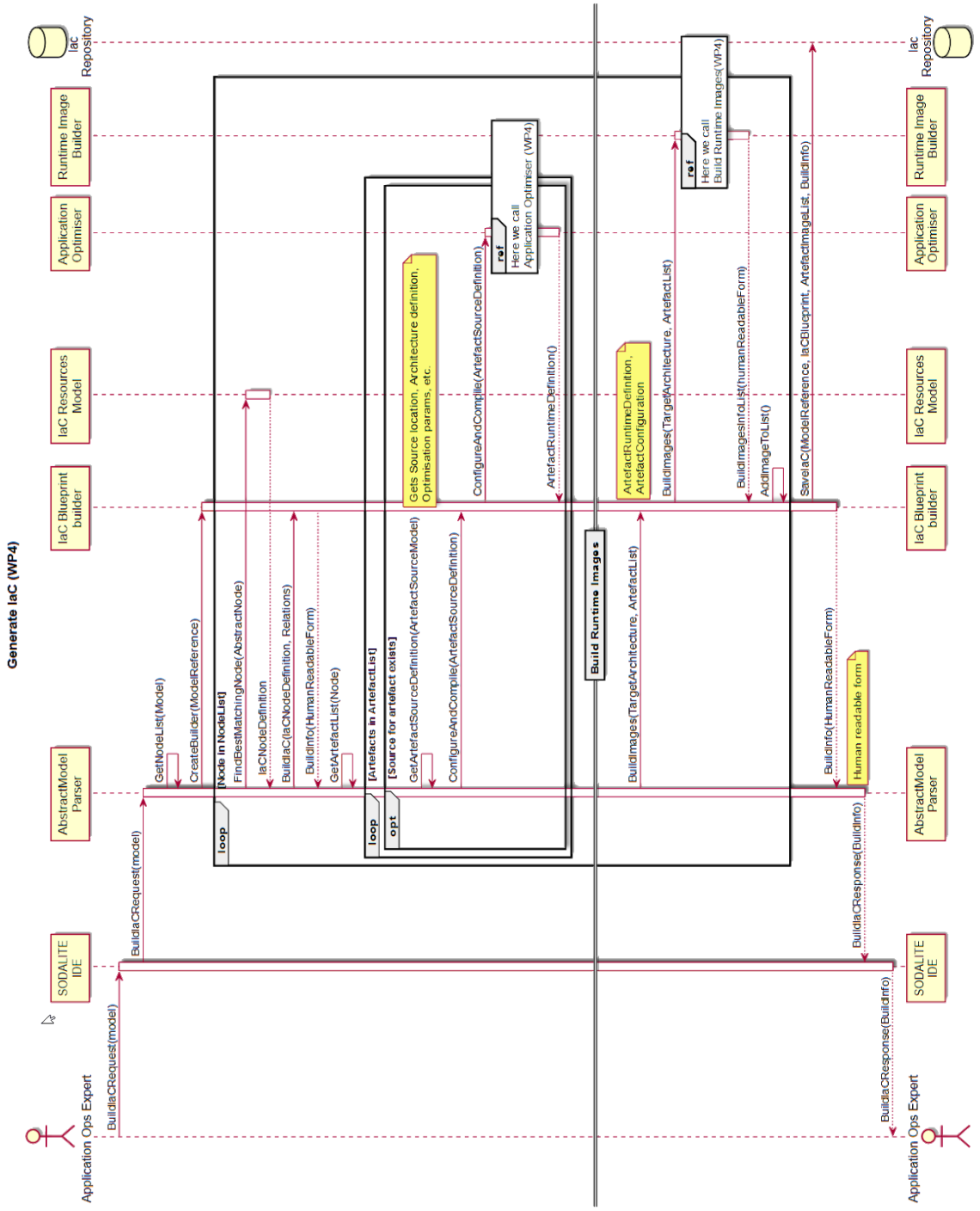

Figure 6 - Use case and Sequence diagram coverage

4.1 Background and concepts

As already foreseen at the conception of the project, the main objectives of the development of the SODALITE solution are the decoupling from monolithic applications, integration of some sort of container or unikernel enabled system, minimal size, highest flexibility and transparency of the supporting system. Until the rise of container technology, virtualization was basically bound by the building of virtual machines and specialized reduced application-centered kernel implementations known as unikernels. Open sourcing Docker[74] container technologies created a completely new perspective of virtualisation. Several aspects contribute to the widespread usage of the deployment of containerized applications on private and public cloud infrastructures. The simplicity of defining and building application environments, reusability and portability and most

certainly the ease of deployment. The portability of the applications constitutes a key feature when running in heterogeneous environments.

Most of the technologies and tools built around containers are well documented and open sourced with a very alive and vast community of developers and supporters, backed by industry leading IaaS (Infrastructure-as-a-Service) giants such as Amazon, Google, Microsoft and others.

Choosing the right container technology and tools for building up the runtime environment is a critical part of application design and deployment pipeline for bringing orchestration to HPC and different Cloud environments which are one of the key goals of the SODALITE project. A few HPC Container technologies were considered:

- Singularity[75],
- CharlieCloud[76],
- SARUS[77].

The decision to use **Singularity** containers was made by considering several aspects of the mentioned technologies, but still focusing on two main requirements for the selection of container technologies to use in the project:

- Popularity of the technology, ease of use, and availability of tools and support and
- Performance on HPC systems (native support of hardware, for example network and GPUs)

Singularity appeared as a specific container design targeting HPC systems that is widely adopted in several supercomputer centers. It does not need any superuser escalation to run and it is as integrated with the host's system as possible to ensure performance. Singularity can leverage Docker images as a way to easily share container images.

The consortium will, nevertheless, keep an open door for the introduction of possible new virtualisation technologies throughout the lifetime of the project and evaluate the support based on the evaluation model. The in depth analysis of comparable technologies and the reasoning behind the selection of the used technologies is further developed in Section 5.2 of the document D5.1 - Application deployment and dynamic runtime-initial version.

4.2 Components

4.2.1 Motivation

The preparation of the runtime images can become a difficult task for a DevOps user when building and deploying an application in a heterogeneous environment. The existence of different tools for building images on different platforms using their proprietary DSLs poses another difficult task for the DevOs teams that use such tools.

SODALITE proposes to use an existing set of well known TOSCA standard Cloud topology application deployment workflows known as TOSCA *service templates* and a vast library of Ansible modules for configuration and interaction with standard container technologies to prepare a reusable and extendible workflow for building runtime images. By encapsulating this TOSCA service template in the REST API with the xOpera lightweight orchestrator the image building becomes an easily configurable and extensible library accessible from any component that needs to setup a process of image building.

4.2.1 Runtime Image Builder

By setting the parameters of the image building process the user defines a build context for creation of the new image to be created. The *Runtime 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 and can be configured to run different workflows. The workflows for image building are usually run offline to have the images prepared and pushed to the registry, before the orchestrator starts with the execution of the blueprint e.g., provisioning the infrastructure and deployment of the application. This encapsulation enables the image building functionality to be accessible from any component in SODALITE or be just reused in a separate blueprint if needed. The extendable nature of TOSCA blueprints provides a high level of

reusability of the code for supporting the image building process. Image builder also supports session handling and authentication/authorization by JWT tokens making it easy to integrate with Identity and Access Management providers.

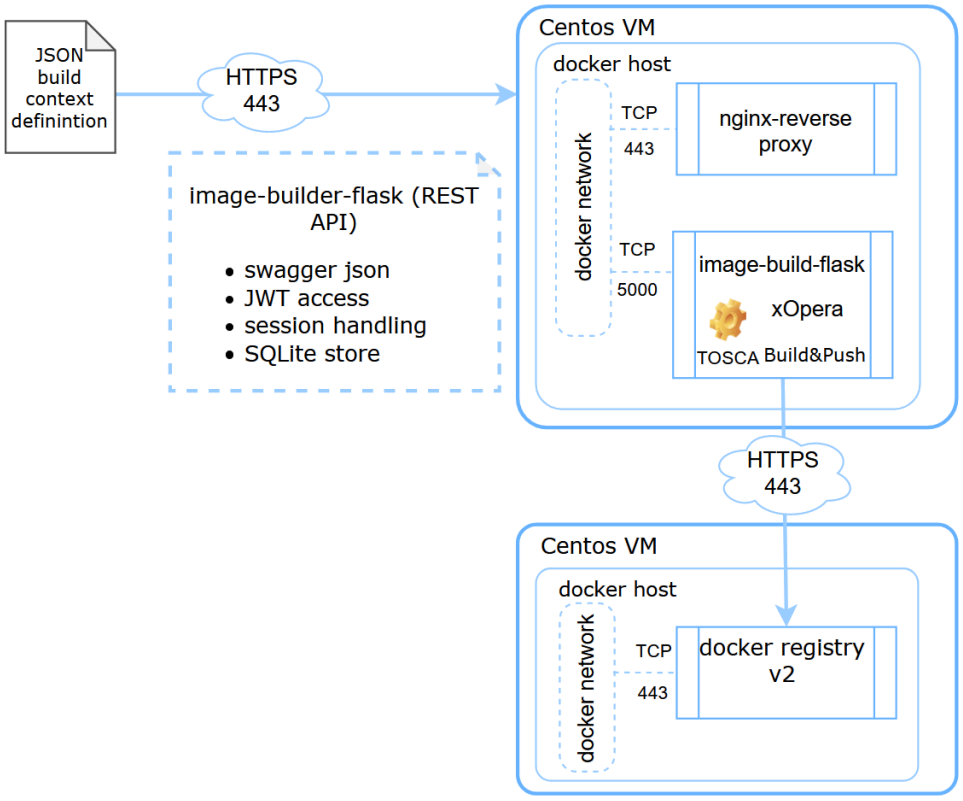

Figure 7 - Image builder architecture

The capabilities of Runtime Image Builder are:

- get image build definition (dockerfile, docker-compose, similar),
- compilation,
- packetization using templates,
- push to SODALITE docker registry.

Roles that interact with component (i.e. App Expert, ResExpert):

- App Expert.

Software dependencies:

- Python,
- Ansible,
- Docker,
- Singularity,
- SSH.

Requirements (what it should do):

- requires a dockerfile definition/configuration or a prebuilt image if already existing

Composed of:

- REST API,
- xOpera,

- TOSCA/Ansible blueprint with image building building workflows.

Depends on (other components):

- Image Registry,
- Semantic Knowledge Base (API),
- IDE,
- Concrete Image Builder.

Repositories:

https://github.com/SODALITE-EU/image-builder

4.2.2 Concrete Image Builder

Concrete image builder

The capabilities of *Concrete Image Builder* are:

- get image build definition from *Image Builder*
- packetization using templates
- push to SODALITE Image Registry (Docker/Singularity).

Roles that interact with component (i.e. App Expert, ResExpert):

- App Expert (indirectly - through the pipeline).

Software dependencies:

- Python,
- Ansible,
- Docker,
- Singularity.

Requirements:

- requires a dockerfile definition/configuration.

Composed of:

- Dockerhost engine,
- Singularity engine.

Depends on (other components):

- *Image Builder,*
- *Image Registry.*

Repositories:

https://github.com/SODALITE-EU/image-builder

4.2.3 Image Registry

Image Registry stores the executable runtime image of the artifact defined in the application design process and built in the SODALITE deployment preparation process. This registry and the images are accessible through a docker-like interface describing the access to a specific image through filtering, labels, image IDs.

The capabilities of *Image Registry* are:

- storing docker images,
- Push/Pull images.

Roles that interact with component (i.e. App Expert, ResExpert):

- *App Expert* (indirectly - through the pipeline).

Software dependencies:

- Python,
- Ansible,
- Docker.

Requirements (what it should do):

- requires a dockerfile definition.

Composed of:

- Dockerhost engine.

Depends on (other components):

- *Semantic Knowledge Base (API),*
- *Image Builder.*

Repositories:

https://github.com/SODALITE-EU/iac-management

4.3 Development status

It is one of the development goals that any deployment should be possible through SODALITE platform. Therefore a deployment blueprint for deploying the *Orchestrator* components xOpera REST API, the database for internal persistence of deployed blueprints and the TLS secure private docker *Image Registry* are used for the deployment of the application artifacts deployed on OpenStack testbed Cloud.

A simple docker *Image Builder* developed in an IaC approach using TOSCA/Ansible playbooks is in development. The images built are then pushed to the *Image Registry* and used by the *Orchestrator*.

It is also planned to support the building of Singularity images for the artifacts deployed in HPC environments.

4.4 Next steps

As for further advancements on this part of the work, it is planned to:

1. Improve security management (SSL, public access certificate management through LetsEncrypt).
2. Improve secret management in TOSCA and Ansible playbooks - possibly using a detached secret vault service (SSH keys, certificates, passwords and tokens).
3. Improve the definition of TOSCA nodes for docker defined images and containers.
4. Singularity image builder for the artifacts deployed in HPC environments.
5. Create a Singularity Image registry that will hold the Singularity images built by the concrete image builder.
6. Usage of standard code patterns and templates for the configuration application deployments and infrastructure.

5 Deployment Preparation

The Generation of IaC blueprint builds on the abstract application definition and deployment model from the WP3 Modelling layer and uses the tuple of matching IaC node definition and abstract application artifact definition with functional and nonfunctional requirements for preparing an optimal IaC blueprint and runtime artifacts for subsequent deployment in WP5 Runtime Layer. The Infrastructure as Code Layer orchestrates the parsing, building, verifying the IaC blueprint with topology and application optimisation in focus, while still keeping reference to the underlying abstract model source at all times. Figure 2 shows the internal architecture of this layer which is introduced in the deliverable D2.1.

5.1 Background and concepts

Deployment Preparation component is dedicated to the generation of an IaC blueprint from the abstract model obtained by the SODALITE IDE. The input is a file in JSON format that should be compatible with a grammar that is transformable to TOSCA language.

The final IaC blueprint is generated in TOSCA language which is an OASIS standard language to describe topologies of cloud based web services, their components, relationships, and the processes that manage them.

The approach taken in SODALITE is extensible to other languages or future standards, because *Deployment Preparation* component has two sub-components; one is an interpreter of the abstract model provided by SODALITE IDE, and the other one is the TOSCA generator which could be complemented or replaced with other language generators eventually.

The snowUC use case, which was introduced in D6.1, is used as a running example for the deployment preparation process. An abstract model of snowUC use case has been defined via IDE, which in turn provides a JSON format replication of the model.

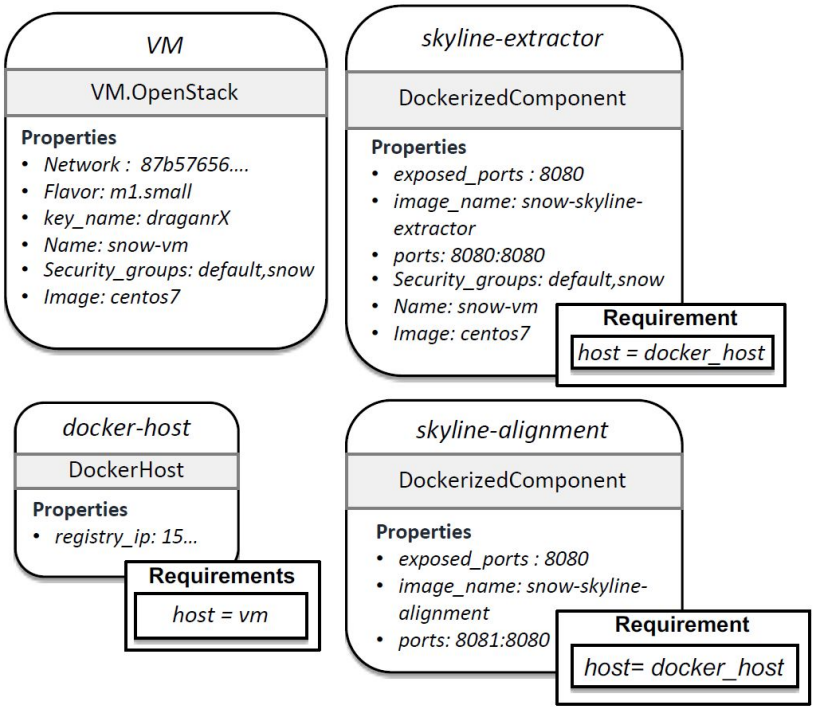

Figure 8 - The TOSCA file generated from the abstract model of snowUC use case

Once the input is verified as being correct, in the right format, and having all the required information, a TOSCA file is generated accordingly. Figure 8 shows a snippet of the TOSCA file

generated from the content of snowUC use case model. It depicts a topology definition that uses four types of nodes and their requirements. The final output, of course, would be a file that explains the IaC requirements in TOSCA, but the figure is only a visual representation for easier grasp of the content. Figure 9 shows one of the node types defined for snowUC with all of its defined properties, attributes, requirements and capabilities.

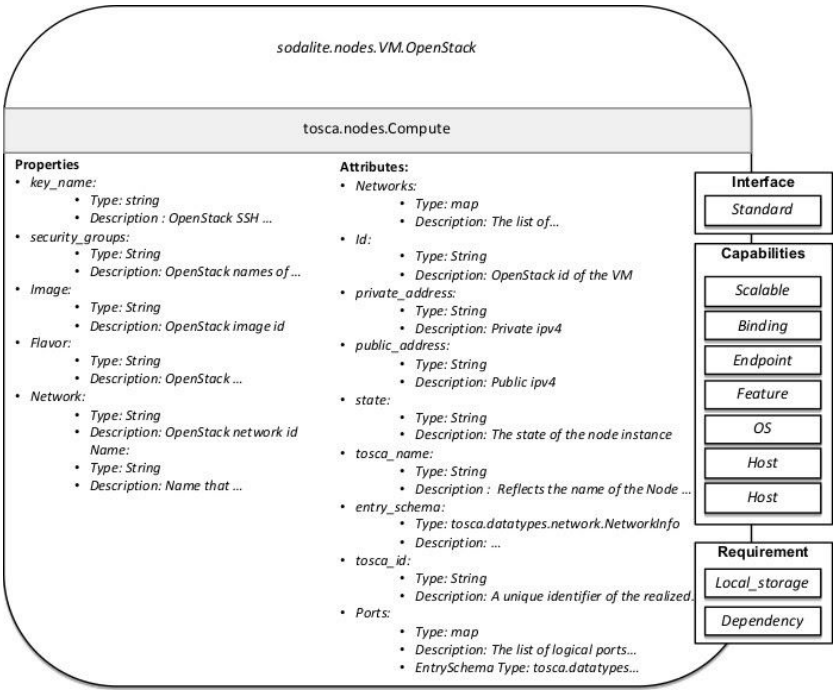

Figure 9 - An example of a TOSCA node with assigned properties and attributes, used in snowUC use case

5.2 Components

Deployment Preparation is realised through two separate components: *Abstract Model Parser* and *IaC Blueprint Builder*. They together address the UML use case UC3, *Generate IaC code*. More in detail, the first component parses the AADM provided by the SODALITE modeling layer, and the second one generates the IaC blueprint (expressed in TOSCA language) and returns information about the IaC building process in a user friendly format.

5.2.1 Motivation

The deployment preparation components have the objective to keep the modeling layer separated from the specific infrastructural languages used for the orchestration of deployment. Thanks to the IaC Builder, DevOps experts can rely on the automation introduced by orchestrators in the management of the deployment and operation of complex applications. Without the IaC Builder, DevOps experts would need to know how to write IaC code to be able to exploit orchestrators. With the IaC Builder they can focus on modeling the main assets they must deploy and on associating some resources to these. The IaC Builder will take care of all the burden associated with the generation of the corresponding IaC code. Owners of specific complex applications can exploit the IaC Builder to improve their DevOps processes and to reduce the time needed to move from the conception of the application to its running instance.

The organization of the deployment preparation in two different subcomponents enables the possibility to keep the SODALITE modeling language decoupled from its target implementation,

which, in our case, is TOSCA. The Abstract Model Parser creates an abstract syntax tree that the IaC Blueprint Builder translates in TOSCA. Changing the target language is possible by replacing the implementation of the IaC Blueprint Builder. Similarly, changing the source modeling language is possible by changing the implementation of the Abstract Model Parser.

5.2.2 Abstract Model Parser

The *Abstract Model Parser* is the central component for the preparation of the deployable IaC blueprint and related Actuation scripts.

Its main function is to abstract the parsing of the abstract deployment model from building the deployable IaC. It feeds the *IaC Builder* component with all the data provided by the *App Ops Expert* and needed for the selection,building of IaC Nodes (Blueprint) and preparation of the Actuation scripts (playbooks).

Input: Takes input from the SODALITE IDE as the reference to the abstract application deployment model. It is based on the POLIMI extensive knowledge of modelling and parsing UML deployment diagrams into IaC blueprints, e.g., TOSCA deployment blueprint.

The component allows the SODALITE IDE to:

- start the parsing process,
- cancel the parsing process at any given time,
- return resulting build time information to the user in a human readable form.

Output: Produces the output for the user based on the process of parsing abstract application deployment model.

Programming languages/tools: Python

Depends on: This component interacts with different components enabling the user to parse the abstract application deployment model and build IaC code through REST API calls to other SODALITE components:

- *IaC Blueprint Builder,*
- *IaC Resources Model.*

Critical factors: This component should be able to take input from the SODALITE IDE through a web API allowing the user to cancel the parsing process at any given time.

Repositories:

https://github.com/SODALITE-EU/iac-blueprint-builder

5.2.3 IaC Blueprint Builder

This component internally produces the IaC blueprint based on the input provided in the abstract application deployment model passed to the *Abstract Model Parser*. It flattens the application model topology in a node list and for any given node:

- returns the best matching IaC node definition from the IaC Resources Model repository,
- sets provided parameters,
- internally builds relations to other nodes.

For any selected node it then checks the artefacts to be deployed on that node.

In case the abstract model holds information about the artefact source and the source is available, it triggers the call to the Application Optimiser component in order to try to start the compilation and optimisation, defined in the model.

After all the artefacts are built as runtime binaries and configured, this component calls the Image Builder component to build and pack the artefact images ready for deployment.

At the end of the process of creation of the IaC and the building of Artefact images, it saves the resulting IaC in the IaC Repository and returns the build time information in a human readable form.

Input: Abstract application deployment model, IaC Resources Model

Output: IaC blueprint (TOSCA) with actuation scripts (Ansible playbooks). Returns information about the IaC building process in human readable form to be shown to the user.

Programming languages/tools: Python

Depends on:

- SODALITE IDE,
- *Abstract Model Parser,*
- *IaC Resources Model,*
- *Application Optimiser,*
- *IaC Repository.*

Critical factors: This component should be able to take input from the SODALITE IDE through a web API allowing the user to cancel the IaC building process at any given time.

Repositories:

https://github.com/SODALITE-EU/iac-blueprint-builder

5.3 Development status

The definition of a formal grammar to be used for automatic parsing of the input provided from SODALITE IDE and verifying the correctness of the inputs is being worked on. This grammar should provide a deterministic and well-defined mapping between the structure of the input and the notation of TOSCA. At the moment this mapping is partially defined, however the work needs to be extended so that no exception would happen during runtime.

5.4 Next steps

As for further advancements on this part of the work, it is planned to:

1. Provide a consolidated prototype of the IaC Blueprint Builder that produces verified and correct TOSCA topology definitions.
2. Incorporate the Ansible standard into the IaC Blueprint Builder, so that it will be able to generate Ansible correct definitions of IaC building process as well as TOSCA.

6 Performance Optimisation

Performance of an application that is deployed using IaC on heterogeneous infrastructure target is paramount. In SODALITE, performance optimisation maps the optimal application parameters to the infrastructure target. The application parameters can also be autotuned during run time. The performance of an application can be determined using the performance model of the application and infrastructure. The application modelling extracts the application parameters that influence the performance of an application and the infrastructure modelling will help to extract the performance characteristics of the infrastructure target like peak performance and memory bandwidth. In SODALITE, performance optimisation is achieved by two main components: *Application optimiser* and *IaC Model repository*. *Application optimiser* applies and validates optimisations to an application for a specific infrastructure target and then builds an optimised container or executable. The *IaC Model repository* stores the application and infrastructure performance model along with details about the optimisations selected by the AoE for a specific deployment.

6.1 Background and concepts

HPC places utmost importance on application performance, and the optimisation generally involves manual profiling and tuning of application to suit target hardware. Additionally, the optimisation process is not portable and needs to be repeated when moving to other HPC systems. The wide variety of cloud targets with hundreds of different server configuration provides flexibility but lacks the control and performance of HPC systems. In a software defined infrastructure, automating the optimisation of application deployments for heterogeneous targets remains an unsolved problem. Container virtualization has fastened the convergence of HPC and cloud due to its ease of use, portability, scalability, and the advancement of user-friendly runtimes.

Figure 10 shows the application optimisation requirements in a heterogeneous target. The target infrastructure can have multiple combinations of CPU, GPU or FPGA with different memory hierarchy. Target's File System and Network are also diverse and are usually shared by multiple nodes. With different schedulers for HPC and Cloud, optimal workflow orchestration becomes complex. Optimising all applications for diverse targets is out of scope for this project and instead it will focus on three different application types: AI Training & Inference, Big Data Analytics and Traditional HPC applications like Solver. This broad spectrum represents the majority of HPC applications and the demonstrating use cases in SODALITE. The list below shows the mapping of actual applications to application types:

1. AI Training and Inference - Pixelwise Mountain Skyline detection CNN[78] training and PolimiDL[79] inference,
2. Big Data Analytics - HiBench Suite[80] from Intel,
3. Traditional HPC - Code Aster[81] based Solver for in-silico clinical trials.

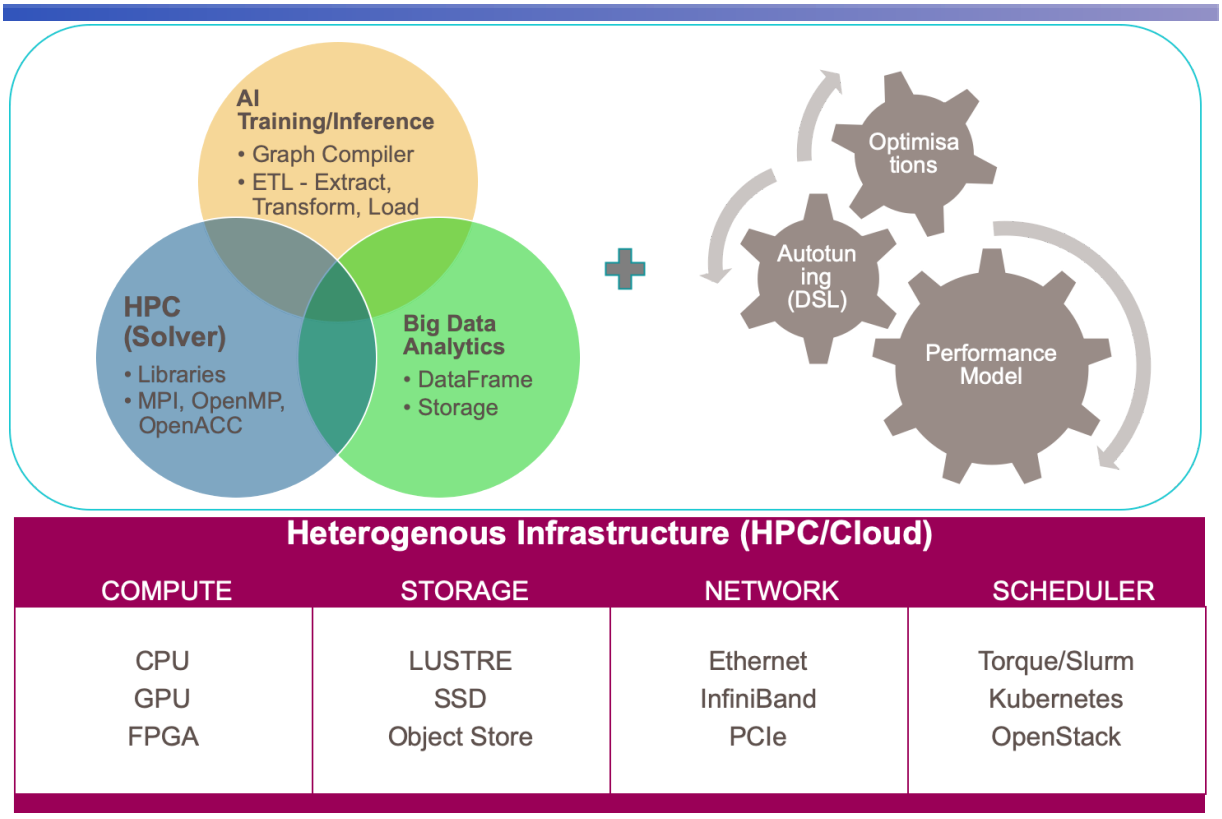

Figure 10 - Performance Optimisation for applications deployed in a Heterogeneous infrastructure

For these applications, the following optimisations will be enabled:

1. Autotuning – Application parameters can be autotuned for performance improvement. The DSL based autotuner developed as part of the CRESTA EU project will be used.
2. Multi Architecture support will enable applications to efficiently use diverse execution platforms like CPUs, GPUs or FPGAs. The application will be built for a particular target architecture or use specific target libraries.
3. Specific optimisations for application groups such as AI training/inference, HPC data analytics and traditional HPC application (Solver)
    1. AI training will be optimised with target specific libraries and Graph compilers[82]. The Extract, Transform, Load (ETL) pipeline will be optimised by improving data movement by prefetching, caching and reuse of data.
    2. Big Data Analytics applications like Apache Spark[83], DASK[84], NVIDIA's cuDataFrame[85] based on DataFrame API will be optimised for target hardware and storage.
    3. Solver (MPI) applications will be optimised by using efficient solver libraries like PETSC[86] and MUMPs[87] for different targets. HPC Standards MPI, OpenMP and OpenACC will be enabled to support performance scaling and portability.
4. Applications will be delivered in an optimised container like Docker or Singularity to ensure portability across different targets.

The following tools or libraries developed as part of different EU projects will be used to build the application optimiser.

6.1.1 CRESTA Autotuning framework

As part of the CRESTA[88] European project, a DSL-based autotuning framework was developed (initial implementation) by CRAY. This focuses on addressing the inherent complexity of the latest

and future computer architectures. Autotuning is the process by which an application may be optimised for a target platform by making automated optimal choices of how the application is built and deployed. DSL that was developed exposes choices within an application for optimisation. This framework will form the basis for autotuning in the *Application Optimiser* component.

6.1.2 Universal Data Junction

Universal Data Junction (UDJ) is a library-based transport that provides efficient communication of data between applications. It provides a capability to describe data that may be distributed and to communicate that data using put/get semantics. Distributed data (to multiple processes within an application) may be redistributed during transport. Various underlying (backend) transports are provided and may be selected at runtime.

6.1.3 Maestro data orchestration middleware

Maestro data orchestration middleware[89] addresses ubiquitous problems of data movement in complex memory hierarchies and at many levels of the HPC (High Performance Computing) software stack. This middleware framework provides object-like data abstractions for management and reasoning about user data in applications and across workflows, with the ultimate goal of optimising data-movement across the memory-storage hierarchy.

6.1.4 MAMBA - Managed Abstract Memory Arrays

A library-based programming model for C, C++ and Fortran based on Managed Abstract Memory Arrays, aiming to deliver simplified and efficient usage of diverse memory systems to application developers in a performance-portable way. MAMBA[90] arrays exploit a unified memory interface to abstract memory from both traditional memory devices, accelerators and storage. This library aims to achieve good performance portability with an easy-to-use approach that requires minimal code intrusion.

Application optimiser will explore enabling UDJ, Maestro and MAMBA libraries for optimising applications.

6.2 Components

Performance Optimisation will be enabled in SODALITE by the Static *Application optimiser* and the *IaC Model Repository* components.

The components developed and deployed are part of different UML use cases defined within SODALITE UC12: Map Resources and Optimisations (WP3) and UC15 (Statically Optimize Application and Deployment) and UC3 (Generate IaC code).

The following subsections describe the current status of the components and the layout of the development plan.

Motivation

For deploying commercial applications for Climate modelling, Material Science, there is a wealth of expertise on how these applications perform, how they can be optimised by adding resources and how they can scale across infrastructures. For HPC applications, it can be very difficult to determine this without deep application knowledge and profiling expertise is often required. We are attempting a minimal and novel approach that can be used by an expert without having to undertake exhaustive study of applications. This can be expanded further to cater for additional knowledge, profiling data or autotuning.

6.2.1 Application Optimiser

Static *Application Optimiser* aims to improve performance of an application for a given target platform based on the optimisation options selected. Figure 11 shows the architecture of the *Application Optimiser* and its dependencies. The application optimiser acts on the *Optimisation Recipe* which contains the mapping of optimisations to Application tasks and Infrastructure targets. This recipe is retrieved from the *IaC Model repository* which also hosts the application and infrastructure *Performance Model*. Based on the optimisations in the recipe, the optimisations are configured and validated. For this, Application Optimiser requires the application code written in a standard *High level API* along with the application inputs and configuration. This enables Optimiser to make performance decisions based on the available target. Optimiser uses the prebuilt optimised containers from the *Image Registry* and modifies them to build an optimised container for the application deployment. Any changes to runtime/deployment and job scripts for submission to HPC resources are also made at the same time.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python,
- Docker,
- Singularity.

Requirements (what it should do):

Application Optimiser will build an optimizer container or executable based on the Application requirements, artifacts and user selected optimisations. Optimiser will use the Application and Infrastructure performance model stored in the IaC Model repository to make performance decisions.

Composed of:

- Python,
- Container scripts.

Depends on (other components):

- IaC Model repository,
- Image Registry.

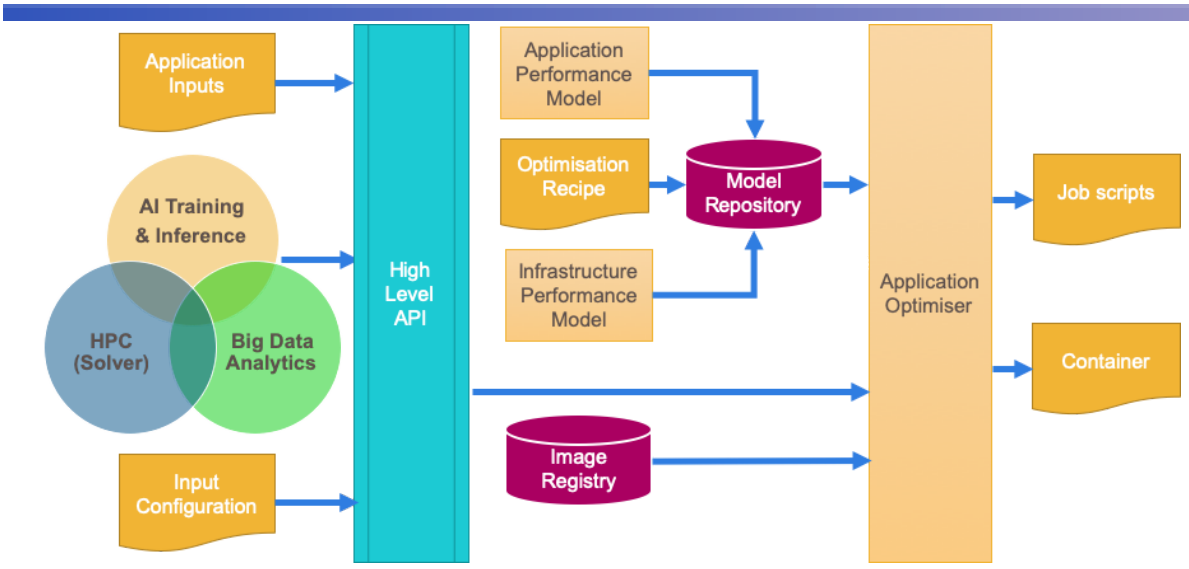

Figure 11 - Architecture of Static Application Optimiser

6.2.2 IaC Model Repository

IaC Model repository is a part of the Knowledge Base and it contains:

1. Performance Model of an infrastructure based on benchmarks.
2. Performance Model of an Application based on scaling runs done in the past.
3. Mapping of Optimisations and applications and their suitability for a particular infrastructure.
4. Optimisation recipe for a particular deployment. This contains selected optimisations by the user for an application and infrastructure target.

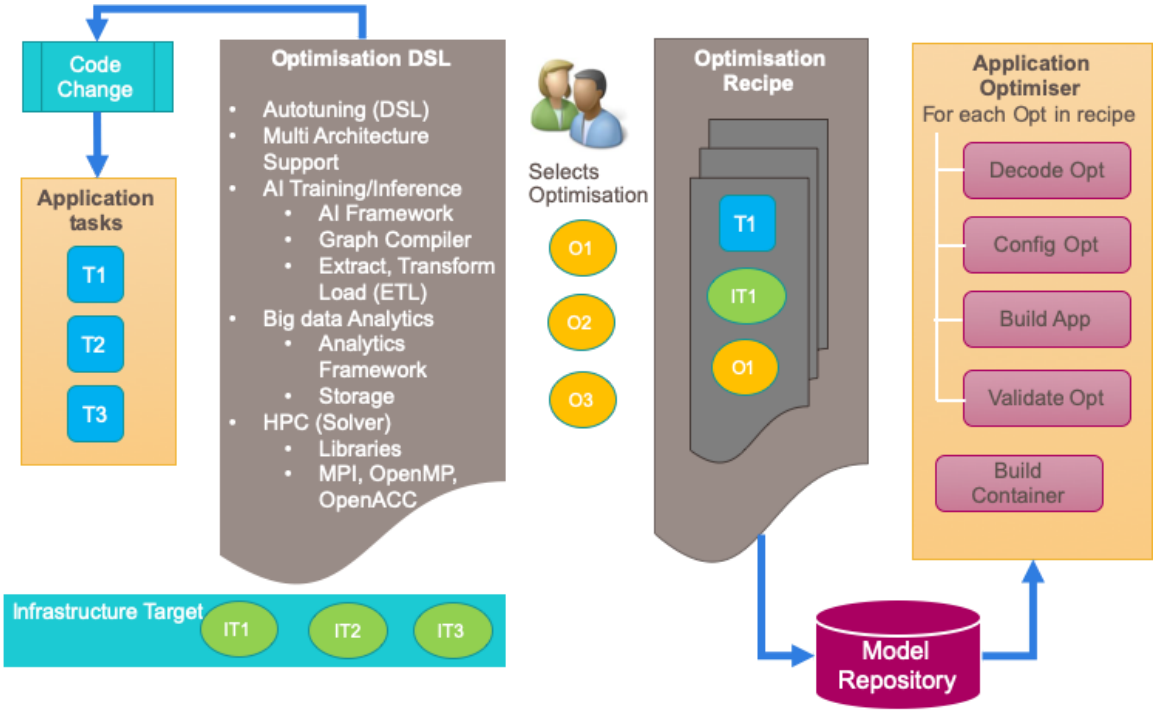

Figure 12 - Architecture of Optimisation DSL and Optimisation Recipe

Figure 12 shows how the Optimisation recipe is built from the Optimisation DSL. Optimisation DSL contains all optimisation options for a particular application type along with autotuning and multi

architecture support choices. For example, for AI training workloads, the Application expert can select the AI framework to use along with optimisations like Graph compiler and ETL options.

Some of the optimisation options require code change in applications or for applications to use a high level standard API. The mapping of selected optimisations along with the Infrastructure targets and the Application tasks will be stored as an Optimisation recipe in the IaC Model Repository. The Static application optimiser component will retrieve the recipe from the repository and will decode, configure, build and validate the optimisations.

The performance model of application and infrastructure will be obtained based on an offline run of application and benchmarks on different targets. This will also be stored in the IaC Model Repository. This information will be used by the Application Optimiser to map optimisations to targets.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python.

Requirements (what it should do):

IaC Model Repository will store the Application and Infrastructure performance model based on benchmarking and also the optimisation recipes for a particular application deployment.

Composed of:

- Python,
- Knowledge Base.

Depends on (other components):

- Application and Infrastructure Performance Model.

6.3 Development status

The initial version of Application optimiser will be delivered only in M18. The following Application optimisation components are currently in development and as part of the work:

1. Studied the performance and portability of container technologies like Docker, Singularity, Sarus, CharlieCloud, uDocker and Shifter.
2. Baseline Performance of Skyline Extraction training (AI training) was measured and profiled. This showed the performance bottlenecks in the AI training workflow and was helpful for identifying the parameters that influence performance.
3. Baseline Performance of Density mapping and Solver component of Clinical Trials use case was measured. This component was deployed as a container.
4. Prototype of Application and Infrastructure performance model is developed. This includes running benchmarks and applications on the HPC testbed.
5. Containers for different AI frameworks (TensorFlow, PyTorch) for deploying AI training workloads in HPC and Cloud are developed.

6.4 Next steps

The initial version of Application optimiser will be delivered in M18.

As part of this work, we intended to:

1. Study performance of AI training and inference workloads using MLPerf benchmarks and identify features that influence performance AI training in general. Enable Graph compiler and Extract Load Transform (ETL) based optimisations for AI training.
2. Study performance of different solver methods and libraries for Solvers (like PETSc and MUMPS) and identify features that influence performance. Enable MPI, OpenMP and OpenAcc based optimisations for Solvers
3. Study performance of Big data frameworks and identify features that influence performance. This will be enabled using common Dataframe API and storage based optimisations for Big data Frameworks. Intel's HiBench benchmark will be used as a driving use case for this study.
4. Develop Application optimiser to enable optimisations for applications based on the Performance Model developed in WP3. Application optimiser will map the features that influence optimization to the available infrastructure to build an optimised container that can be deployed.
5. Build optimised containers for deploying applications like AI Training/Inference, Big Data Analytics and HPC Solver.

7 IaC Verification, Defect Prediction and Correction

Analytics and Semantic Decision Support is mainly responsible for two SODALITE UML use cases: UC11 (Define IaC Bugs Taxonomy) and UC5 (Predict and Correct Bugs). In addition, the use case UC4 (Verify IaC) is also supported in collaboration with WP3. Infrastructure as code (IaC) simplifies the provision and configuration of the IT infrastructure at scale. As the size and complexity of IaC projects increase, it is critical to maintain the code and design quality of IaC Scripts. To this end, the detection and correction of defective and erroneous IaC scripts is of paramount importance.

7.1 Background and concepts

Within the scope of SODALITE, a bug is a software smell or an antipattern. A software smell is any characteristic in the artifacts of the software that possibly indicates a deeper problem or quality issue [32]. There are different types of software smells such as architectural smells, design smells, implementation/code smells and linguistic smells. The software smells can negatively impact software quality attributes such as maintainability, effort/cost, reliability, change proneness, testability and performance.

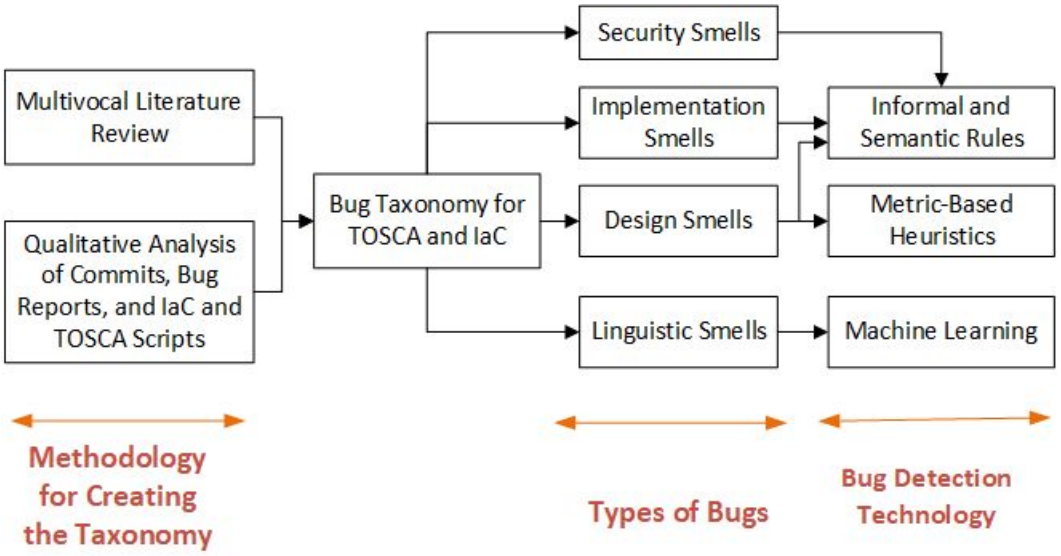

Figure 13 - An Overview of the Analytics and Semantics Decision Support

Figure 13 provides an overview of our *Analytics and Semantic Decision Support*. First a multivocal literature review was conducted, including both academic literature and gray literature on IaC best and bad practices and smells. The use of the bad practices as well as the deviations from the best practices are defined as anti-patterns/smells (as in the research literature[33]). This initial catalog of anti-patterns and smells (the IaC bugs taxonomy) are further revised and extended by the bugs found from a qualitative analysis of bug commits, bug reports/tickets and IaC code scripts.

Four key types of smells and antipatterns as IaC Bugs are considered: implementation smells, design smells, security smells, and linguistic smells. Implementation smells are quality issues such as naming convention, style, and formatting, and Design smells are quality issues in the structure of the TOSCA blueprints and IaC scripts [34]. Security smells are recurring coding patterns that indicate security weaknesses[35]. Linguistics smells are recurring poor practices related to inconsistencies among the naming, documentation, and implementation of an entity [36].

The defect prediction can be implemented using different techniques such as rule-based reasoning, metric-based heuristics, and machine learning (data-driven reasoning). The most appropriate technique for detecting each smell will be selected based on the nature of a smell and its fix, the SODALITE technology stack, and the size and quality of the dataset for the smell.

The use case UC4 (Verify IaC) focuses on checking the constraints over the structures of the TOSCA blueprints and Ansible scripts as well as the constraints over the provisioning workflows/plans. As the SODALITE approach is grounded on semantic modeling (ontologies) and reasoning, the verification of the structural constraints uses the ontological reasoning. The standards such as SHACL(Shapes Constraint Language) and SPARQL Query Language are used. To verify the provisioning workflows, Petri net is used, a widely used formal model for verifying workflows and business processes.

7.2 Components

The components in the Analytics and Semantic Decision Support can be broadly divided into two categories: bug prediction and correction, and verification. *Bug Predictor and Fixer, Predictive Model Builder*, and *IaC Quality Assessor* supports bug prediction and correction for IaC. *IaC Verifier, Topology Verifier*, and *Provisioning Workflow Verifier* implement the verification of IaC.

7.2.1 Motivation

The main objective of our analytics and semantic decision support for IaC is to enable IaC developers to create valid and defect-free IaC artifacts with ease. They should be able to find and fix the syntactical/structural and semantic errors in IaC artifacts, find and fix code smells, design smells, security smells, anti-patterns in IaC artifacts prior to deploying and executing them. To this end, we include the components for verifying the syntax and semantics of IaC codes as well as the corresponding provisioning workflows, for assessing the quality of the IaC artifacts with metrics, and for detecting and fixing different types of smells and bugs in IaC codes.

7.2.2 Bug Predictor and Fixer

Bug Predictor and Fixer detects the smells in TOSCA and Ansible artifacts and suggests corrections or fixes for each smell. *Application Ops Expert (AOE)* can select the desired fixes from the suggestions, and apply the selected fixes to repair the defective artifacts. *Bug Predictor and Fixer* uses a model or a set of heuristics to predict the smells. This model (built by *Predictive Model Builder*) could be a rule based or machine learning based model.

The capabilities of Bug Predictor and Fixer are:

- detect smells,
- suggest fixes for smells,
- apply fixes to repair the defective IaC artifacts.

Roles that interact with component (i.e. App Expert, ResExpert):

- App Expert.

Software dependencies:

- Python,
- Java,
- Ansible-Lint,
- JAX-RS 2.0 Supported Web Server.

Requirements (what it should do):

Detect bugs in TOSCA and Ansible artifacts and suggest/apply the corrections for the identified bugs

Composed of:

- REST API: expose bug prediction and correction capabilities as RESTFul service operations,
- backend.

Depends on (other components):

- *Predictive Model Builder,*
- *IaC Quality Assessor,*
- *Semantic Knowledge Base.*

Repositories:

https://github.com/SODALITE-EU/defect-prediction

7.2.3 Predictive Model Builder

This component builds the models that can find the smells in TOSCA and Ansible artifacts. A rule-based model for detecting implementation and security smells in Ansible and TOSCA is used. In particular, the semantic reasoning over the SODALITE ontologies developed in WP3 is performed. The machine learning based models will be used for detecting linguistic smells in Ansible such as named-based bugs, variable misuses, and task name and logic inconsistencies.

The capabilities of Predictive Model Builder are:

- build a suitable model for predicting bugs,
- update the model as necessary to improve the prediction performance.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python,
- Java.

Requirements (what it should do):

Build a rule-based or machine learning based model that can predict bugs in TOSCA and Ansible artifacts

Composed of:

- backend.

Depends on (other components):

- IaC Quality Assessor,
- Semantic Knowledge Base,
- Bug Predictor and Fixer.

Repositories:

https://github.com/SODALITE-EU/defect-prediction

7.2.4 IaC Quality Assessor

This component can calculate different software quality metrics for TOSCA and Ansible artifacts. These metrics are used by the heuristics that predict the design smells in TOSCA and Ansible. The

metrics include general metrics such as Lines of Code, Lines of Comments, and Number of Blank Lines, as well as specific metrics such as cohesion and coupling metrics for Ansible tasks, roles, and playbooks.

The capabilities of *IaC Quality Assessor* are:

- calculate quality metrics for TOSCA,
- calculate quality metrics for Ansible.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python,
- Java.

Requirements (what it should do):

Calculate the software quality metrics for TOSCA and Ansible artifacts.

Composed of:

- backend.

Depends on (other components):

- Bug Predictor and Fixer.

Repositories:

https://github.com/SODALITE-EU/iac-quality-framework

7.2.5 IaC Verifier

This component acts as a facade to the *Topology Verifier* and *Provisioning Workflow Verifier*, and coordinates the processes of verification of the application deployment topology and provisioning workflow. It provides a uniformed REST API for all types of verifications.

The capabilities of *IaC Verifier* are:

- expose a uniformed REST API for all types of verifications,
- Coordinate topology verification and provisioning workflow verification.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python,
- Java.

Requirements (what it should do):

Enable the RESTFul consumption of the verification service capabilities.

Composed of:

- REST API: expose IaC verification capabilities as RESTFul service operations,
- backend.

Depends on (other components):

- *Verification Model Builder,*
- *Topology Verifier,*
- *Provisioning Workflow Verifier.*

Repositories:

https://github.com/SODALITE-EU/verification

7.2.6 Topology Verifier

This component verifies the constraints over the structures of the TOSCA blueprints and Ansible scripts. This will consider the verification of the requirements of the nodes, the node-relationships, the capabilities of the nodes, and node substitutability. The topology verifier uses ontological reasoning rules over SODALITE ontologies for implementing verification logics. It also implements semantic search and reuse capabilities on top of the SODALITE knowledge graph (WP3), providing the REST API (supported by the Semantic Reasoner in WP3) for searching the Knowledge Base (KB), e.g. to get all nodes from the KB or to get the properties of a specific node type.

The capabilities of *Topology Verifier* are:

- verify the constraints over TOSCA blueprint structure,
- verify the constraints over Ansible script structure,
- support searching and reuse of TOSCA entities.

Roles that interact with component (i.e. App Expert, ResExpert):

There is no direct interaction of AppOps and Resource Experts with the topology verifier.

Software dependencies:

- Python,
- Java,
- RDF4J V3.0.0 (for SPARQL execution, RDF API, etc.).

Requirements (what it should do):

- verify that TOSCA application topologies are semantically valid (e.g. there are no missing required properties, validation conditions for sources of relationships, etc.),
- verify the constraints over Ansible script structure,
- generate a validation report to be sent to the IDE with suggestions on how to address the validation errors,
- implement the backend services of the REST API developed in WP3. This involves context-aware services for:
    - returning information about certains nodes (e.g. properties, attributes, capabilities, etc.)
    - extracting and returning saved AADMs in the KB (in JSON).

Depends on (other components):

- *Verification Model Builder,*
- *IaC Verifier,*

- *Semantic Knowledge Base,*
- *Semantic Reasoner*.

Repositories:
https://github.com/SODALITE-EU/verification

7.2.7 Provisioning Workflow Verifier

This component verifies the constraints over the deployment (provisioning) workflow of the application using one of the widely used techniques for verifying workflows such as Petri Net. The workflow is described in the Ansible scripts in terms of tasks, roles, plays, and variables.

The capabilities of Provisioning Workflow Verifier are:

- map and transform Ansible scripts into a formal model (Petri Net),
- verify the constraints over the provisioning workflow.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python,
- Java,
- Petri Net tools.

Requirements (what it should do):
Verify the constraints over the provisioning workflow described in the IaC (Ansible) artifacts.

Depends on (other components):

- *Verification Model Builder,*
- *IaC Verifier.*

Repositories:
https://github.com/SODALITE-EU/verification

7.2.8 Verification Model Builder

This component builds the models required to verify the deployment model and its provisioning workflow, for example, a knowledge base instance for ontological (semantic) reasoning on the topology, and a petri net representation for the provisioning (deployment) workflow.

The capabilities of Verification Model Builder are:

- build a model that can verify the deployment model and its provisioning workflow.

Roles that interact with component (i.e. App Expert, ResExpert):

- N/A.

Software dependencies:

- Python,
- Ansible,
- Java.

Requirements (what it should do):

Build a formal model that can verify the topology and the provisioning workflow of a deployment model described in the IaC artifacts.

Depends on (other components):

- *Semantic Knowledge Base,*
- *IaC Verifier.*

Repositories:

https://github.com/SODALITE-EU/verification

7.3 Development status

The IaC bug taxonomy for Ansible is being developed based on a literature review on IaC smells and Ansible best and bad practices. The taxonomy also has been extended by adding security smells for both Ansible and TOSCA.

To detect the implementation smells and security smells in Ansible, the Ansible-Lint tool is being extended by adding informal rules for detecting each smell. With collaboration with another H2020 project RADON[91], the tool support for calculating several IaC metrics for Ansible is being developed. Also a metric-based heuristics for detecting design smells is being developed in the bug catalog. To detect the security smells in TOSCA, semantic reasoning rules are under development using the SODALITE ontologies from WP3. Moreover, investigation is underway into data-driven defect prediction for Ansible: deep learning based approach for named-based bugs and variable-misuses, and NLP and machine learning based approach for task name, logic, module documentation inconsistencies.

As regards to the verification of IaC, WP3 has developed the basic support for verifying the deployment topology modelled in the TOSCA blueprint. More specifically, the semantics of the *Topology Verifier* are built on top of the reasoning infrastructure provided by the *Semantic Reasoner* (WP3), implementing the verification logic on top of the SODALITE RDF graphs. Capitalising on the format Ontology Design Pattern (ODP) of SODALITE (see D3.1), the module is able to detect inconsistencies in TOSCA topologies and provide suggestions on how to address these errors. In the first version of the semantic verification of TOSCA topologies, the module supports the detection of missing required properties, basic form of data value mismatches and validation of TOSCA requirements (source of relationships).

Additionally the logic for context-aware discovery and reuse of information from the KB has been developed. These services are consumed by the SODALITE IDE, assisting users in defining the TOSCA topologies. The first version of the module supports basic discovery of resources and node templates (see D3.1, Section 2.3.1 API Interfaces).

Also, RESTFul APIs for both verification and defect prediction have been developed. The mappings from Ansible to Petri net have been defined to support the verification of the workflow described in the Ansible playbooks and roles.

7.4 Next steps

The following are the next steps for the bug prediction and corrections.

- *Improve the bug taxonomy by adding the bugs found from a qualitative analysis of bug fix commits and issue tickets*. We need to identify possible types of bugs in IaC code in addition to types of smells and anti-patterns that have so far identified.
- *Complete the ongoing works on data-driven approaches to defect prediction.* Our current support only considers the smells and anti-patterns that can be detected using informal

rules and ontological reasoning. There are types of bugs/smells whose presence can be better predicted using machine learning, and NPL based (data-driven) techniques.

- *Further develop semantic approach defect prediction as SODALITE ontologies evolve, for example, ontological models for Ansible scripts and ontological models for infrastructure and application design patterns.* The semantic web technologies is a key pillar of the SODALITE approach. Currently, SODALITE only has an ontology for TOSCA. However, the relevant tasks (WP3) improve the current TOSCA ontology with semantic modeling of TOSCA policies, and add an Ansible ontology. Consequently, our current semantic approach to defect prediction should also be extended to detect the more smells in TOSCA and Ansible. Furthermore, we can utilize the existing semantic web techniques such as ontology mapping and alignment, and pattern detection and recommendation to build a uniformed approach predicting defects across different IaC languages, as well as to recommend fixes.

The following are the next steps for the IaC verification.

- *Complete the verification of the deployment topology described in the TOSCA blueprint*. As the TOSCA ontology evolves with the support for more TOSCA features including policies (WP3), our verification supports will also be extended.
- *Add the verification of the structural aspects of Ansible scripts*. SODALITE (WP3) will also develop an ontology for Ansible, which enables us to verify the structural semantics of Ansible scripts via ontological reasoning.
- *Complete the verification of the provisioning workflow of a given deployment model*. The imperative workflow of Ansible plays and the provisioning workflows of TOSCA should be verified for common types of control and data flow errors. Our current Petri net based tool should be extended using the results from the existing research studies on Petri net based verification of data and control flow errors in general workflows.

The following are the next steps for the semantic search and reuse capabilities.

- Extend the context-aware discovery services with additional searching functionality (e.g. searching by keywords, support additional filtering criteria). To this end, the following updates are foreseen:

  a) To enrich the SODALITE conceptual meta-model (developed in WP3) by incorporating external semantic networks and lexicon databases (e.g. BabelNet and WordNet), so as to support advanced searching functionalities capitalizing on templates and keyword-based searching,

  b) implement a combination of native Description Logics reasoning and rule-based reasoning to take into account the semantics of the generated Knowledge Graphs.
- Enrich the validation report, both in terms of the content and quality of suggestions. This involves the extension of the validation logic to take into account additional semantics of the TOSCA specification, as well as domain knowledge that will exploit the specifics of the defined models. Part of this activity will be the enrichment of the IDE capabilities (WP3) to provide support to the end users, trying not only generate and report validation errors, but also to enrich the submitted models in cases where the validation errors can be automatically resolved, e.g. to insert default values or to automatically resolve nodes' requirements based on resource specifications.

8 Conclusion

This document has set out to report on the status of the IaC management layer at the end of the first year of the SODALITE project. It provides a report on the state of the development of the individual components and the issues encountered during the development stages.

The development of the individual components and tools will of course advance and intensify in the following period of the project. The challenges that WP4 will be facing in the next period will be represented in the form of integration for all the tools and techniques, with the utmost consideration for security related affairs. It is expected to achieve considerable steps forward in the definition of the TOSCA models and Ansible playbooks, these advancements will then of course reverberate also through the remaining WPs and of course also in the developments of the Use Cases. This report will be followed-up by D4.2 in M24 that will present the progress in the development in the following year.

References

1. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csd01/TOSCA-Simple-Profile-YAML-v1.2-csd01.html
2. https://puppet.com/
3. https://www.chef.io/ansible
4. https://www.ansible.com/
5. https://www.eclipse.org/
6. https://wiki.eclipse.org/Xpand
7. https://www.eclipse.org/acceleo/
8. https://wiki.eclipse.org/Orion
9. https://ace.c9.io/
10. https://codemirror.net/
11. https://dslforge.org/
12. https://www.eclipse.org/sirius/
13. https://www.eclipse.org/graphiti/
14. https://protege.stanford.edu/
15. http://graphdb.ontotext.com/
16. https://martinfowler.com/bliki/InfrastructureAsCode.html
17. M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin and D. A. Tamburri, "Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach," 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, 2018, pp. 156-15609, doi: 10.1109/ICSA.2018.00025.
18. M. Guerriero, M. Garriga, D. A. Tamburri and F. Palomba, "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), Cleveland, OH, USA, 2019, pp. 580-589, doi: 10.1109/ICSME.2019.00092.
19. Wettinger, J., Breitenb¨ucher, U., Kopp, O., Leymann, F.: Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel. Future Generation Computer Systems 56, 317–332 (2016)
20. Wettinger, J., Breitenb¨ucher, U., Leymann, F.: Standards-based DevOps automation and integration using TOSCA. In: Proc. UCC 2014. pp. 59–68 (2014)
21. Wettinger, J., Behrendt, M., Binz, T., Breitenb¨ucher, U., Breiter, G., Leymann, F., Moser, S., Schwertle, I., Spatzier, T.: Integrating configuration management with model-driven cloud management based on TOSCA. In: Proc. CLOSER 2013. pp. 437–446 (2013)
22. M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin and D. A. Tamburri, "Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach," 2018 IEEE International Conference on Software Architecture (ICSA), Seattle, WA, 2018, pp. 156-15609, doi: 10.1109/ICSA.2018.00025.
23. Brogi, A., Soldani, J.: Matching cloud services with TOSCA. In: Proc. ESOCC 2013. pp. 218–232 (2013)
24. Brogi, A., Soldani, J.: Reusing cloud-based services with TOSCA. In: Proc. Informatik 2014. pp. 235–246 (2014)
25. Brogi, A., Soldani, J.: Finding available services in TOSCA-compliant clouds. Science of Computer Programming 115-116, 177–198 (2016)
26. Soldani, J., Binz, T., Breitenb¨ucher, U., Leymann, F., Brogi, A.: ToscaMart: A method for adapting and reusing cloud applications. Journal of Systems and Software 113, 395–406 (2016)
27. Li, F., V¨ogler, M., Claeßens, M., Dustdar, S.: Towards automated IoT application deployment by a cloud-based approach. In: Proc. SOCA 2013. pp. 61–68 (2013)
28. Franco da Silva, A., Hirmer, P., Breitenb¨ucher, U., Kopp, O., Mitschang, B.: Customization and provisioning of complex event processing using TOSCA. Computer Science – Research and Development pp. 1–11 (2017)

29. A. Palesandro, M. Lacoste, N. Bennani, C. Ghedira-Guegan and D. Bourge, "Mantus: Putting Aspects to Work for Flexible Multi-Cloud Deployment," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, CA, 2017, pp. 656-663, doi: 10.1109/CLOUD.2017.88.

30. Chirivella Pérez, Enrique & Alcaraz Calero, Jose & Wang, Qi & Gutiérrez-Aguado, Juan. (2018). Orchestration Architecture for Automatic Deployment of 5G Services from Bare Metal in Mobile Edge Computing Infrastructure. Wireless Communications and Mobile Computing. 2018. 1-18. 10.1155/2018/5786936.

31. M. Baughman, R. Chard, L. T. Ward, J. Pitt, K. Chard, and I. T. Foster,"Profiling and predicting application performance on the cloud." in UCC,2018, pp. 21–30.

32. C. Wu, T. Summer, Z. Li, A. Woodard, R. Chard, M. Baughman,Y. Babuji, K. Chard, J. Pitt, and I. Foster, "Paraopt: Automated application parameterization and optimization for the cloud," in 2019 IEEE International Conference on Cloud Computing Technology and Science(CloudCom). IEEE, 2019, pp. 255–262.

33. M. Mohammadi and T. Bazhirov, "Comparative benchmarking of cloud computing vendors with high performance linpack," inProceedings of the 2nd International Conference on High Performance Compilation,Computing and Communications, 2018, pp. 1–5.

34. Continuous evaluation of the performance of cloud infrastructure for scientific applications,"arXiv preprint arXiv:1812.05257, 2018..

35. 2020. [Online]. Available: https://www.epcc.ed.ac.uk/blog/2020/06/benchmarking-oracle-bare-metal-cloud-dirac-hpc-workloads

36. T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam, "Con-fadvisor: A performance-centric configuration tuning framework for containers on kubernetes," in2019 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2019, pp. 168–178.

37. Aws compute optimizer. https://aws.amazon.com/compute-optimizer/,last accessed 12. June 2020.

38. S. Krishnan and J. L. U. Gonzalez, "Google compute engine," inBuilding your next big thing with Google cloud platform.Springer,2015, pp. 53–81.

39. D. Brayford, S. Vallecorsa, A. Atanasov, F. Baruffa, and W. Riviera,"Deploying ai frameworks on secure hpc systems with containers." in2019 IEEE High Performance Extreme Computing Conference (HPEC).IEEE, 2019, pp. 1–6.

40. R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," inProceedings of the In-ternational Conference for High Performance Computing, Networking,Storage and Analysis, 2017, pp. 1–10.

41. Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, New York, NY, USA, 189-200

42. Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In 2018 11th International Conference on the Quality of Information and Communications Technology. 220–228.

43. Rahman, Akond, Chris Parnin, and Laurie Williams. "The seven sins: security smells in infrastructure as code scripts." Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019

44. Rahman, Akond, et al. "Gang of eight: A defect taxonomy for infrastructure as code scripts." Proceedings of the 42nd International Conference on Software Engineering, ICSE. Vol. 20. 2020.

45. Guerriero, Michele, et al. "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry." 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE.

46. Rahman, Akond, Rezvan Mahdavi-Hezaveh, and Laurie Williams. "A systematic mapping study of infrastructure as code research." Information and Software Technology 108 (2019): 65-77.

47. Bellendorf, Julian, and Zoltán Adám Mann. "Specification of cloud topologies and orchestration using TOSCA: a survey." Computing (2019): 1-23.

48. Brogi, Antonio, Antonio Di Tommaso, and Jacopo Soldani. "Sommelier: a tool for validating TOSCA application topologies." International Conference on Model-Driven Engineering and Software Development. Springer, Cham, 2017.

49. Brogi, Antonio, et al. "A Petri net-based approach to model and analyze the management of cloud applications." Transactions on Petri Nets and Other Models of Concurrency XI. Springer, Berlin, Heidelberg, 2016. 28-48.

50. Vetter, Arthur. "Detecting operator errors in cloud maintenance operations." 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2016.

51. Di Modica, Giuseppe, et al. "Implementation of a fault aware cloud service provisioning framework." 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE, 2018.

52. Mülle, Jutta, Christine Tex, and Klemens Böhm. "A practical data-flow verification scheme for business processes." Information Systems 81 (2019): 136-151.

53. Sharma, Tushar, and Diomidis Spinellis. "A survey on software smells." Journal of Systems and Software 138 (2018): 158-173

54. Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, New York, NY, USA, 189-200

55. ulian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In 2018 11th International Conference on the Quality of Information and Communications Technology. 220–228.

56. Rahman, Akond, Chris Parnin, and Laurie Williams. "The seven sins: security smells in infrastructure as code scripts." Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019

57. Rahman, Akond, et al. "Gang of eight: A defect taxonomy for infrastructure as code scripts." Proceedings of the 42nd International Conference on Software Engineering, ICSE. Vol. 20. 2020.

58. Rahman, Akond, and Laurie Williams. "Characterizing defective configuration scripts used for continuous deployment." 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2018.

59. Rahman, Akond, Chris Parnin, and Laurie Williams. "The seven sins: security smells in infrastructure as code scripts." Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019

60. Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). ACM, New York, NY, USA, 189-200

61. Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code Smells in Infrastructure as Code. In 2018 11th International Conference on the Quality of Information and Communications Technology. 220–228.

62. Brabra, Hayet, et al. "On semantic detection of cloud API (anti) patterns." Information and Software Technology 107 (2019): 65-82

63. Dimitrios L Settas, Georgios Meditskos, Ioannis G Stamelos, and Nick Bassiliades. 2011. SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. Expert Systems with Applications 38, 6 (2011), 7633–7646

64. Molka Rekik, Khoulou Boukadi, Walid Gaaloul, and Hanêne BenAbdallah. 2017. Anti-pattern specification and correction recommendations for semantic cloud services. In 50th Hawaii International Conference on System Sciences.

65. Dimitrios L Settas, Georgios Meditskos, Ioannis G Stamelos, and Nick Bassiliades. 2011. SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. Expert Systems with Applications 38, 6 (2011), 7633–7646

66. Brabra, Hayet, et al. "On semantic detection of cloud API (anti) patterns." Information and Software Technology 107 (2019): 65-82

67. Molka Rekik, Khoulou Boukadi, Walid Gaaloul, and Hanêne BenAbdallah. 2017. Anti-pattern specification and correction recommendations for semantic cloud services. In 50th Hawaii International Conference on System Sciences.

68. Kumara, Indika, et al. "Towards Semantic Detection of Smells in Cloud Infrastructure Code" The 10th International Conference on Web Intelligence, Mining and Semantics (WIMS 2020). ACM, 2020

69. Kumara, Indika, et al. "Towards Semantic Detection of Smells in Cloud Infrastructure Code" The 10th International Conference on Web Intelligence, Mining and Semantics (WIMS 2020). ACM, 2020

70. Sharma, Tushar, and Diomidis Spinellis. "A survey on software smells." Journal of Systems and Software 138 (2018): 158-173

71. Mülle, Jutta, Christine Tex, and Klemens Böhm. "A practical data-flow verification scheme for business processes." Information Systems 81 (2019): 136-151.

72. Xiang, Dongming, et al. "A guard-driven analysis approach of workflow net with data." IEEE Transactions on Services Computing (2019).

73. Liu, Cong, et al. "Petri net based data-flow error detection and correction strategy for business processes." IEEE Access 8 (2020): 43265-43276.

74. https://www.docker.com/
75. https://sylabs.io/singularity/
76. https://hpc.github.io/charliecloud/index.html
77. https://github.com/eth-cscs/sarus
78. Frajberg D., Fraternali P., Torres R.N. (2017) Convolutional Neural Network for Pixel-Wise Skyline Detection. In: Lintas A., Rovetta S., Verschure P., Villa A. (eds) Artificial Neural Networks and Machine Learning – ICANN 2017. ICANN 2017. Lecture Notes in Computer Science, vol 10614. Springer, Cham
79. D. Frajberg, C. Bernaschina, C. Marone, and P. Fraternali, "Accelerating deep learning inference on mobile systems," in International Conference on AI and Mobile Services. Springer, 2019, pp. 118–134.
80. https://github.com/intel-hadoop/HiBench
81. www.code-aster.org
82. Graph compilers for AI training and inference - blog
83. Apache Spark - Unified Analytics Engine for Big Data https://spark.apache.org/
84. DASK- Scalable Analytics in Python https://dask.org/
85. Open GPU Data Science | RAPIDS (https://github.com/rapidsai/cudf)
86. PETSC - Portable Extensible Toolkit for Scientific computation https://www.mcs.anl.gov/petsc/
87. MUMPS : a parallel sparse direct solver http://mumps.enseeiht.fr
88. https://www.cresta-project.eu/
89. https://www.maestro-data.eu/
90. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf
91. http://radon-h2020.eu/