Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

Application deployment and dynamic runtime - initial version

D5.1

ATOS

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	Application deployment and dynamic runtime - Initial version		
Authors	Román Sosa González (Atos), Mario Martínez Requena (Atos), Indika Kumara (JADS/UVT), Dragan Radolović (XLAB), Nejc Bat (XLAB), Kamil Tokmakov (USTUTT), Kalman Meth (IBM), Giovanni Quattrocchi (POLIMI)		
Reviewers	Dennis Hoppe (USTUTT) Damian A. Tamburri (UVT)		
Dissemination level	Public		
History of changes	Román Sosa González	Outline created	19.09.2019
	Román Sosa (section 1, 2, 7) All (section 2.1) Kamil Tokmakov, Dragan Radolović, Nejc Bat (section 3) Mario Martínez, Kalman Meth (section 4) Dragan Radolović, Nejc Bat (section 5) Indika Kumara, Giovanni Quattrocchi (section 6)	Initial partner contributions	20.12.2019
	All	Additional partner contributions	14.01.2019
	Román Sosa González	Adding references, captions, etc.	v1.0 15.01.2020
	All	Reactions to comments of first review	v1.1 24.01.2020
	Román Sosa (section 1, 2, 3.4.2, 3.5.2, 3.6, 4.4) Kamil Tokmakov (section 3) Nejc Bat (section 3.4.1, 3.5.1, 3.6) Mario Martínez (sections 4.1, 4.2, 4.5, 4.6) Kalman Meth (section 4.2) Nejc Bat (section 5) Indika Kumara, Giovanni Quattrocchi (section 6)	Addressing revision of deliverable request by EC review report	v2.0 03.07.2020
	All	Reactions to comments of second internal review	v2.1 17.07.2020

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 reports on the development status after M12 of the SODALITE Runtime Layer and the integration of its components with the rest of the SODALITE platform. This is the first of three deliverables in the context of WP5, to be released annually during the runtime of the project. This deliverable complements D3.1[14] and D4.1[15], also released at M12, and the interested reader is encouraged to read these deliverables to get a better understanding of the overall technology stack of the SODALITE platform.

The main focus of the deliverable is to present the objectives and challenges that need to be addressed by the Runtime Layer, the components that compose the architecture of the Runtime Layer, and the expected innovation.

The Runtime Layer has three objectives: (1) orchestration of the deployment of applications on heterogeneous infrastructures, (2) collection of runtime monitoring information, and (3) adaptations of applications for performance improvements.

To achieve these objectives, SODALITE took the approach of distributing the software to be deployed taking advantage of Lightweight Runtime Environments, and decided to use Singularity for managing the HPC infrastructure and Docker for the rest of infrastructures.

The Runtime Layer currently supports the deployment of containerized applications in OpenStack and (preliminarily) HPC, performs monitoring at infrastructure level and is progressing in the refactoring/adaptation activities. On adaptation, the NodeManager component is able to manage vertical scalability of Kubernetes containers to meet a specific SLA.

The next steps of the platform development comprise the integration of the different components and improving the deployment on HPC, the support of TOSCA workflows, extending the monitoring to application level and the finalization of the refactoring components.

Glossary

Acronym	Explanation
AADM	Abstract Application Deployment Model
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 Programming Interface
APM	Application Performance Monitoring
AWS	Amazon Web Services
CAMP	Cloud Application Management for Platforms
CI/CD	Continuous Integration / Continuous Deployment
CIMI	Cloud Infrastructure Management Interface
CPU	Central Processing Unit
CSAR	Cloud Service Archive
DSL	Domain-Specific Language
FaaS	Function as a Service
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HPA	Horizontal Pod Autoscaler
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
ID	Identifier
IDE	Integrated Development Environment
IPMI	Intelligent Platform Management Interface

ISO	International Organization for Standardization
JSON	JavaScript Object Notation
KVM	Kernel Virtual Machine
LRE	Lightweight Runtime Environment
ML	Machine Learning
NSM	Network Service Mesh
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
OCCI	Open Cloud Computing Interface
OCI	Open Container Initiative
Ops	Operations
OS	Operating System
OVN	Open Virtual Network
PBS	Portable Batch System
PCI	Peripheral Component Interconnect
PID	Proportional–Integral–Derivative
POSIX	Portable Operating System Interface
QoS	Quality of Service
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
SCSI	Small Computer System Interface
SLA	Service Level Agreement
SSH	Secure SHell
TOSCA	Topology and Orchestration Specification for Cloud Applications
TUN	network TUNnel

UC	(UML) Use Case
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
UUID	Universally Unique IDentifier
Veth	Virtual Ethernet
VM	Virtual Machine
VPA	Vertical Pod Autoscaler
YAML	YAML Ain't Markup Language

1 Introduction

We are now in an era of heterogeneous, software-defined, high-performance computing environments: cloud servers, GPUs, FPGAs, Kubernetes, FaaS, etc. Complex applications use complex deployments, where a component can be installed on an infrastructure that offers the best performance, depending on its requirements. In this context, SODALITE aims to address this heterogeneity by providing tools that allow developers and infrastructure operators to enable faster development, deployment and execution of applications on different heterogeneous infrastructures [5]:

- a pattern-based abstraction library with support for application, infrastructure and performance abstractions;
- a model for designing and programming infrastructures and applications, based on the abstraction library;
- a deployment platform that statically optimizes the abstract applications on the target infrastructures;
- automated optimization and management of applications at runtime.

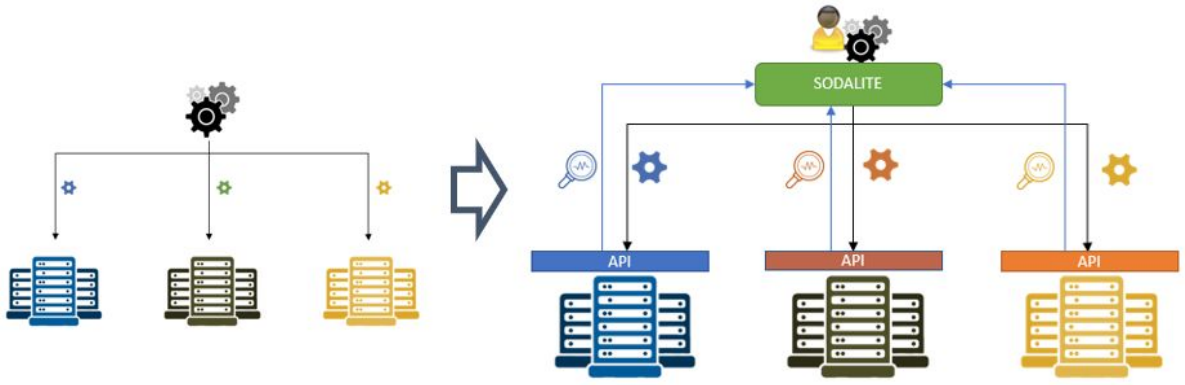

Figure 1 - SODALITE helps to manage complex deployments

In particular, the Runtime Layer of SODALITE is responsible for the orchestration, monitoring and refactoring of applications on these infrastructures. The objectives of the Runtime Layer are:

- **Orchestrating the initial deployment of an application**. The Runtime Layer gets the TOSCA[1] blueprint of an application and performs the deployment of each of the software components on the specified targets, which may be heterogeneous (e.g. an Nginx server on a VM on Openstack and a AI training job to be run by a Torque workload manager on an HPC cluster).
- **Collecting runtime monitoring information and logs at different levels**: application, runtime environment and infrastructure. With this information, it is possible to understand issues related to the application's performance, and react in a manual or automated way.
- **Enabling adaptation of the application to improve its performance**. In order to make an application fulfill its performance goals, different mechanisms are applied at runtime. E.g. vertical or horizontal scalability, migration and topology changes.

This deliverable presents the initial specification and implementation during year 1 of the development of the SODALITE Runtime Layer, focusing on the concepts, principles and existing

[1] https://www.oasis-open.org/committees/tosca

tools that are relevant to the WP5 results, the design of the different systems that compose the Runtime Layer and the achieved and expected innovations. The deliverable D2.4 - Guidelines for Contributors to the SODALITE Framework [60] elaborates on the quality of the developed artefacts and the guidelines for software and release management.

Throughout the document, we are using the terms Application Ops Experts (AOE), Resource Experts (RE) and Quality Experts (QE). The following table provides a mapping between these roles and the processes defined in the ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes:

SODALITE Roles	ISO/IEC/IEEE standard 12207 processes
Application Ops Experts (AOE)	Operation processes and maintenance processes
Resource Experts (RE)	Infrastructure management and Configuration management processes
Quality Experts (QE)	Quality Management and Quality assurance processes

The document is structured as follows:

- Section 2 gives an overview of the SODALITE architecture and its *Runtime Layer*, provides a functional description and status at M12 of the building blocks in the *Runtime Layer*, and explains the expected innovation contributed by SODALITE.
- Sections 3 to 6 report about the *Runtime Layer* tasks. They all follow the same structure: (1) an initial introduction addressing the concepts or methods relevant for the task outcome; (2) an state of the art that describes the current innovation in the topic and how the proposed solution advance the state of the art; (3) the UML use cases that are covered, followed by a description of the sequence diagrams for each of the use cases; (4) a description of the architecture and the components in development; (5) the development status of the components; (6) the next steps in the development of each task.
    - Section 3 describes the deployment and reconfiguration of the SODALITE applications on heterogeneous infrastructures.
    - Section 4 describes how SODALITE applications are monitored during execution time.
    - Section 5 describes the concept of the Lightweight Runtime Environment and technologies being considered for its development.
    - Section 6 describes the techniques and tools to refactor SODALITE applications to improve their performance.
- Section 7 summarizes the document, offering the current status of the WP status and next steps to be addressed.
- Appendix A shows the list of metrics currently supported by the SODALITE Monitoring.
- Appendix B shows a description of the software baseline used for the *Runtime Layer* developments, which were already presented in D6.1[16].

2 The SODALITE Runtime Layer

The section gives an overview of the SODALITE and *Runtime Layer* architectures, including the relationships that exist between the *Runtime Layer* and the other blocks of SODALITE. Then, it provides a functional description of the building blocks in the *Runtime Layer,* including the challenges tackled by the respective building block and the development status at M12. The section finishes explaining the expected innovation contributed by SODALITE.

2.1 Overview of the SODALITE architecture

The SODALITE architecture, presented in D2.1 - Requirements, KPIs, evaluation plan and architecture - First version[5], is shown in Figure 2.

The SODALITE architecture is divided into three main subsystems, corresponding to (i) design and modeling (**Modeling Layer**), (ii) generation of IaC and static optimizations (**Infrastructure as Code Layer**) and (iii) runtime execution and optimizations (**Runtime Layer**). The *Modeling Layer* is the interface to end users (Application Ops Experts, Resource Experts and Quality Experts), where they can design an abstract deployment model and obtain information about the application at runtime. The *IaC Layer* takes the abstract deployment model and generates Infrastructure as Code for deployment, verifying the IaC to predict defects and performing static optimizations on the applications. The *Runtime Layer* takes the IaC blueprint as input and manages the deployment of the application, monitors the execution and suggests alternative deployments to optimize the execution.

SODALITE applications are executed inside a Lightweight Runtime Environment (**LRE**). Concretely, each application artifact is in the form of a container (Singularity for HPC, Docker for the rest of cases) and executed by the corresponding container engine.

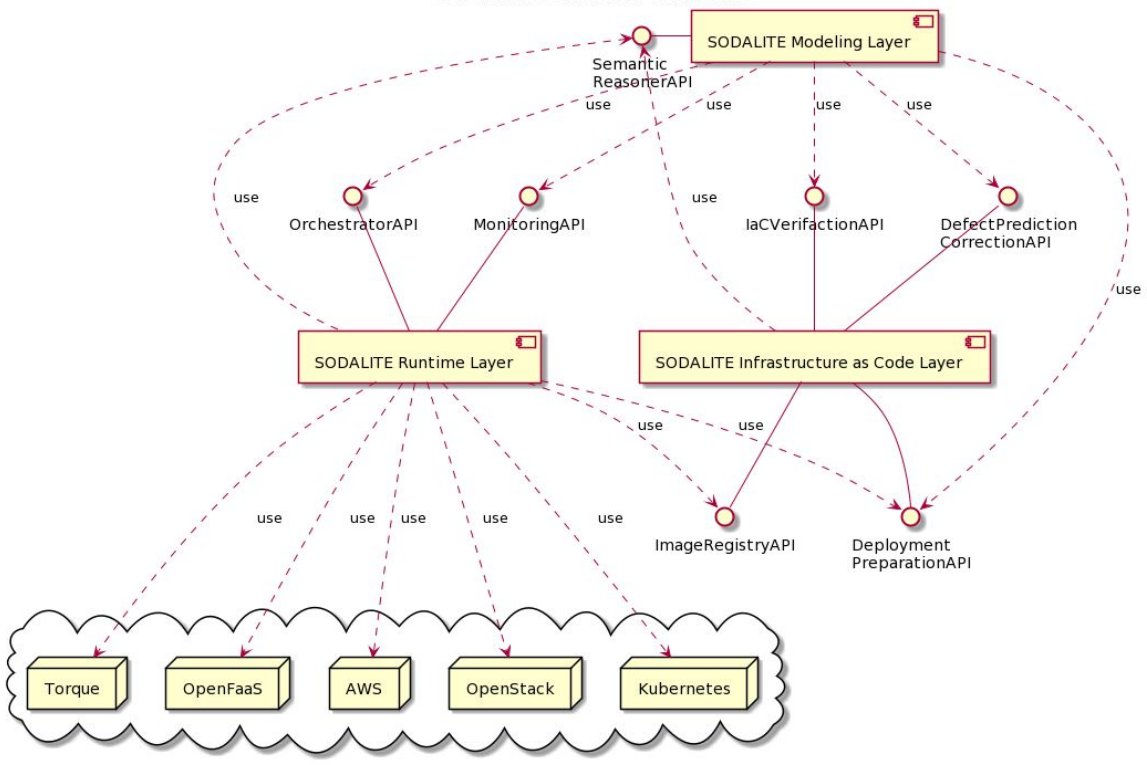

Figure 2 - SODALITE architecture

2.2 Runtime Layer architecture

To address the three main objectives of the *Runtime Layer* presented in the introduction (orchestration, monitoring, adaptation), this layer is composed of three building blocks (see Figure 3), corresponding to each of the objectives.

- The ***Orchestrator*** block is composed of the orchestrator itself and a set of components that facilitate the deployment and reconfiguration of an application, and the management of a specific infrastructure. Infrastructures are managed via their specific execution platform manager. We are targeting to support the infrastructures shown in the figure.
- The ***Monitoring*** block is composed of a monitoring server and a set of probes that retrieve metrics from the different monitoring targets: VMs, HPC nodes, runtime environment, applications, etc.
- Finally, the ***Refactoring*** block, responsible for applying adaptations to applications to improve its performance, is composed of the *Refactoring Option Discoverer*, the *Node Manager* and the *Deployment Refactorer*.

The terms reconfiguration and refactoring are used in this document as follows:

- **Reconfiguration** is the action applied to a deployment to modify an undesired or improvable state. It may be a simple modification (like restarting a crashed service) or a complex modification (like changing the topology of the application). Reconfigurations are applied by the *Orchestrator*.
- **Refactoring** is the action that proposes a change in the deployment model of the application; change that modifies the quality attributes of the application without changing the application functionality. In this sense, refactorings are proposed by the *Refactoring*, while the *Orchestrator* is the component that applies the changes to (i.e. reconfigures) the application deployment.

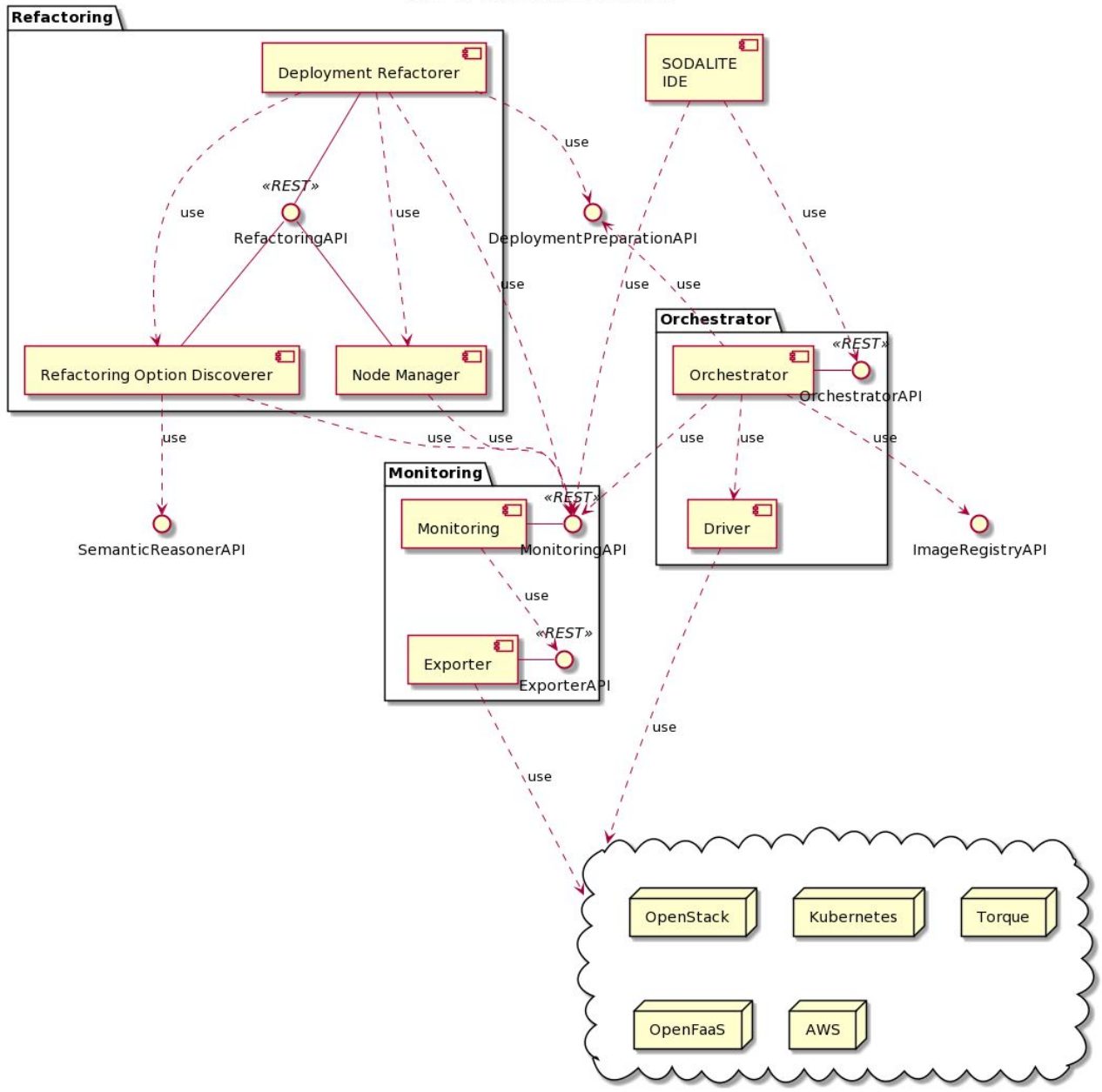

Figure 3 - Runtime Layer Architecture

As shown on the general SODALITE architecture (see Figure 2), the *Runtime Layer* components need to interact with other components in the SODALITE platform to fulfill their objectives. Specifically, the WP5 components depend on:

- **IDE**[14]. The IaC blueprint is sent to the *Orchestrator* via the IDE, where the Application Ops Expert approves the deployment. Also, any modification in the deployment must be approved by the Expert. With regards to the *Monitoring* block, the users will be able to visualize monitoring information in the IDE.
- **Deployment Preparation**[15]. Given that the input to the *Orchestrator* is the ID of a blueprint, the *Orchestrator* gets from the *Deployment Preparation* the actual content of the blueprint, i.e., the TOSCA file and the Ansible playbooks.

- **Semantic Reasoner**[14]. The *Refactoring* block uses the model contained in the Semantic Reasoner to discover additional deployment alternatives. These alternatives are saved back to the Semantic Reasoner.

The following subsections provide a functional description and status at M12 of the components in the *Runtime Layer*. For more details about them, refer to sections 3, 4 and 6.

2.2.1 Orchestrator

The *Orchestrator* is in charge of executing deployment and reconfiguration of SODALITE applications. It takes an IaC blueprint (i.e., a TOSCA+Ansible playbooks) and performs the deployment of the applications on heterogeneous infrastructures. The infrastructures or middlewares to be supported are public/private clouds (e.g. AWS, OpenStack), batch systems used in HPC (e.g. PBS Torque, Slurm) and other orchestration systems such as Kubernetes and OpenFaaS. In case of deployment refactoring either triggered by the runtime optimization or application iterations, the *Orchestrator* makes the necessary adaptations to the deployment. For example, moving a module from one resource to another or the software updates for the module. See Section 3.4 for a detailed description of the *Orchestrator*.

The main challenge to face is the automation of the different management actions (provisioning, configuration, data management, authentication & authorization and administration) over the heterogeneous resources (cloud, HPC and edge) offered by different infrastructure providers. Moreover, the deployment over heterogeneous resources must be accomplished effectively, meaning that the deployment process should be executed in parallel whenever possible.

Currently, the *Orchestrator* supports the management of applications in OpenStack and support of Torque HPC clusters, accessed by SSH. A REST API has been added to the Orchestrator, which facilitates its invocation by other components. It is also able to perform basic reconfiguration (e.g., migration of an application to another VM) if a redeployment is triggered and the blueprint has changed.

2.2.2 Monitoring

It gathers metrics from the application execution at different levels: application level, runtime environment level and infrastructure level. The information collected by monitoring will be used by *Refactoring* and will be available to the SODALITE experts though a dashboard. See Section 4.4 for a detailed description of the *Monitoring*.

The main challenge is to fill the gap between application metrics and the system metrics, so SODALITE is able to relate both kinds of metrics to gain a better understanding of the application behaviour. The system metrics are provided by the cloud providers, which monitor their infrastructure (CPU and memory in virtual and bare metal machines, network performance, etc), but having a good performance of application at the level of a virtual machine does not mean that the performance is good at the application level, due to other factors, like multitenancy.

Currently, the *Monitoring* is able to collect infrastructure metrics, including consumption and networking metrics. For this activity, a Skydive analyzer (networking metrics) and a Prometheus server have been installed on Cloud Testbed, together with NodeExporter (multiple VM metrics) automatically installed on new VMs and the installation of a new developed IPMI Exporter (consumption metrics) on the physical nodes of the Cloud Testbed. We are using Grafana as a visualization tool.

2.2.3 Deployment Refactorer

This component refactors the deployment model of an application in response to violations in the application goals. It also derives the node-level goals from the application goals. The goals are monitored at runtime by collecting the necessary metrics. A machine learning based predictive model is used to select a valid set of refactoring options to derive a valid variant of the deployment model (the new deployment model). The new abstract deployment model is transformed to TOSCA and IaC Scripts using the *Deployment Preparation* module. The new refactoring options as well as the changes to the existing refactoring options can be discovered at runtime. See Section 6.4.1 for a detailed description of this component.

In SODALITE, we consider heterogeneous applications, which can be deployed differently using alternative deployment options for the individual components of the application. The deployment model/topology of such an application is a set of alternative deployment model variants, each having different performance and other non-functional characteristics. There may exist many such variants, and the application needs to be able to switch between these variants at runtime for reasons such as reducing cost and resource usage, and preventing violations of performance goals. Consequently, the key challenges for the deployment refactoring are:

- Predicting the impact of a given deployment model variant on the quantitative metrics such as response time and throughput based on the historical metric data for a subset of other variants.
- Selecting a deployment model variant most suited at a given moment (e.g., for a specific workload range) and switching the current deployment to the selected variant (i.e., deployment adaptation).
- Taking into account security and privacy vulnerabilities (e.g., GDPR violations and security anti-patterns) and performance anti-patterns in deployment model variants.
- Responding to ad-hoc events generated by the application and the *Orchestrator*.
- Using the deployment options discovered by the *Refactoring Option Discoverer*. This changes the number of possible deployment variants for the application.

Currently, the *Deployment Refactorer* provides an ECA (event-condition-action) rules based refactoring decision making and an initial machine learning based model for performance-driven refactoring, which has been tested with Google Cloud and RUBiS cloud benchmark application

2.2.4 Node Manager

The *Node Manager* component, part of the *Refactoring* building block, is responsible for re-configuring at runtime deployed containers in order to fulfill requirements on the response time (e.g., response time < 0.5s). The re-configuration is carried out using control-theory based planners that dynamically change the allocated CPU cores and GPUs of running containers (i.e., vertical scalability). See Section 6.4.2 for a detailed description of this component.

The main challenges to face in the design and implementation of this component are:

- Managing heterogeneity. CPUs and GPUs are inherently different. GPUs are faster but cannot be allocated as fine-granular as CPUs. In fact, GPUs can be only allocated entirely while CPUs can be shared among different processes. Moreover, CPUs are used to load and instruct GPUs consuming part of their processing power.
- Managing fast-changing environments. Nowadays applications are highly dynamic and their resource allocation must fast enough to precisely follow the incoming workloads and changes in the execution environment.
- Managing concurrent applications. Modern architectures (e.g., microservices) deploy multiple applications in the same cluster and in the same machine. Controlling a single

application is not enough and multiple concurrent processes with different requirements must be overseen with a comprehensive solution.

Currently, the *Node Manager* is able to perform dynamic allocation of GPUs and CPUs among TensorFlow applications deployed in a Kubernetes cluster to prevent response time violations.

2.2.5 Refactoring Option Discoverer

This component is responsible for discovering new refactoring options and changes to existing refactoring options based on design patterns and anti-patterns (in general, topology level defects). For example, a new instance of an infrastructure design pattern that may offer better performance or security may be found. See Section 6.4.3 for a detailed description of this component.

The key challenges for Refactoring Option Discoverer include:

- discovering TOSCA-compliant resources and deployment model fragments using deployment design patterns;
- predicting the impact of a given discovered resource/fragment on the performance and other quantitative metrics of the application if the application uses them;
- taking into account security and privacy risks of the discovered resources/fragments, for example. The resource matchmaking needs to consider the deployment policies (specified as TOSCA policies) of the resources and security and privacy implications (e.g., GDPR violations) of such policies.

Currently, the *Refactoring Option Discoverer* is able to perform discovery of refactoring options based on ontological reasoning.

2.3 Runtime Layer Workflow

A simplified view of the *Runtime Layer* workflow is shown in Figure 4. The pink boxes represent the WP5 building blocks, while the white box represents the *Deployment Preparation*, from the *IaC Layer*. The whole WP5 workflow starts when the *Deployment Preparation* generates the IaC blueprint, which is sent to the *Orchestrator* after it has been approved by the Application Ops Expert.

The *Orchestrator* performs the deployment of each of the application modules onto the corresponding deployment targets and starts them. Then, the deployment is continuously monitored at different levels (infrastructure, LRE and application). At the same time, the *Refactoring* is evaluating the monitoring information to decide if a reconfiguration or a refactoring action is needed. In case of a reconfiguration action (which may be triggered by the *Refactoring* or by the *Orchestrator* itself), it is applied by the *Orchestrator*. In case of a refactoring action, where the *Refactoring* proposes an alternative deployment configuration, and involves modifying the deployment, the workflow transitions to the *Design Layer*. There, when the Application Ops Expert approves the new deployment in the IDE, a new IaC blueprint is generated and the WP5 workflow starts again: this time, the Orchestrator executes the partial redeployment to reflect the changes with regards to the previous deployment.

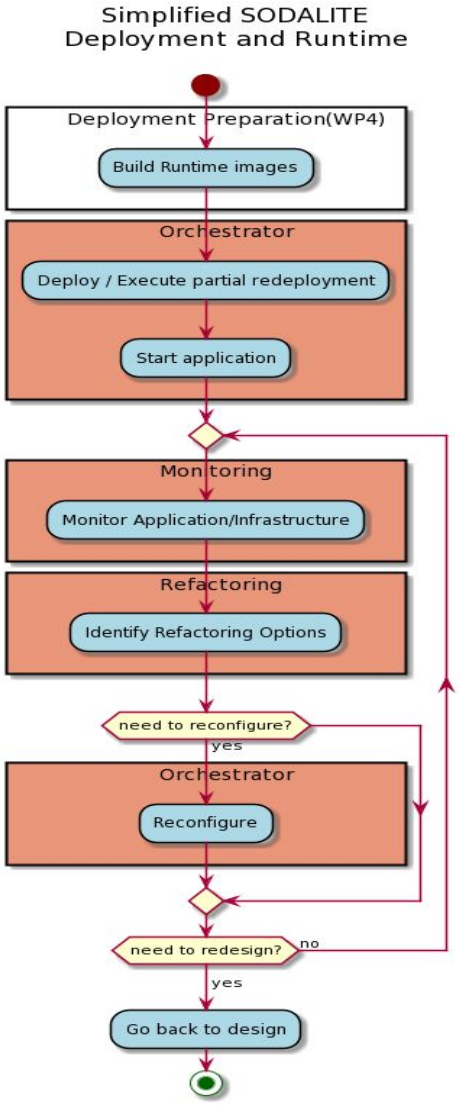

Figure 4 - Deployment and Runtime workflow

2.4 Innovation

This section presents an overview of the expected innovation in SODALITE. A more complete description and references to the state of the art can be found in sections 3.2, 4.2, 5.2 and 6.2.

The SODALITE *Orchestrator* enables the deployment of applications on heterogeneous infrastructures, based on blueprints compliant with TOSCA YAML v1.3. The usage of standards is key to avoid the vendor lock-in problem, but the same problem arises if existing TOSCA solutions (like Cloudify) use their own DSLs for e.g. deployment modelling, which are described in the TOSCA standard. Also, current initiatives are not fully TOSCA compliant, not open source, or specific features (like multi-tenancy or updating a deployment) are not free. The SODALITE *Orchestrator* is easily extendable, as it is IaC-based: as long as a platform can be modelled with a standard modelling language and actuation engines (e.g. Ansible), the support for it can be provided.

The SODALITE *Orchestrator* differs from other approaches that follow an intrusive architecture that require modifications to the infrastructure configuration. Our approach is to orchestrate resources via the existing resource managers and execution platforms.

Besides this, the support for TOSCA workflows for the declaration of workflows of HPC jobs is in progress, advancing with regards to other tools like Croupier, which use a non-standard approach to achieve the same objective.

To extend the number of workload managers supported by SODALITE, we plan to use ALDE as a uniform REST interface that interacts with them (currently, Slurm). ALDE is being enhanced not only to support Torque but also to provide multi-tenancy. There is no existing solution that provides similar functionality.

We have a novel approach for the deployment optimization problem. We adopt the dynamic software product lines view, finding the optimal deployment variant between a set of allowed variants. The behavior and performance of the variants are modeled using a ML approach, while we perform dynamic discovery of new deployment options using semantic matchmaking and search-based heuristics. Another improvement is that SODALITE will also detect performance anti-patterns, being able to correct them in application deployments.

For the management of Kubernetes clusters, we intend to use the *Predictive Deployment Refactorer* for horizontal scaling, replacing the Kubernetes HPA, while using the *Node Manager* for vertical scaling, overriding the Kubernetes VPA. VPA is currently in beta (as of June 2020) and cannot work concurrently with HPA[2]. SODALITE is ahead of VPA and HPA because (i) it allows for both horizontal and vertical scalability at the same time (ii) its vertical scalability does not need pod rebooting and it is faster.

Refactoring actuations are driven by the metrics collected by the monitoring system. In this sense, monitoring is working to improve the refactoring actions in general, and in HPC in particular, by providing networking monitoring information and HPC jobs-related information (in the form of an HPC exporter for Prometheus) as part of the decision process to optimize the deployment of applications.

2 https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler#limitations_for_vertical_pod_autoscaling

3 Deployment and reconfiguration

The deployment and reconfiguration task manages the lifecycle of applications deployed on heterogeneous and hybrid infrastructures, i.e., HPC, Cloud, Kubernetes, etc. This requires the use of cross-systems orchestrators. The SODALITE orchestrator consumes IaC (TOSCA blueprints and Ansible playbooks) generated by the WP4 Deployment Preparation tool (described in D4.1[15]) and performs the deployment or reconfiguration of the application, initiating the collection of monitoring metrics.

In this section, we first describe the general concepts of and requirements for cross-systems orchestration, followed by the related work, which highlights our research contributions. Next, we present the architecture of our orchestrator building block, followed by a detailed description of each major component. Finally, we present the current development status and the deployment plan for each component.

3.1 Background

In the SODALITE project we orchestrate the lifecycle of a heterogeneous application, the components of which are executed on various execution platforms, such as HPC, Clouds and Edge Computing, over the course of runtime. Moreover, apart from the CPU, the execution platforms can offer specialized hardware, such as GPUs and FPGAs, and the heterogeneous application can utilize them to further accelerate the performance. These execution platforms provide remote endpoints for user access, operation and usage. For example, HTTP/HTTPS based REST APIs are widely used for Clouds, and SSH based remote access to the HPC clusters is common. Therefore, the task of the orchestrator is to employ these endpoints in order to deploy and reconfigure the components of the application onto the respective platform and hardware.

In the following subsections we provide a brief background on the execution platforms and introduce the key concepts for SODALITE orchestration.

3.1.1 Execution platforms

As the first prototype, we consider the support of not all, but only selected execution platforms and environments, namely OpenStack, Kubernetes and Torque. As such, OpenStack represents the IaaS model, Kubernetes represents the orchestrating system for containerized workloads and Torque is an HPC workload manager handling batch jobs, hence a variety of environments is considered. The description of these platforms in the SODALITE testbeds are presented in the deliverables D6.1[16] (Section 2.4) and D6.2[17] (Section 2.1).

OpenStack. OpenStack provisions virtualized compute resources, e.g. virtual machines, storage, and networks, via its services, meaning that before the deployment of the application, the virtualized infrastructure needs to be previously established. Each OpenStack service provides a REST API for its management. For example, in order to create a virtual server, an HTTP POST request needs to be sent to the OpenStack endpoint with the parameters (e.g. image, flavor, network, SSH key) of the server in JSON format:

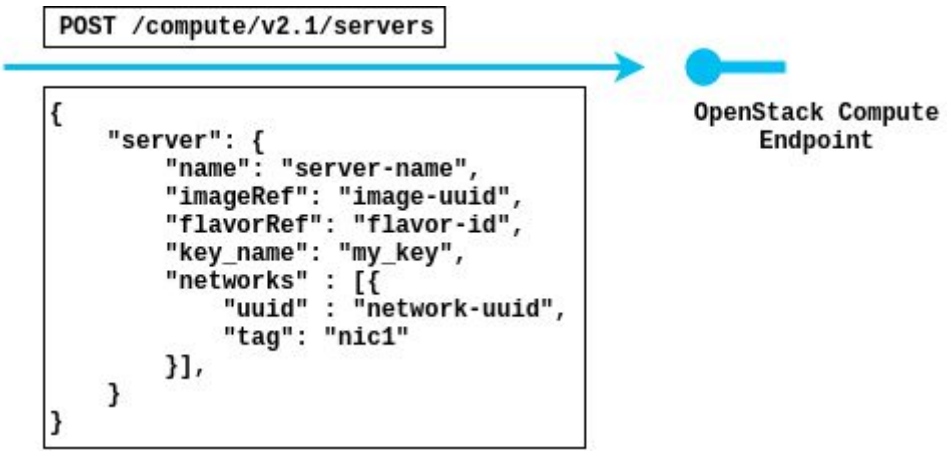

Figure 5 - Create a VM using the Openstack REST API

The response is sent back containing the information about the server, e.g. its UUID. The deployment status of the server can be issued via HTTP GET request to the same path as shown above. After the server is created, it can be accessed via SSH to further deploy the application's components and dependencies.

Kubernetes. Being an orchestrator itself, Kubernetes automates lifecycle management of the containerized applications, which are encapsulated into Pods - Kubernetes deployment units. Commonly, a declaration file that defines a desired deployment state of the application is submitted to the control plane of Kubernetes, which then ensures the compliance of the actual state to the desired one. As such, the file declares the image of the application, needed resources and replicas, as well as interconnection of the Pods into Services. The container technologies conforming OCI, such as Docker and Singularity, and their image formats can be used in Kubernetes to deploy and provide an execution runtime for the application.

As an example, consider the deployment of a Nginx web-server in Kubernetes - the declaration, shown below, specifies the selectors and labels for the particular deployment along with container parameters: Nginx Docker image pulled from the remote Docker registry, port mapping and number of replicas. The declaration is then serialized in the JSON format and submitted to the Kubernetes API via HTTP POST request:

Figure 6 - Create a deployment using the Kubernetes REST API

In order to obtain the status of the newly deployed pods, the HTTP GET request can be sent to the Kubernetes API.

Torque. It is a batch system and workload manager, operating the compute resources, storage and other hardware used in traditional HPC systems. A user connects to the front-end (login) nodes via SSH and submits the job specifying various parameters, such as environment, maximum execution time (walltime), number of nodes, processors and GPUs together with the script executing the application workflow. An example below shows a PBS job script that requests two nodes with twenty processors per node and an hour to execute a compiled application (app.exe). Then, the job script is submitted to Torque via ***qsub*** command to be executed on the requested resources.

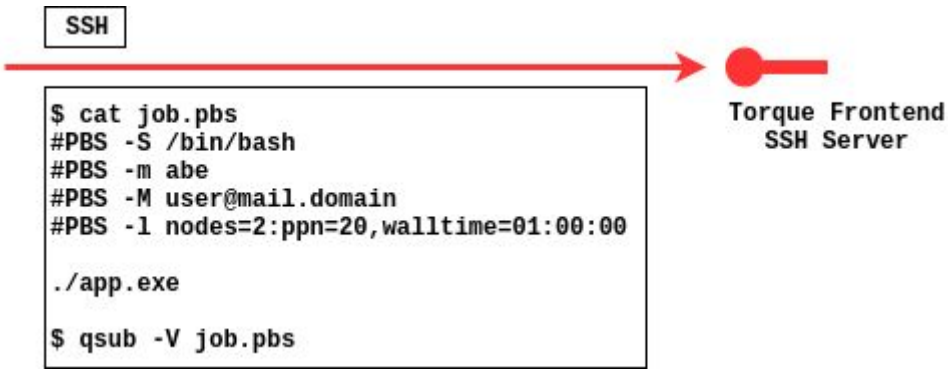

Figure 7 - Example of PBS job

After the job submission, the ID of the job is returned, which can be passed to the ***qstat*** command to get the status of the job.

3.1.2 IaC-based meta-orchestration

Infrastructure-as-Code (IaC) refers to the automated management and provisioning of an infrastructure for the application stack deployment using software and respective definition files rather than manual setup and configuration. Although there exist plenty of IaC tools, Ansible is becoming a de facto standard for infrastructure management. It has a concept of reusable playbooks, inventories and modules to perform repeatable (idempotent) deployment tasks on a particular infrastructure set, without the need for special agents to be preinstalled on the targeted infrastructure (unlike other tools like Chef or Puppet).

Ansible provides several convenient modules, which enable interaction with the particular platform. For OpenStack there are several modules available[3], allowing the creation of various components of the virtual infrastructure: virtual machines - *os_server*, networks - *os_networks*, block storage - *os_volume*, etc. The dedicated module - *k8s*[4] - for management of Kubernetes objects allows creation of deployment pods and services. As for Torque, there are no modules; however, the shell module[5] supports executing commands on the remote cluster, e.g. the ***qsub*** command.

However, centralized management of multiple execution platforms at scale becomes cumbersome with Ansible, especially with the dependencies between different playbooks, since it is still lacking a robust and elegant way for inter-playbooks coordination. Such coordination involves management of different playbooks and should be done on a higher level of orchestration.

Orchestration is the automated (re)configuration, coordination, and management of computer systems and software. It oversees the deployment and runtime of all the components of an application in the infrastructure. Additionally, it can perform other tasks like healing (manage errors), scaling and logging. The management of underlying virtual/baremetal resources are performed by execution platforms explained in the previous subsection, which can be seen as low-level resource orchestrators, hiding complex processes of resource management behind their REST API and other interfaces. In this context, meta-orchestration refers to indirect application lifecycle and resource management through these low-level resource orchestrators, as shown in Figure 8.

[3] https://docs.ansible.com/ansible/latest/modules/list_of_cloud_modules.html#openstack
[4] https://docs.ansible.com/ansible/latest/modules/k8s_module.html
[5] https://docs.ansible.com/ansible/latest/modules/shell_module.html

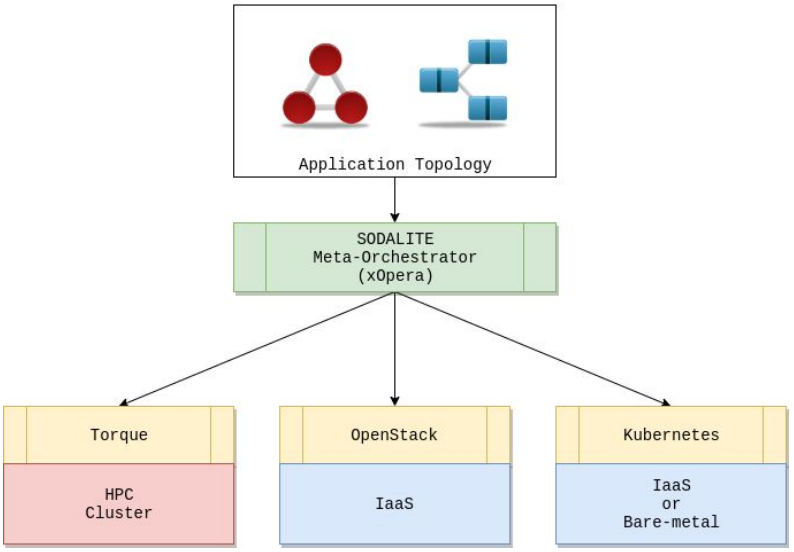

Figure 8 - Meta-orchestrator

Interfacing various execution platforms imposes a set of requirements that the meta-orchestrator should satisfy:

- **Agentless architecture.** The use of agents is not possible / recommended, as those agents have to be integrated in the lower-level orchestrators. Therefore, the meta-orchestrator needs to establish communication with the layer below without the use of agents.
- **Application topology definition based on open standards.** The meta-orchestrator needs to understand standard definitions that allows the application developer to describe a portable application topology. The usage of standardized DSL also benefits in extensibility of supporting other platforms - instead of extending the source code of the orchestrator, a new definition using the same DSL can be provided to support other platforms.
- **Interoperability of hybrid infrastructures.** The orchestrator should simultaneously target multiple infrastructure types to utilize characteristics of a particular type to achieve best performance, e.g. usage of HPC for compute intensive tasks and edge computing for latency sensitive services. As a consequence, the interoperability should be performed on multiple levels: data management, authentication and authorization procedures, and computation.
    - The data movement across the infrastructures should be performed seamlessly, i.e. the orchestrator needs to be aware of different types of data transfer protocols and services offered by various infrastructure providers and transparently adapt the deployment for the usage of particular data transfer clients and tools.
    - Similarly, the authentication and authorization procedures should also be abstracted from the user.
    - Container as an application component enables computation interoperability between multiple computing hosts.
- **Reconfiguration.** Due to the fact that the application development and deployment are nowadays continuous, shipping new releases frequently, the orchestrator should cope with the partial reconfiguration and redeployment of the application in an efficient way, i.e. updating only affected components of the application topology. Alternatively, the reconfiguration can be triggered on the infrastructure level in order to satisfy QoS

parameters, e.g. increase of responsiveness of the application by provisioning greater resources, hence the orchestrator should handle this task as well.

In order to address these requirements, we developed an orchestration service (presented later in Section 3.4) around the open source lightweight orchestrator - xOpera. It is driven by Ansible and utilizes OASIS TOSCA standard, a widely-adopted industry-developed and supported standard for the orchestration of cloud applications, as a high-level standard definition of application topologies. Due to agentless execution of Ansible, xOpera also follows agentless architecture.

With the large set of IaC blueprints (TOSCA+Ansible) and container images generated by the SODALITE IaC Management Layer, we target interoperability and optimal deployment over hybrid/heterogeneous infrastructure. The orchestration system also addresses reconfiguration with the versioning and management of the deployment state.

To put it together, an example of the orchestration involving multiple execution platforms, specifically related to the Vehicle IoT use case and its continuous ML model delivery (MLOps[6]) to the Edge, is depicted in Figure 9. For brevity, ML processes such as data analysis or model verification/validation are omitted. A meta-orchestration of an ML-based service on Edge is considered, where the continuous training process is offloaded to more compute-capable HPC clusters as part of MLOps actions. The orchestrator initially deploys the service to Edge (Kubernetes) and then performs an MLOps cycle. It transfers training data to the HPC cluster, submits the job for ML training and monitors the job execution. After the job is executed, the inference model can then be transferred by the orchestrator via data management utilities and integrated into the business logic of the service at run-time, thus completing the MLOps cycle. At the next request, the orchestrator repeats the cycle.

6 https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

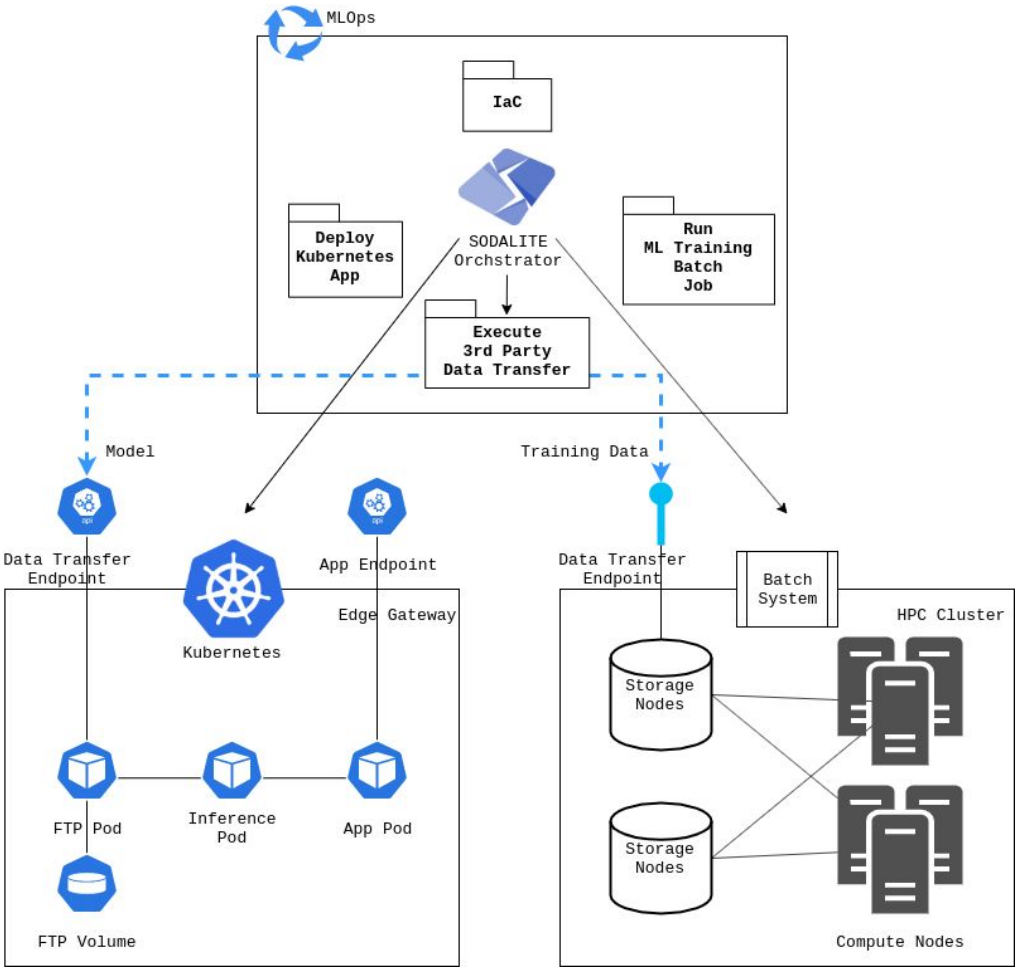

Figure 9 - An example of cross-system orchestration between Kubernetes and HPC batch systems: microservice application deployment, submission of the batch job and data transfer between the systems

3.1.3 Enabling IaC-based Monitoring, Refactoring and Deployment Failure Handling

Apart from the deployment and reconfiguration, the orchestrator prepares endpoints and environment for other SODALITE components of the *Runtime Layer*. In order to conform to the QoS/SLA, monitoring is needed. *Predictive Deployment Refactoring* (see Section 6) is a key component for runtime optimization, which relies on the monitoring; therefore the collection of the monitoring metrics needs to be established.

It should be noted that the orchestrator operates only with IaC, therefore SODALITE *IaC Management Layer* must prepare IaC blueprints that install monitoring exporters or container images that encapsulate them, depending on what level (application, container, operating system or infrastructure) needs to be monitored. Based on these IaC artifacts, the orchestrator will setup, configure and connect the monitoring agents and exporters to the centralized monitoring server during the deployment time. Section 4 introduces the monitoring framework for the *Runtime Layer* of SODALITE and an analysis of the existing tools in the market.

With respect to the mechanisms of the deployment refactoring, the orchestrator needs to derive the differences between current and new deployments, and to apply these differences. The development is ongoing and it is expected to be released in M18. The orchestrator will utilize the capabilities of the execution platforms in order to achieve needed reconfigurations and ensure their validity. For example, the vertical scalability of the virtual machines in OpenStack can be

performed by the *resize*[7] (with instance shutdown) operation, and the *live-migration*[8] (without instance shutdown) mechanism can be used in order to migrate an instance into another node (e.g. because of the policy change or interferences with other instances of the same node).

The orchestrator should also handle failures occurring during the deployment of the application. Here again, we rely on the IaC-based failure management. The orchestrator expects failure handling tasks at the actuation level, meaning that the actuation playbooks shall contain Ansible directives[9] that detect possible failures and either retry the failed task or perform some other actions such as alerting. For example, the following snippet checks whether an endpoint is available as the post-deployment action and retries 10 times before failing the task:

```
- name: wait for endpoint to be responsive, otherwise fail
  wait_for:
       host: "{{ api_address }}"
       port: "{{ api_port }}"
       timeout: 60
       state: present
  register: check_endpoint
  until: check_endpoint is success
  retries: 10
  delay: 6
```

There exist the cases, when the application starts at the deployment time, such as HPC jobs deployment, that a successful execution of the deployment does not necessarily mean successful execution of the jobs. In the HPC cases, the workload managers often provide the status of the jobs (e.g. via ***qstat*** command for Torque), which can be automatically parsed and deducted. As such, the example below shows the parsed information of whether a job has been completed and its exit status. In this case, it indicates unsuccessful termination due to exit code of 1 (every non-zero exit code is treated as an error).

```
$ qstat -f JOB_ID | grep 'job_state' | grep -o '.$'
C

$ qstat -f JOB_ID | grep 'exit_status' | grep -o '.$'
1
```

The actuation part of xOpera is able to periodically poll for these checks via Ansible's Asynchronous Actions and Polling mechanism[10]. It will perform a task periodically multiple times until a certain condition is met. A sample below presents a list of tasks of checking the job completion and afterwards its exit code. It asynchronously polls the job status every 10 seconds for an overall 2000 seconds. Once it is finished, the exit code is parsed and Ansible fails the deployment, if the code is non-zero.

[7] https://docs.openstack.org/nova/latest/user/resize.html
[8] https://docs.openstack.org/nova/latest/admin/live-migration-usage.html
[9] https://docs.ansible.com/ansible/latest/user_guide/playbooks_error_handling.html
[10] https://docs.ansible.com/ansible/latest/user_guide/playbooks_async.html

```
- name: Run continuously qstat to monitor the status of the job
  shell: qstat -f {{ JOB_ID }} | grep 'job_state' | grep -o '.$'
  register: job_monitor
  until: "job_monitor.stdout == 'C'"
  delay: 10
  retries: 200
  async: 2000
  poll: 10

- name: Wait for job completion
  async_status:
    jid: "{{ job_monitor.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 200

- name: Check the exit status
  shell: qstat -f {{ JOB_ID }} | grep 'exit_status' | grep -o '.$'
  register: job_exit_status

- fail:
    msg: "Job stopped with non-zero exit {{ job_exit_status.stdout }}"
  when: "job_exit_status.stdout != '0'"
```

In any case, the orchestrator registers the deployment status as failed, and the status can be retrieved via the orchestrator REST API, thus notifying other SODALITE components, e.g. IDE, about the failed deployment.

The failure handling during runtime of the application can be addressed by monitoring and deployment refactoring, triggering alerts and failure mitigation actions (e.g. redeployment) via TOSCA v1.3[11] standard, which defines Policies, Triggers and Workflows directives to be executed by the orchestrator. These allow an orchestrator to execute policies by checking a condition (e.g. a service is down) and based on that, trigger an event or workflows (e.g. restart the service). This possibility will be explored more in the next years of the project.

Following this approach, many use case specific status-checkers can be developed and integrated into the generated IaC as part of the failure handling of the application deployment in order to ensure their proper functioning at runtime. Furthermore, the future versions of the orchestrator will allow migration to the TOSCA-defined failovers.

3.2 Related work

Orchestrating over multiple infrastructures is not a novel concept. Many researchers tried to utilize the capabilities and capacities of specific infrastructure to enhance the performance of application [43, 44]. As such, there were attempts to utilize Cloud computing for HPC to create so-called elastic clusters [45], or integrate Fog/Edge computing to enable low-latency computations at the network edge [46, 47]. However, these approaches follow specific intrusive architecture requiring modification in infrastructure configuration (e.g. VPN-based elastic clusters [43, 45]), whereas the approach we undertake is to orchestrate these resources via the existing resource managers and execution platforms (meta-orchestration) [48], since the infrastructure providers concern about the security issues and costs of modifications. We target IaC based orchestration, hence as long as

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

a platform can be modelled with a standard modelling language and actuation engines, the support of the platform can be provided.

3.2.1 The standardization problem

There exist orchestrators that target a workflow execution over distributed hybrid infrastructure (combination of Cloud, HPC, etc.). Pegasus[12] and Kepler[13] are workflow management systems, portable across multiple infrastructures; however, they use specific (non-standard) formats to define a workflow, and the consequence is the lack of portability across different workflow management systems. Having a non-standardized format leads to the interoperability issues [49]; therefore, a standardized approach is required. With regards to the workflows standardization, the Common Workflow Language[14] is an open standard initiative, which has multiple implementations such as Arvados[15] and Toil[16]. The TOSCA standard defines declarative and imperative workflows and its workflow management has been evaluated as promising [50, 51]. We will evaluate TOSCA-based workflow management for the execution of Clinical Trial use case.

There is research done on the orchestrating application and resources of Cloud/Edge continuum in the projects such as DECENTER[17], Basmati [52], PrEstoCloud[18] and mF2C[19]. Most of them built the proprietary tools and services (e.g. FogAtlas[20]) based on existing open source orchestration systems such as Kubernetes. IaC support for deployment, configuration and meta-orchestration of Kubernetes clusters enables the SODALITE orchestrator to target edge computing environments, which will be evaluated in Vehicle IoT and GPU Snow use cases.

While some orchestration tools operate in a single vendor-specific infrastructure (e.g. Heat[21], used for orchestrating OpenStack private clouds, CloudFormation[22] for AWS public cloud, Kubernetes[23] for container-based clouds and Pegasus[24] Workflow Management System for HPC), there is a demand for the orchestration that manages multiple heterogeneous environments. One of the consequences is that infrastructure resources and access points must be transformed into IaC, as it is done with Terraform[25] for multi-cloud/multiservice provisioning. Moreover, in order to avoid a negative effect of vendor lock-in, such orchestration additionally needs to be standardized.

Although there exist standards to cope with portability and interoperability, such as TOSCA (Topology and Orchestration Specification for Cloud Applications), OCCI[26] (Open Cloud Computing Interface), CAMP[27] (Cloud Application Management for Platforms) and CIMI[28] (Cloud Infrastructure Management Interface), we focus on TOSCA standard due to enhanced portability across multiple cloud models (XaaS). Other examples of application modelling languages include Cloud Application Modelling and Execution Language (CAMEL) [53] that supports modeling and execution

[12] https://github.com/pegasus-isi/pegasus
[13] https://kepler-project.org
[14] https://www.commonwl.org
[15] https://github.com/arvados/arvados
[16] https://github.com/DataBiosphere/toil
[17] https://www.decenter-project.eu
[18] https://prestocloud-project.eu/
[19] https://www.mf2c-project.eu/
[20] https://fogatlas.fbk.eu/
[21] https://docs.openstack.org/heat/latest/
[22] https://aws.amazon.com/cloudformation/
[23] https://kubernetes.io/
[24] https://github.com/pegasus-isi/pegasus
[25] https://www.terraform.io/
[26] https://occi-wg.org/
[27] https://www.oasis-open.org/committees/camp/
[28] https://www.dmtf.org/standards/cmwg

of applications over multiple clouds and Cloud Modelling Framework (CloudMF) [54] that exploits cloud deployment configurations not only at design time but also at runtime for model-based reconfigurations of provisioned cloud services [55]. The concept of application modelling is detailed in D3.1[14].

Several research EU projects have explored the usage of TOSCA in the context of multi-cloud deployment, and subsequent adaptation and reconfiguration.

SeaClouds[29] provided solutions to enable seamless adaptive multi-cloud management of complex applications, by supporting deployment, monitoring and migration of application modules over multiple heterogeneous cloud offerings. SeaClouds adopted TOSCA for blueprint definition, and used Apache Brooklyn as orchestrator. Brooklyn uses a CAMP-like format for blueprints, so support for TOSCA was added. SeaClouds worked with two model layers, both in TOSCA: the Abstract Deployment Model (ADM) and the Deployable Application Model (DAM). In SeaClouds, users use the IDE to specify application components (web applications, application servers, databases…) with desired QoS, and needed resources, specified by their constraints (e.g.RAM, disk, maximum cost), generating an ADM as a result of this process. Then, the Planner takes the ADM, does a matchmaking of cloud offerings that satisfy the constraints and outputs a list of DAMs together with their estimated cost. After the user chooses a DAM, this is sent to the Deployer to start the deployment. After failing of QoS constraints, reconfiguration was first tried (i.e., acting on the same deployment, like restarting a component). If reconfiguration is not sufficient, a replanning is tried, which involves the Planner generating a new DAM and Deployer deploying the new deployment[57]. SeaClouds was focused on IaaS (e.g., AWS) and PaaS (e.g. Heroku), and therefore did not target neither HPC nor edge resources.

IBIDaaS[30] builds a unified framework that allows users to use and interact with Big Data technologies, facilitating cross-domain data-flow and increasing the speed of data analysis. IBIDaaS uses TOSCA as a blueprint for deployment on the two types of supported infrastructures: OpenStack private clouds and GPU commodity clusters. There is no tool for the creation of the TOSCA files: they are provided by the users of the framework. Moreover, they must be tailored to the Orchestrator actually used, which is based on Cloudify. At runtime, an adaptation engine can apply predefined elasticity rules in case the service conditions are not satisfied, but no more complex actions are enforced.

The **SWITCH** workbench [58] offers composition and modelling of time-critical cloud applications, their infrastructure planning and provisioning based on the hardware requirements, and self-adaptation based on QoS constraints. To achieve composition and modularity, Docker containerization is used. The applications are modelled in IDE and converted into the TOSCA blueprints, which then are further transformed into the Docker Compose file for the actual deployment in the cloud environment. SIDE targets the deployment of specific cloud applications, namely time-critical, and does not provide the connections to HPC/Edge platforms. The self-adaptation allows reconfiguration of underlying infrastructure resources, however, it does not support deployment updates, where new application components can be introduced [59].

In the field of HPC workload managers, where we can highlight PBS/Torque and SLURM, they provide a way of interaction, which consist of launching commands (e.g., qsub, sbatch) to send jobs to the HPC queue, that is not friendly from other software components to use. Moreover, the interoperability between workload managers is often omitted. There are initiatives working in this direction. Slurm REST API[31] is a REST API that allows clients to interact with Slurm, besides using

[29] http://www.seaclouds-project.eu/
[30] https://www.ibidaas.eu/
[31] https://slurm.schedmd.com/rest.html

the command line interface or the C API. Since it is fully integrated with Slurm (it is actually part of the Slurm product), it cannot be extended for other workload managers. On the other hand, QuantumHPC provides a set of node.JS libraries[32] for the management of Slurm and PBS/Torque, which could be used for implementing REST APIs for both workload managers. This represents a good starting point for a uniform REST interface supporting different workload managers, but the libraries are expected to be run as a particular user on the frontend node of the HPC cluster. Therefore, they do not provide multi-tenancy, hindering the ability to provide workload-manager-as-a-service. We address these aspects by introducing ALDE as one of the orchestration drivers, which is being enhanced to provide a multi-tenant uniform interface for managing jobs on both Slurm and Torque in an interoperable way.

3.2.2 TOSCA Orchestrators

Finally, there are several TOSCA orchestrators that are suitable for deployment on heterogeneous infrastructures, and as such, to be used in SODALITE.

OpenTOSCA [56] is a TOSCA Runtime Environment to deploy and manage Cloud applications. It enables the automated imperative or declarative provisioning of applications that are modeled using TOSCA XML 1.0. The provisioning is based on Build and Management Plans, which can be modelled using BPEL or BPMN workflow languages or generated by the Plan Generator. The OpenTOSCA runtime can also perform so-called situation-aware deployment, where the Plan Generator creates adaptation plans in order to change the deployment based on the given situation at runtime. The runtime focuses on the deployment on IaaS clouds (e.g. OpenStack, AWS), but does not support other targets such as Kubernetes, HPC clusters or edge resources.

INDIGO PaaS Orchestrator[33] allows instantiation of resources on the hybrid virtualized infrastructures (private, public clouds, virtual grid organizations) with the use of TOSCA YAML Simple Profile v1.0. It is integrated with other INDIGO services to enable best placement of the resources based on SLA and monitoring from the available list of cloud providers. In order to deploy, configure and update IaaS resources, the orchestrator uses an Infrastructure Manager (IM) [34] that interfaces with multiple cloud sites in a cloud-agnostic manner. Although the INDIGO PaaS orchestrator allows to spin up a virtual cluster (e.g. managed by batch systems such as PBS Torque/Slurm/Mesos) using TOSCA, the workflow management of the jobs is not directly supported and it assumes the usage of workflow management systems (e.g. Kepler) on top of deployed virtual infrastructure. Similarly, the partial reconfiguration is done on IaaS resources and it does not operate on the application.

Alien4Cloud[35] is a platform for application design, lifecycle management and deployment on cloud platforms. It provides a topology editor for design of the application, which translates the topology to Alien4Cloud DSL based on TOSCA blueprints. The underlying orchestrator (e.g. Cloudify or Yorc) then executes these blueprints that can be backed with custom implementation artifacts in Ansible/bash. Similarly to our orchestration system, Alien4Cloud additionally provides features such as versioning and reconfiguration; however, the later is only supported in the premium version.

Yorc[36] is an official orchestrator for Alien4Cloud - an open-source platform for UI and TOSCA based application design and lifecycle management over Cloud infrastructure. Yorc extends Alien4Cloud to support hybrid infrastructures, such as Clouds and HPC, with the support for OpenStack, AWS,

[32] https://github.com/quantumhpc
[33] https://github.com/indigo-dc/orchestrator
[34] https://github.com/grycap/im
[35] https://alien4cloud.github.io/
[36] https://github.com/ystia/yorc

Google Cloud, Kubernetes, Slurm and other platforms. Yorc natively implements normative types of TOSCA v1.2 with some types specific to the DSL of Alien4Cloud, as well as its own DSL specific to the supported platforms. The implementation of TOSCA interfaces can be done with Bash, Python or Ansible. Updating a deployment is a feature of a premium version of Yorc.

Croupier[37] is a Cloudify[38] plugin, which supports HPC infrastructure (based on Torque and Slurm) and executes batch jobs. **Cloudify**, in turn, orchestrates deployment and workflows of cloud applications with its own DSL based on TOSCA, supporting plugins for cloud systems, such as OpenStack, AWS, Kubernetes, and various node lifecycle implementations with Ansible, scripts and other utilities. Together with Cloudify, Croupier manages hybrid infrastructure (HPC + Clouds) via agent (Cloudify Manager) or agentless (e.g. via Ansible plugin) and allows to update a deployment by submitting a blueprint with the desired deployment state. Cloudify/Croupier uses its own DSL for deployment modelling, resulting in a vendor lock-in, which will negatively affect the portability and adoption of the components due to deviation from the TOSCA standard. The free (community) version of Cloudify has a limited feature set, lacking in authentication methods and multi-tenancy.

Puccini[39] not being an orchestrator, but rather a TOSCA parser and compiler, it allows to translate a rich set of various TOSCA-based Profiles, including TOSCA Simple Profile v1.0-v1.3, Cloudify, OpenStack HEAT, Kubernetes, into the *Clout* language - an intermediate format that an orchestrator needs to process to perform deployment. Currently, there are translators to Kubernetes and OpenStack cloud infrastructures, as well as BPMN and Ansible. **Turandot**[40] is an exemplary orchestrator based on Puccini and supports only Kubernetes.

xOpera is an open source lightweight orchestrator compliant with TOSCA Simple Profile v1.2 and in the process of being v1.3 compliant. It uses Ansible for the implementation and configuration of the node lifecycle and relationships, therefore managing the target nodes and infrastructure agentlessly. In the roadmap of xOpera development is the support of TOSCA workflows, policies, groups and deployment updates.

xOpera allows developers to create infrastructural code (IaC) working at the blueprint level (TOSCA) and without unnecessarily focusing on low level details that may require detailed knowledge of specific infrastructural languages and resources. These blueprints are processed by xOpera and executed through Ansible playbooks. In this way, applications can be deployed on multiple diverse computing platforms, clusters and supercomputers with homogeneous or heterogeneous node architectures, including resources available on the Cloud.

*Comparison and analysis***.**

While the considered orchestrators provide cross-system orchestration, not all of them can be integrated to the SODALITE platform due to lack of support for hybrid infrastructures, issues with adoption and standardization. Yorc provides a rich set of features and TOSCA compliance; however, we would like to pursue an open and free distribution of the SODALITE components. With the deployment updates being locked to the premium version, Yorc would hinder such a distribution. Croupier uses its own DSL for deployment modelling, resulting in a vendor lock-in, which will negatively affect the portability and adoption of the components due to deviation from the TOSCA standard. Therefore, being open source and lightweight, xOpera was chosen as the base orchestrator for SODALITE; however, it requires extensions according to its roadmap in order to build more sophisticated use cases.

[37] https://github.com/ari-apc-lab/croupier
[38] https://cloudify.co/
[39] https://github.com/tliron/puccini
[40] https://github.com/tliron/turandot

Therefore, the orchestrator system we develop will advance state-of-the-art with a standards compliant open-source solution that will have enterprise-like features, such as hybrid infrastructure support (Cloud/Edge/HPC), versioning, multitenancy and reconfiguration, filling the gap for less capable open source orchestrators in terms of functionalities and feature-set.

In principle, underlying orchestration tools can be interchangeable, as long as they are compliant with TOSCA. We will research more tools and contribute to the development of xOpera in the future versions of SODALITE Runtime Layer.

3.3 UML use cases

D2.1 "Requirements, KPIs, evaluation plan and architecture" introduces initial requirements for each layer of SODALITE architecture and derives the UML Use Cases. Within the Runtime Layer, the deployment and reconfiguration covers the following UML Use Cases:

- **UC6: Execute Provisioning, Deployment and Configuration**. As introduced before, it takes the IaC blueprint, provisions the resources and deploys the application.
- **UC7: Start application**. It is intended for applications that do not act as services, like HPC applications. These kinds of applications take an input, process the input and give a result; and the idea behind the use case is that they are deployed once, but can be executed several times.
- **UC10: Execute partial redeployment**. As part of the refactoring process, a new TOSCA file is inferred, specifying an alternative deployment for the application. The redeployment process takes the new specification and performs the necessary changes in the deployment.

The sequence diagram of the **Execution Provisioning, Deployment and Configuration** use case is shown in Figure 10, which describes the interaction of the SODALITE components with the Execution Platform Managers. We considered OpenStack as a provisioner of virtual resources, Kubernetes as an environment and orchestration for containerized application and Torque as a batch job system provisioning bare-metal CPU and GPU resources. The considered execution platforms provide a Lightweight Runtime Environment (LRE) to deploy application artifacts - OpenStack and Kubernetes employ a Docker runtime, whereas Singularity is used within the Torque environment. The Application Ops Expert initiates the deployment via the SODALITE IDE, after IaC blueprints and playbooks have been generated, and submits them to the *Orchestrator*. Upon this, the *Orchestrator* is triggered to provision resources via the Execution Platform Managers for the deployment and configuration of the application components.

Specifically, before the deployment of the application, OpenStack preliminarily provisions virtual resources such as virtual machines (VMs), networks and storage. As soon as the VMs are ready, the orchestrator installs the selected LRE and then deploys and configures the application. In Kubernetes, the deployment declaration, containing information about needed resources and application artifacts, is submitted to the Kubernetes API, and the resource provisioning and application deployment is autonomously executed. For Torque managing CPU and GPU clusters, the *Orchestrator* uploads the artifacts (or pulls LRE images from the registries) and the job description scripts specifying needed resources to the user workspace (e.g. home directories) located on the front-end nodes.

When the deployment of the heterogeneous application components is completed, the application is ready for its execution. Optionally, the *Orchestrator* configures the monitoring platform to initiate the collection of metrics.

As an example, the POLIMI Snow use case (see Section 4.1 of D6.2[17]) is composed of several components, to be deployed on the cloud or on HPC. The TOSCA file corresponding to the whole

application declares where each component must be deployed. The *Orchestrator* follows the procedure described above to deploy the application.

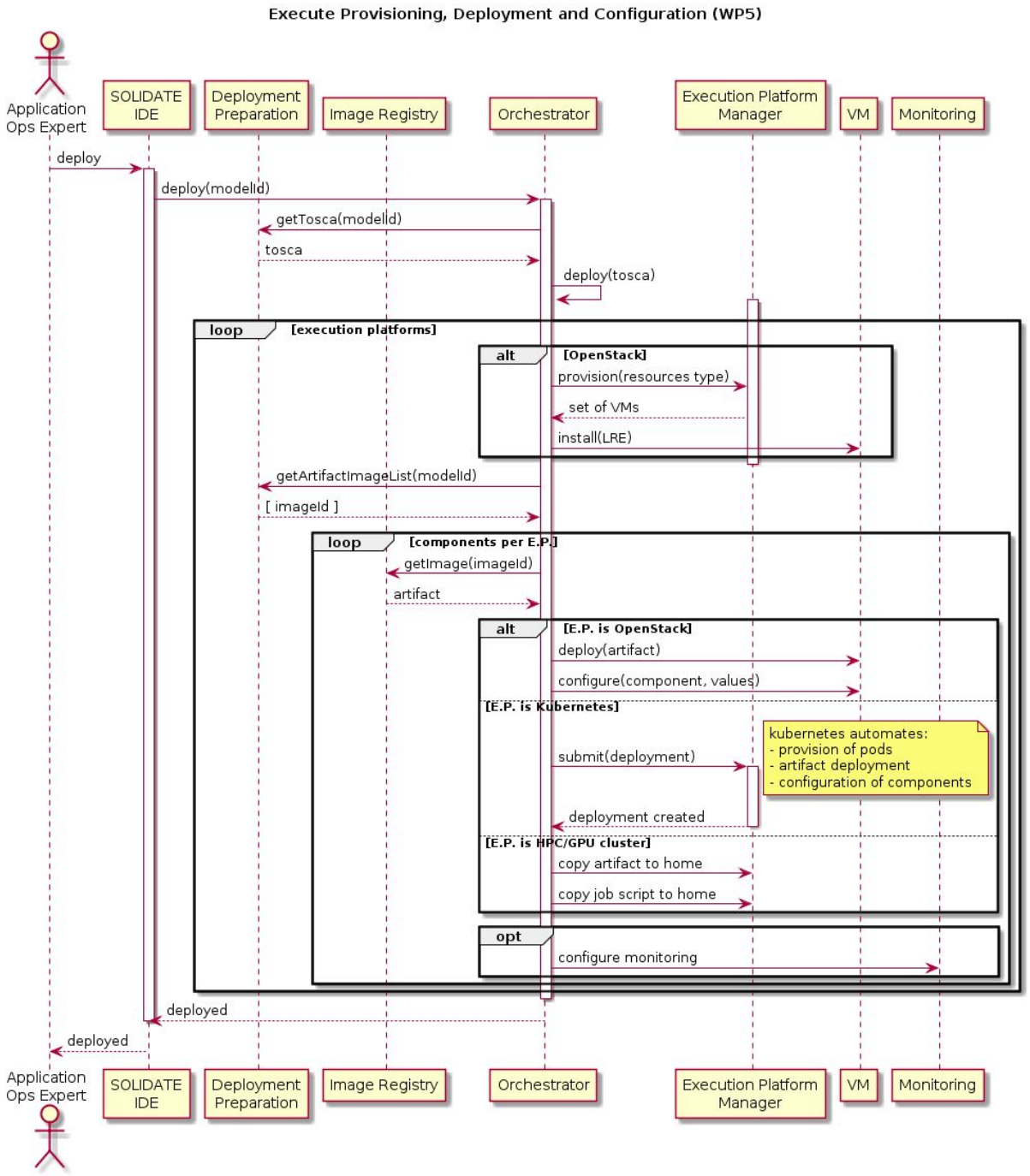

Figure 10 - UC6 sequence diagram

The sequence diagram of the **Start application** use case, shown in Figure 11, describes the steps performed by the *Orchestrator* to start the application. The application components containing services (e.g. web servers, REST APIs) are usually started after the deployment and run until they are terminated. The batch components (e.g. HPC jobs), which have a distinct beginning and end,

are started after the trigger from the *Orchestrator* and can be executed several times for a single deployment.

The Application Ops Expert uses SODALITE IDE to start an application. This action triggers the *Orchestrator* to start the execution of application components. Upon the component start, the execution environment notifies the *Monitoring Platform* about the component health and readiness to send monitoring metrics. The *Monitoring Platform* then registers the component, and the *Orchestrator* initiates the collection of the metrics for certain purposes (e.g. to check the application health).

The *Daily Median Aggregation* component of the Snow use case is executed daily taking as input a list of images obtained within a day by a webcam. In the context of SODALITE, the process described above is triggered by the Application Ops Expert or a REST call to the *Orchestrator* to start the execution. Since this component is intended to be deployed on HPC, the result is that it is submitted to the HPC workload manager to be executed.

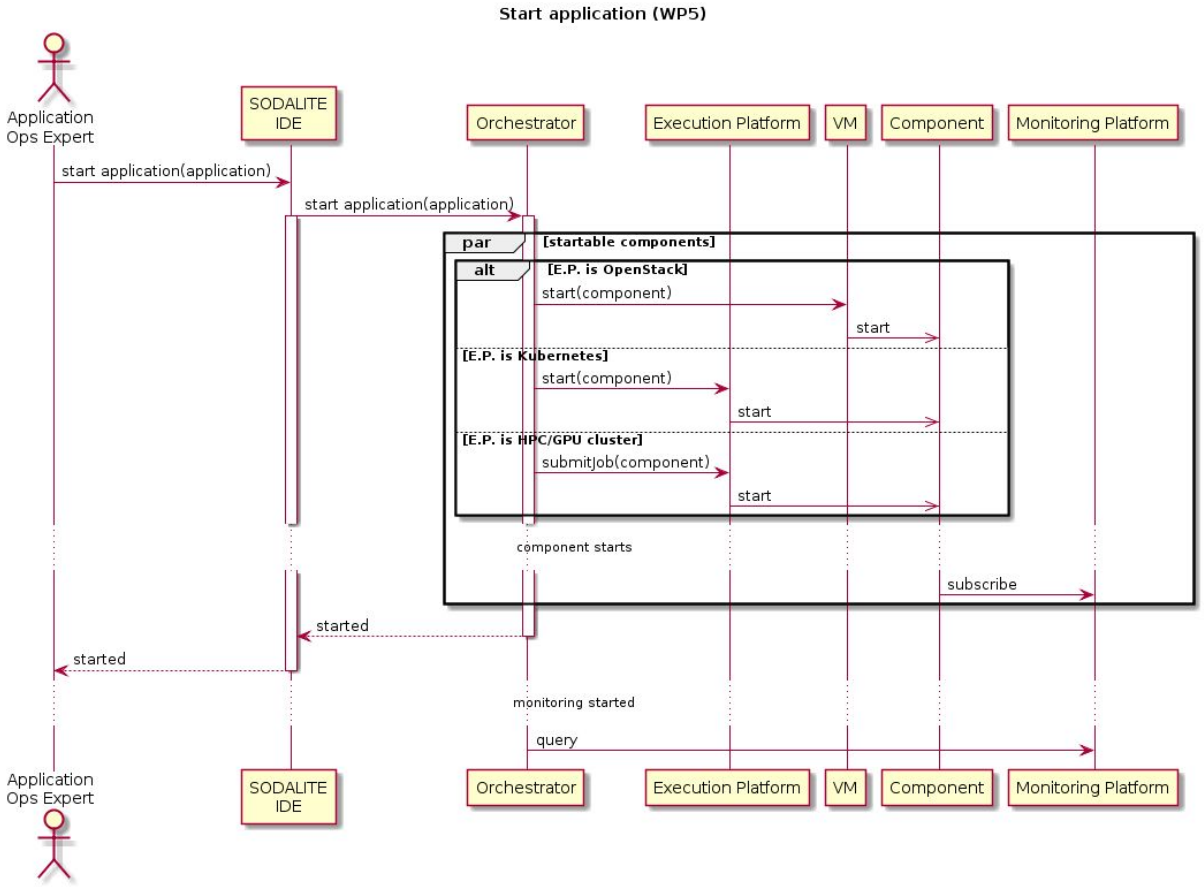

Figure 11 - UC7 sequence diagram

The *Refactoring* module collects the monitoring metrics and then decides whether the application needs a redeployment. If this is the case, the *Refactoring* options are identified and the Deployment Model is then updated, and **Execute partial redeployment** use case takes place as shown in Figure 12. The Application Ops Expert approves the redeployment via SODALITE IDE, which requests the *Deployment Preparation* module to prepare the IaC blueprints and playbooks of

the updated deployment and to submit them to the *Orchestrator*. The *Orchestrator* derives the difference between the current and updated deployments and applies adaptation actions on the Execution Platforms until the current state of deployment becomes the updated state.

For the Cloud based platforms, these adaptation actions include any form of scaling (e.g. scale up/down, in/out), migration to another Execution Platform, the deployment of new or removal of current application components. Due to inflexibility in scaling at runtime for the systems managing HPC and GPU clusters, the scaling adaptation actions are not included in the sequence diagram (Figure 12), however, the remaining actions applicable to the Cloud platforms (e.g. migration, adding new and removing old components) can be implemented.

In the POLIMI Snow use case, the *Skyline Extractor* component can be migrated from a VM without GPUs to a VM with GPUs in response to a low response time. If the refactoring process proposes the migration and the new deployment alternative is accepted by the Application Ops Expert, the *Orchestrator* applies the new deployment and performs the migration of the component to the VM specified in the new TOSCA file.

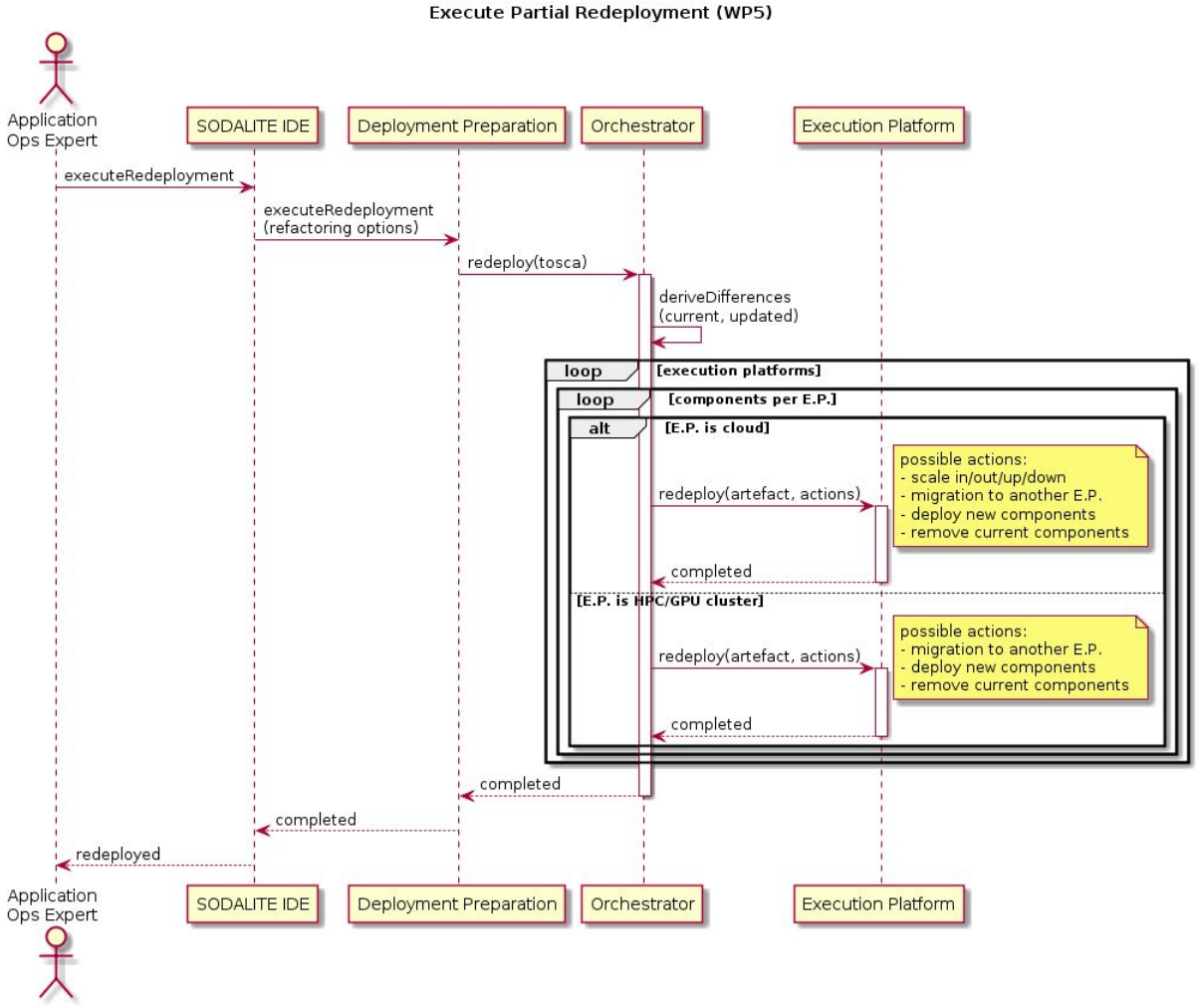

Figure 12 - UC10 sequence diagram

3.4 Architecture

The architecture of the *Orchestrator* is presented in Figure 13. The *Orchestrator* exposes a REST API for deployment management and accepts a CSAR (Cloud Service Archive), which contains IaC artifacts, TOSCA blueprints and Ansible playbooks, generated by the *Deployment Preparation* component. It should be noted that the IaC artifacts already contain the concrete resource definitions that an application component will be deployed on, and those resources were already preselected by other SODALITE components such as *Semantic Reasoner*[14] and *Application Optimizer*[15] during design time or *Predictive Deployment Refactorer* during runtime. It is the task of the *Orchestrator* to interface workload managers and execution platforms and execute the IaC to provision these resources.

When an application deployment is requested, the *Orchestrator* validates the supplied CSAR, registers it in the persistence mechanism and creates a version (tag) of the deployment. Similarly to software releases, the purpose of versioning is to have multiple deployment versions of a single application. Each version may include different execution platforms, optimization options and component updates. As a backend for versioning, the integration of Git is considered.

Internally, the *Orchestrator* core contains a TOSCA Parser, Deployment State Management and a placeholder for various Drivers and Actuations. The Parser operates on the version v1.2 of TOSCA Simple YAML Profile specification and is being developed to support the latest v1.3. The Deployment State Management is responsible for accounting the deployment state of TOSCA nodes and relationships and storing their design time (node properties) and runtime (node attributes) parameters, such that the deployment status can be monitored and the next deployment iterations can be compared against current version. The deployment versions comparison helps to efficiently update and reconfigure running application topology, handling only those nodes/relationships that need modifications.

In order to execute a declarative workflow and its lifecycle operations for nodes/relationships defined in a TOSCA, various implementation artifacts are needed. Currently, the *Orchestrator* supports only implementation artifacts based on Ansible, which is a part of Drivers and Actuations. It interfaces with various APIs and endpoints, e.g. *IaaS Management APIs*, *Platforms APIs* and *Batch System APIs* (refer to Section 3.1.1 for details) in order to request the resources needed for the deployment, configure and deploy the application components on top of them as described in UML use cases (Section 3.3).

The usage of containers as an application component enables computation interoperability between multiple computing hosts (Section 5). In SODALITE, optimized container images of applications are provided and can be optionally selected as part of static optimization [15]. In this case, the *Orchestrator* uses *ImageRegistryAPI* in order to pull these optimized images, otherwise, other image registries are used such as Docker Hub and Singularity Hub. Moreover, the *Orchestrator* might install monitoring agents and exporters, and configure the centralized monitoring server using *MonitoringAPI* to accept monitoring metrics (Section 4).

Advanced orchestration features, e.g. data management, platform abstractions for vendor-agnostic deployment and interoperability in HPC (e.g. via ALDE) and authentication/authorization, are not yet supported, and Ansible will be complemented with other services in the future releases of the *Orchestrator* to provide such support.

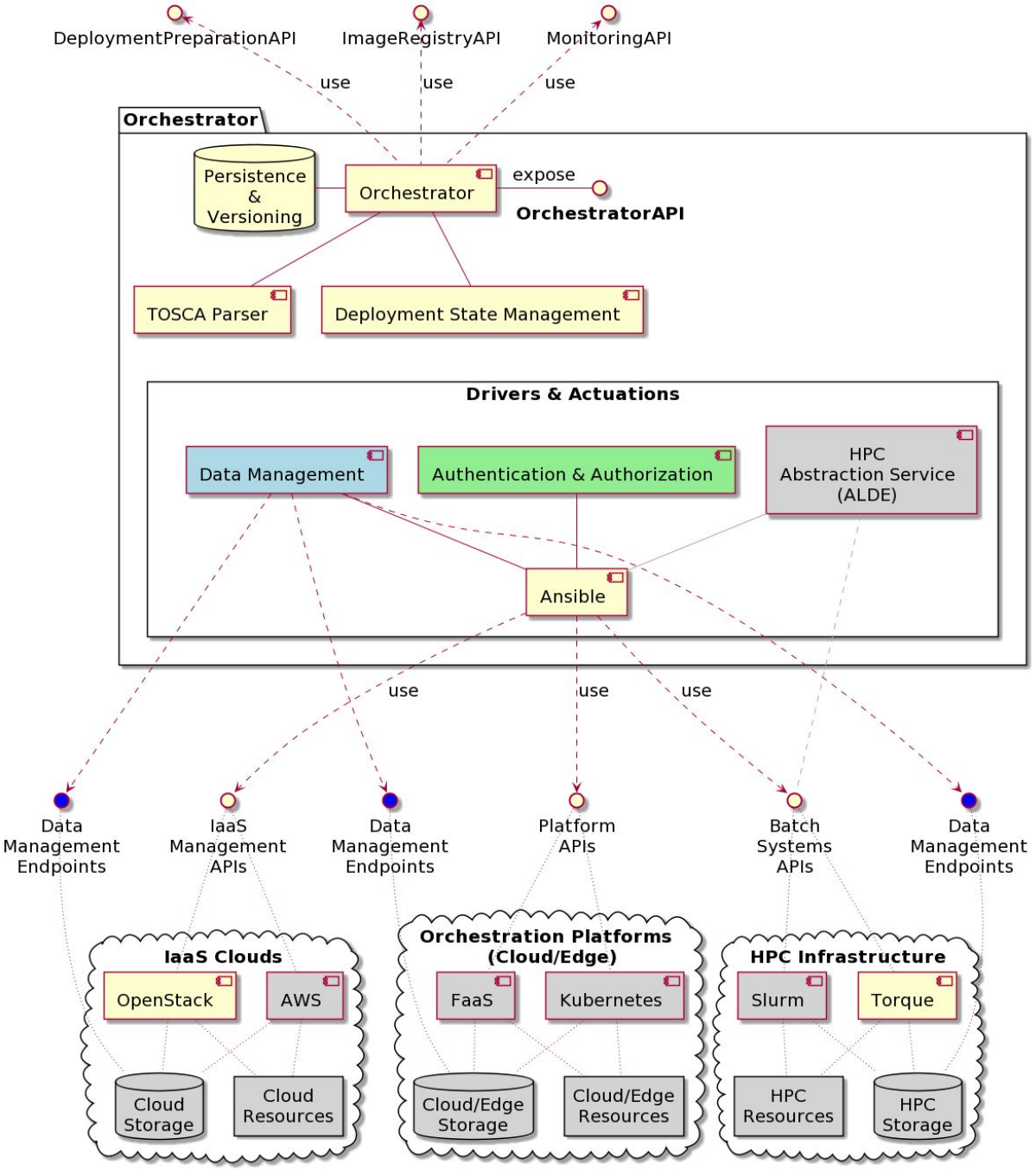

Figure 13 - Orchestrator Architecture

One of the main challenges for the *Orchestrator* is to incorporate various transfer protocols and endpoints to achieve transparent data management via IaC across multiple infrastructure providers. At this point, the *Orchestrator* can incorporate Ansible's built-in modules for data transfer, e.g. Files modules (copy, fetch) or URI modules (get_url, uri); however, these modules do not support any advanced features such as security, performance, third-party transfer, etc. Therefore, in the following year, advanced data transfer protocols such as GridFTP will be explored and their IaC support will be provided.

Another challenge is to perform authentication and authorization for the multiple infrastructures, platforms or resources access, as well as the access to the *Orchestrator* itself. While the latter can be performed at the level of the *Orchestrator* REST API, the access to the multiple infrastructures/platforms requires centralized credential and identity management. We consider Ansible Vault for secure secret management, and will implement the Identity and Access Management with the integration of tools such as Keycloak[41].

In the following subsections, we detail the components we are actively developing.

3.4.1 Orchestrator (xOpera)

xOpera is a lightweight orchestrator compliant with the TOSCA simple YAML Profile v1.2, while support for v1.3 is in progress. Within SODALITE we provided extensions for using xOpera with a REST API.

The capabilities of xOpera are:

- deploying and remove services based on predefined service templates based on TOSCA/Ansible blueprints;
- managing persistence of the deployed blueprints;
- handling status and information returned by the blueprint deployment process;
- managing session tokens for the deployed blueprints.

Software dependencies

- Python 3.6+
- Virtualenv
- OpenSSH client
- PostgresDB
- an operating virtualization environment (e.g. OpenStack)
- Ansible

Composed of

- CLI interface capable of deploying and undeploying of the TOSCA/Ansible blueprint on heterogeneous environment using SSH agentless;
- REST API is the orcherstrator's api call entrypoint, implementing:
  - handling of the deployment process;
  - persistence of the deployed blueprints and versioning;
  - handling of deployment SSH keys.

Roles that interact with the component (i.e. Application Ops Expert, Resource Expert)

- Application Ops Expert

Depends on

- Image Registry

Flexibility, resiliency and scalability aspects

The dockerized xOpera REST API deployment enables the operators to use xOpera in a multiuser and scalable way. The xOpera REST API already uses PostgreSQL as a deployment persistence database and can be configured for failover or in a load balancer setup.

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

The xOpera REST API handles deployments jobs asynchronously. Deployments and undeployments are executed as separate processes and registered in the database for session and status handling, enabling the user to reference a specific deployment process whenever needed. The user has full support for log tracing and status checking through unique session and blueprint tokens.

The dockerized REST API supports flexibility of deployment within Docker Swarm or Kubernetes cluster. This kind of setup provides resiliency and scalability for maximized throughput.

Repositories

https://github.com/SODALITE-EU/orchestrator
https://github.com/SODALITE-EU/xopera-rest-api

Demo

https://www.youtube.com/watch?v=cJUWFeOImOI

This video presents an AOE modelling an AADM on the SODALITE IDE, and how the application can be deployed from the IDE and monitored through a Grafana dashboard.

3.4.2 ALDE

ALDE[42] (Application Lifecycle Deployment Engine) is a REST API responsible for managing the workload scheduling and execution of applications, primarily intended for HPC environments. It is an outcome of the TANGO[43] project. The purpose of ALDE in SODALITE is to provide an abstraction layer between the *Orchestrator* and the underlying workload manager in HPC platforms - facilitating the adoption of new managers - and to simplify the interconnection from the *Orchestrator* to the workload manager thanks to the REST API.

The capabilities of ALDE are:

- compilation: when providing the source code of an application, ALDE can compile for the specific hardware architecture through the use of templates;
- packetization: similar to compilation, ALDE is able to build the Singularity container for the compiled code;
- deployment: finally, ALDE is able to manage the deployment and execution of the application onto the supported workload manager.

Software dependencies

- Python 3.6+
- Virtualenv
- SQLite 3+
- OpenSSH client
- Singularity 2.3.1+

Composed of

- REST API. It is the entry point to the functionalities offered by ALDE.
- ALDE Logic. The backend component, composed of:
    - ALDE Manager. It orchestrates the logic of the rest of ALDE components.

[42] https://github.com/TANGO-Project/alde
[43] http://tango-project.eu/

- Workload scheduler. It manages the scheduling of an application by submitting it to the testbed's workload manager.
- Application builder. It is responsible for compiling an application. The compilation depends on the target architecture.
- Application deployer. It is responsible for deploying an application to the chosen testbed.
- Application packetization. This subcomponent builds a package for an application. The supported formats are: Singularity, Docker, tar.gz or ISO image.

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

- There is no direct interaction of any SODALITE actor with this component.

Depends on

- Workload Manager (in the case of SODALITE, Torque)

Flexibility, resiliency and scalability aspects

In terms of flexibility, although ALDE initially only supported Slurm, it was conceived to be extended with more workload managers. This way, adding support for e.g. Torque implies small refactoring efforts. In addition, since ALDE has a high percentage in test code coverage, it is easier to refactor or add code without breaking existing code.

In case of scalability issues (i.e., a high number of requests/s), ALDE can leverage the infrastructure to achieve horizontal scalability. It must be taken into account that it depends on a relational database, which is by default, SQLite. As such, if ALDE needs to be put into production under a high-demand environment, a high-performance database, like MySQL or PostgreSQL, must be configured. ALDE utilizes SqlAlchemy as a database abstraction layer, and switching from one to another is, in principle, a matter of configuration.

On the other hand, ALDE handles connections to workload managers in an asynchronous way. When the REST API is invoked for a time-consuming task (e.g., generate a singularity container from a template), the task is marked to be done and a background process is responsible for performing the task. This makes ALDE to be very lightweight in terms of processing incoming requests, alleviating the need to scale-out when a peak of requests occur.

Repositories

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

3.5 Development status

3.5.1 Orchestrator (xOpera)

During the SODALITE project, xOpera has been enhanced with a number of features.

- A REST API has been added to make it possible to invoke it easier by other components.
- The deployment on HPC through SSH is now supported.
- It is now possible to store secrets (i.e., passwords or API keys) on separate YAML files, instead of on the blueprint itself. This functionality is also available when the REST API is utilized.

xOpera is able to perform basic reconfigurations (e.g., migration of an application to another VM) if a redeployment is triggered and the blueprint has changed. In SODALITE, the *Orchestrator* works

together with the *Refactoring* to provide this functionality, which will be available to SODALITE users when integrated into the SODALITE workflow.

All deployments should be possible to make through the SODALITE platform. For this reason, we prepared a blueprint for the deployment of xOpera itself, where the following components are deployed: (i) xOpera REST API, (ii) the database for internal persistence of deployed blueprints, and (iii) the TLS secure private Docker Image Registry to be used for the deployment of the application artifacts deployed on OpenStack testbed. This blueprint is deployed successfully.

3.5.2 ALDE

Being an outcome of the TANGO project, it is a component ready to be run in a TANGO environment. As such, ALDE is being adapted to be used inside SODALITE. The work on ALDE has just started, so no tangible progress has been developed. The first objective in Y2 regarding ALDE is the support of Torque and the integration within the SODALITE framework.

3.6 Next steps

Currently, OpenStack, Kubernetes, OpenFaaS and Torque are considered as deployment targets. Support of other targets, such as Slurm, Docker Swarm, public cloud (e.g. AWS) and OpenWhisk, is foreseen as we progress to the next stages of the project, depending on the demand from the use cases. The same applies to the support of different actuations, e.g. based on scripting (e.g. Bash, Python) or Chef/Puppet. The support for the TOSCA workflows is currently being implemented as well. With tools such as ALDE, a REST API can be introduced to the HPC clusters in order to unify the access to the different platforms and to avoid dependencies on different HPC workload managers and their commands, e.g. Slurm with ***sbatch*** and Torque with ***qsub*** commands.

For xOpera the following developments are already underway or planned to be done in Y2:

- fully progress from TOSCA YAML v1.2 to v1.3 to support the latest standard version with enhanced TOSCA features (e.g. TOSCA policies);
- integration of credential management and identity and access management in order to provide authentication/security features (e.g. secret passing) into the deployment phase;
- using parallelisation for deployment efficiency across multiple execution platforms.

With regards to ALDE, the following actions are to be started in Y2.

- Torque support. ALDE currently only supports Slurm, so it must be extended to support the Workload Manager used in our HPC testbed. This will make it possible to validate the idea of ALDE to use one interface for connecting to different workload managers.
- Security layer. An authentication system must be put in place so the REST API calls the underlying workload manager on behalf of the user. The current situation is that ALDE uses per-instance SSH credentials to connect to the host and execute the workload manager commands. Adding this security layer will make ALDE a multi-tenant system.

Although TOSCA already provides a certain degree of interoperability with multiple infrastructures, we are exploring the connection with other projects and initiatives (e.g. EGI[44]) that are working with similar topics as SODALITE. In particular, the meta-orchestration of other tools/services (e.g. IM, libcloud[45]) and open standards (e.g. OCCI) that are specifically addressing interoperability would be explored.

[44] https://www.egi.eu/
[45] https://libcloud.apache.org/

4 Monitoring, tracing and alerting

The monitoring, tracing and alerting task is in charge of collecting all required metrics both for feeding them to other components that might require to know the execution state and for keeping operators informed of the system state and critical changes or events. In the SODALITE workflow, the components that utilizes the metrics gathered by *Monitoring* are the *Refactoring* and the *Optimizer*[15], which need performance metrics (job execution times and throughputs) and cost metrics (resource usage, energy, I/O, network...). In general, SODALITE needs to monitor the following components:

- cloud infrastructure, in order to observe its utilization (CPU, IO, network) and power consumption;
- cloud VMs, to observe how the applications are using their assigned VMs;
- HPC infrastructure, to be able to plan and launch jobs accordingly to the cluster status;
- application execution, to check the throughput and phase.

In this section, we first describe some general concepts about monitoring, followed by the related work, which highlights our research contributions. Next, we present the use cases and architecture of our monitoring component, followed by a description of each major component. Finally, we present the current development status and the deployment plan for each component.

4.1 Background

In the general sense of the word, monitoring means to watch and control a situation and expect to find something about it, but narrowing this definition to the computing field, monitoring is the process of tracking the performance of a system or application. In cloud, the monitoring part is an intrinsic one, as it is required for a diverse set of essential parts, such as resource planning and management, billing, troubleshooting, performance and security management, efficiency and effectiveness analysis, etc.

The resource monitoring can be classified into two categories: high-level and low-level resource monitoring, being the first one the monitoring of the state of the virtual machines, queues and deployment status, and the last one the monitoring of the physical resources, such as power consumption, network status and computing power utilization. These two levels need to be monitored to assure effective platform utilization and application execution.

The monitoring action normally involves a set of more specific tasks, such as tracing and alerting. Tracing is the practice of logging past values in a database in order to possess a historic representation or evolution of a set of desired metrics, and alerting is the procedure where certain changes are automatically notified to operators to bring their attention over them. This set of procedures, among others, such as the visual representation of the system state, congregate to form what is generically known as monitoring.

Before introducing a set of functionalities that a monitoring solution needs to cover, it is important to introduce the set of desirable characteristics of a good monitoring tool and how they will affect its selection[68].

- **Scalability**. In our project, especially in the cloud component, large deployments of several VMs require a monitoring solution capable of extending its range and give service to all of these new VMs and applications in a timely and flexible manner.
- **Portability**. SODALITE aspires to serve a heterogeneous platform, thus requiring that all its tools, including monitoring, to be heterogeneous and capable of operating in different infrastructures.

- **Non-intrusiveness**. Stealing computing power from the executing applications should be a concern and thus, minimizing its footprint is required.
- **Robustness**. In a continuously changing environment, with constant deployments and reconfigurations, a robust and well tested tool is required that can withstand those changes.
- **Multi-tenancy**. The nature of this project requires a multi-tenant environment, needing concurrency and information, allowing tenants to only access the monitoring information of its application.
- **Interoperability**. This concept is related to the portability one, as one involves the other. The monitoring solution will need to be able to interact with many different components, hosted in different architectures and with different backbones.
- **Customizability**. Adapting the monitoring solution to the necessities of the platform will be one of the fundamental parts of the development of this component and thus requiring a great degree of customizability.
- **Extensibility**. New tools will need to be developed to monitor specific parts of the platform (specially HPC), so a solution that eases the development of new components and its integration is mandatory.
- **Usability**. Ease of deployment, maintenance and interaction are basic requirements in general for all software tools and will be highly rated for the monitoring tool used in our project.
- **Affordability**. Aiming to increase performance and reduce cost for the final user with the execution of applications it is mandatory to aim to get the best cost to performance ratio in all components, including monitoring.
- **Archivability**. Having an historic trace of the monitored metrics can be useful for analysing problems' causes or improving the platform, as well as for accountability.

With these characteristics in mind, we needed to have a list of high level functions that a monitoring solution needs to include. In a recent survey[46] on "Best Network Monitoring Software & Tools of 2020" the basic functionalities of a network monitoring tool should be to:

- "scan and detect networked devices with or without agents;
- create a performance metrics baseline or threshold;
- continuously monitor the performance of the entire network;
- send alerts in case the network goes down or if a metric drops below;
- recommend proactive solutions for known issues;
- create visualizations and reports of the performance data".

This previous list is focused towards network monitoring, but it can be extrapolated to the whole SODALITE monitoring component. In general, this part will need to cover the following points that:

- allow you to manage and monitor the full stack, including HPC infrastructure and workload managers, and Cloud infrastructure and virtual machines;
- give full visibility through a unified dashboard and access point;
- the solution should be flexible, scalable, extensible and compatible with other tools;
- allow for Application Performance Monitoring (APM).

One of the utilities of the monitoring is to provide information to improve how applications are deployed and executed in the system. In an agent-based monitoring, this is done by injecting

[46] https://www.webservertalk.com/network-monitoring-software/

monitoring software in both physical and virtual hosts that exposes certain statistics. These statistics are later gathered by a central monitoring solution and distributed to other components. In addition, this information can be logged and displayed in a dashboard. This, together with the alerting system that notifies of critical changes or situations, assists the operators to have a good overview of the state of the system.

Synthesizing, the monitoring and tracing parts are an essential component in the application and deployment loop, as their task is to provide software and system performance insights that other units and operators can use to decide whether to increase, reduce or change the resources assigned to those tasks. On top of that, the alerting system helps to spot, mitigate and prevent problems.

4.2 Related work

There is a large variety of tools available relating to monitoring and tracing. Many such tools are commercially available as well as some good open-source tools. In a cloud environment, where components of an application are connected to one another via network communication, the network emerges as one of the most critical resources that impact application performance.

4.2.1 Monitoring surveys

Many surveys of popular monitoring tools are available. The recommendations of these surveys vary, but several products repeatedly show up in multiple surveys. We list here several surveys of monitoring tools[47].

On the list of "Best Network Monitoring Software & Tools of 2020" are: Solarwinds NPM, Solarwinds SAM, ManageEngine OpManager, PRTG Network Monitor, Nagios XI, Zabbix, NetCrunch, LogicMonitor, Icinga, Spiceworks Network Monitor, Datadog, WhatsUp Gold, ConnectWise Automate, OP5 Monitor, Pandora FMS, Splunk, Monitis, Dynatrace, Thousand Eyes, EventSentry.

In a recent Gartner survey[48] on "IT Infrastructure Monitoring Tools Market", the following monitoring tools received the highest scores: Zabbix, Datadog, VMware vRealize Operations, SCOM by Microsoft, OpManager, Nagios XI, Solarwinds Server and Application Monitor, PRTG Network Monitor, Micro Focus SiteScope, WhatsUp Gold, OnCommand Insight, ManageEngine, New Relic Infrastructure, ScienceLogic SL1 Netreo, IBM Tivoli Monitoring, Sysdig Monitor, eG Enterprise, LogigMonitor.

In a recent survey by G2[49] on "Best Cloud Infrastructure Monitoring Software", the following best Cloud infrastructure monitoring tools were identified: Dynatrace, Datadog, Sumo Logic, LogicMonitor, insightVM, Solarwinds Virtualization Manager, PRTG Network Monitor, Google Stackdiver, Catchpoint, SteelCentral, vRealize Operations, Splunk Insights for Infrastructure, ThousandEyes, MongoDB Cloud Manager, Zabbix, Checkmk, SolarWinds, Zenos, SignalFx, Azure Monitor, Micro Focus Operations Bridge, Blue Matador, Wavefront, New Relic Insights, Opsview Monitor, Pandora FMS, ScienceLogic SL1 Platform, CloudMonix, Netreo.

In a survey from 2019[50], the following monitoring tools are mentioned: Accedian, AppNeta, Cacti, Corvil, Datadog, Dynatrace, ExtraHop, Flowmon, Icinga, InfoVista, Kentik, LiveAction, LogicMonitor, LogRhythm, ManageEngine, Monitis, Nagios, NETSCOUT, Opmantek, Paessler, Plixer, Prometheus,

47 https://www.researchgate.net/profile/Vincent_Emeakaroha/publication/263774524_A_survey_of_Cloud_monitoring_tools_Taxonomy_capabilities_and_objectives/links/5cc96acc4585156cd7be2f3d/A-survey-of-Cloud-monitoring-tools-Taxonomy-capabilities-and-objectives.pdf

48 https://www.gartner.com/reviews/market/it-infrastructure-monitoring-tools

49 https://www.g2.com/categories/cloud-infrastructure-monitoring

50 https://solutionsreview.com/network-monitoring/the-32-best-network-monitoring-tools-to-use-in-2019/

Riverbed, SevOne, SolarWinds, Spiceworks, Statseeker, ThousandEyes, VIAVI, WhatsUp Gold, Wireshark, Zabbix.

4.2.2 Open-source monitoring solutions

Many of the above-mentioned tools are commercial. There are several open-source alternatives such as Prometheus[51], Zabbix[52], Nagios[53] or InfluxData[54] that are suitable to be integrated as the base of the SODALITE monitoring. The following is a description and comparison of these monitoring platforms.

Nagios. Nagios is an open source monitoring tool that is widely used and written in C. It provides a complete package that includes UI, a complete set of monitoring tools and the possibility of developing new ones for additional metrics. There are also new monitoring projects that are based on Nagios, such as Sensu. Despite being open source, the professional, fully featured Nagios XI is not free, complicating automated deployments, monitoring and maintenance of distinct projects.

InfluxDB. InfluxDB is another open source core tool that sells licenses for its more advanced features. It is not advertised as a monitoring system but as a time series database. Despite this, it has a similar functioning to Prometheus, as they share a comparable architecture and way of dealing with the data. Nevertheless, they are aimed for slightly different uses, as InfluxDB is oriented towards horizontal, long term data storage scaling and event logging.

Zabbix. Zabbix is a slightly older, widely used, monitoring tool written in C that has a native web interface for managing and monitoring. It is originally designed with the pull model in mind, but it also supports push mode. It can use a variety of relational databases for storage. Despite it being free and open source, it is limited partly by its core definitions, as it is made to monitor machines instead of services.

Prometheus. Prometheus is a trending monitoring solution that is built around a multidimensional data model that stores time series data identified by a metric name and value pairs that can be accessed via its own flexible querying language. The data is achieved via active scraping, with the typical setup divided between exporters and a central monitoring server. The different exporters have the goal of picking the desired metrics and thus are situated in the end points. Each exporter instance has a specific target. On the other hand, the Prometheus server has the goal of gathering the information from all different exporters and it can do it via pull or push methods, expanding its flexibility. This server is where the storage solution is, not having the option to have distributed storage. On top of that, it also has the option of having an alerting system that can trigger certain actions based on user defined events and, when integrated with Grafana, a fairly complete GUI. Lastly, and thanks to the simplicity of its structure, it is straightforward to develop new exporters for dedicated tasks, adding the option of exposing applications' internal state.

On top of checking all the previously stated requirements, being scalable, flexible and capable of monitoring the whole SODALITE stack, our baseline must be actively developed and widely used. Several recent EU projects have selected Prometheus. One example is the MS04SC project, as it had an infrastructure and set of goals for the monitoring component similar to the ones of this work, needing to extract information from HPC and cloud infrastructures and do so in a sensible and efficient way, choosing Prometheus as its central monitoring solution. Another EU project that

[51] https://prometheus.io/
[52] https://www.zabbix.com/
[53] https://www.nagios.org/
[54] https://www.influxdata.com/

adopted the Prometheus solution was SONATA, using it to monitor virtualized infrastructures for 5G services.

The final decision of choosing Prometheus came after considering its set of advantages. First of all, and contrary to other solutions like InfluxData, Prometheus is originally based on a Pull model, where the central monitoring solution periodically retrieves the metric values from their agents. This contrasts with other solutions that use the Push model, but one benefit that Prometheus has is its flexibility, as it also allows to use the Push model as well, allowing the final user to choose. In addition, Prometheus offers an interoperable solution capable of collecting metrics from distributed and heterogeneous systems that is able to scale well for bigger requirements, as it can from smaller clusters of monitoring services and agents that communicate between each other. Finally, and thanks to its out-of-the-box compatibility with Grafana that covers the graphical interface capabilities that other solutions include, Prometheus was selected based on all the above-listed reasons and its wide use, with companies like AWS and Soundcloud, and EU projects with similar objectives to SODALITE, using it.

With regards to the functionality offered to the rest of the platform, typically an event is generated when a problem is perceived and an administrator then has to investigate what is the source of the problem. There are rule-based systems to perform some standard operations, but mostly it is to raise an event. The administrator then initiates a redeployment. SODALITE is aiming high to do more of this redeployment automatically based on monitoring data, in conjunction with the *Deployment Refactorer*.

4.3 UML use cases

The Monitoring Task covers the UC8 UML use case “Monitor Runtime”.

- **UC8: Monitor Runtime.** Online runtime application behaviour monitoring, gathered from the target delivery platform. Required to create and update patterns describing the behavior of the underlying infrastructure. Additionally, deployment status information and online reconfiguration must be provided to end-users in the dashboard (IDE).

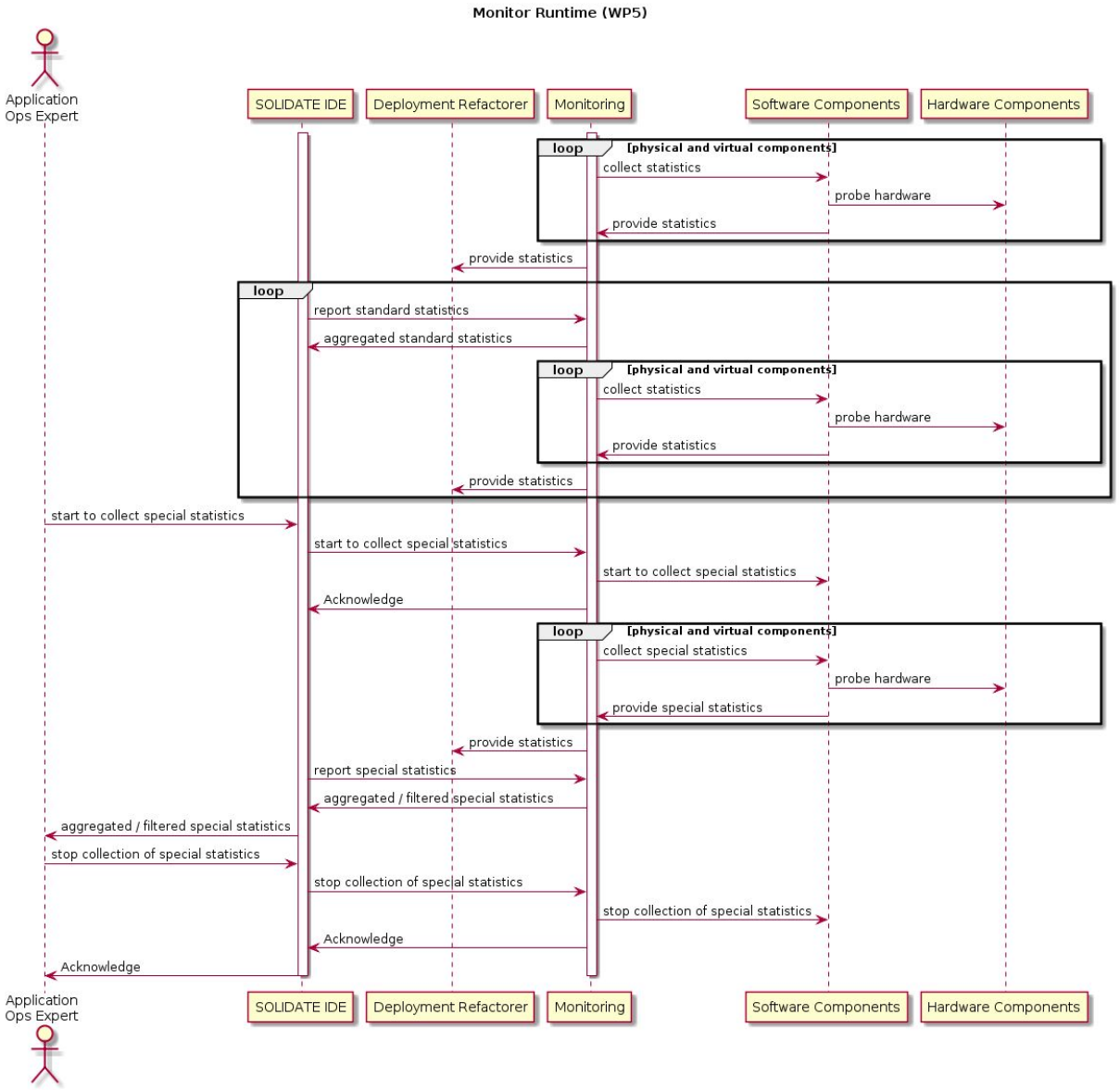

Figure 14 - UC8 sequence diagram

The sequence diagram of the **Monitor Runtime** UC is shown in Figure 14. The *Monitoring* component collects system statistics on an ongoing basis. On each host (whether physical or virtual) there is a software component that interacts with the *Monitoring* component and reports standard system statistics. The statistics are usually collected on each (physical or virtual) host by reading various counters and registers that hold updated system statistics. These combined statistics are collected and periodically reported to a dashboard that is part of the SODALITE IDE. These statistics are also available to be used by the *Deployment Refactorer* component to make placement decisions based on resource usage. In some cases it may be desirable to collect some specific non-standard statistics in order to isolate the cause of some observed anomaly. In this case, the operators can request to collect additional specific statistics. This request is translated by the *Monitoring* component into requests to the agents running on each (physical or virtual) host. When the operators no longer need the collection of the non-standard statistics, they inform the *Monitoring* component to stop the collection of those statistics[5].

4.4 Architecture

The SODALITE *Monitoring* block uses and integrates several Open Source projects, like Prometheus, Grafana, Skydive and some Prometheus exporters. The whole architecture is based on Prometheus, which acts as the central monitoring server. For retrieving monitoring data from the different targets (HPC, VMs, Kubernetes, applications), a set of Prometheus exporters are being integrated or implemented.

The general architecture of the SODALITE *Monitoring* is shown on Figure 15. The Monitoring server (Prometheus) has been set up in pull mode, and connects to the configured set of exporters, which are usually installed at the monitoring targets. For simplicity, the Exporter component in the diagram represents all the possible configured exporters, which obtain metric data from the different monitoring targets: physical nodes, VMs, Kubernetes, HPC, application… For example, a Node Exporter is installed on each VM that has been started on the OpenStack cloud, obtaining metrics about the VM.

Finally, operators can obtain monitoring reports through the IDE, based on Grafana.

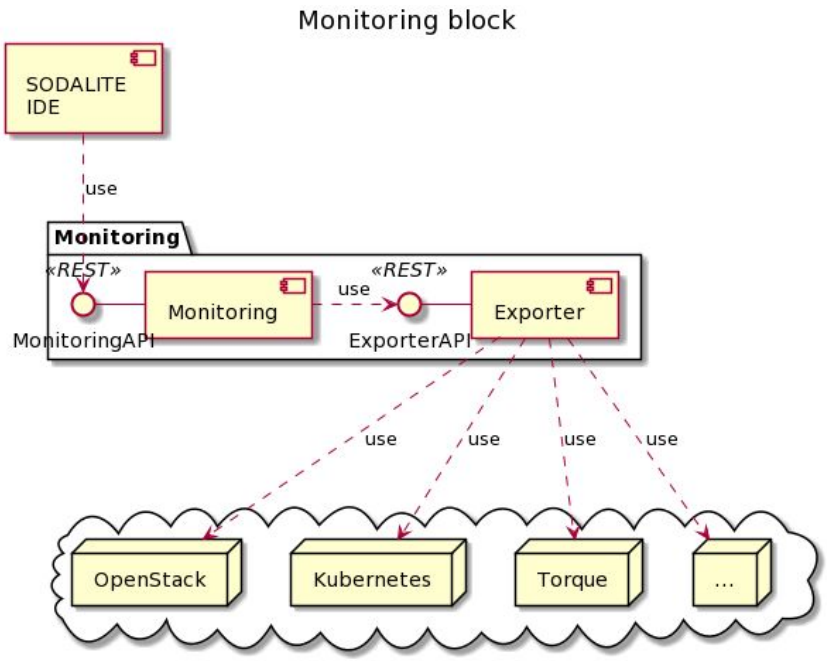

Figure 15 - Monitoring Architecture

In the following subsections we describe those components that we are actively developing.

4.4.1 IPMI Exporter

Simple yet versatile Prometheus exporter created to expose the power measurement given by an IPMI command that can be adapted to display any terminal output.

Software dependencies

- Go (built and tested with version 1.13)
- IPMI

Composed of

- Collector. Executes the desired command and catches the output, isolating the required metric and storing it to a variable.
- Rest API. Exposes the metrics and allows Prometheus server to collect them.

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

- There is no direct interaction of any SODALITE actor with this component.

Depends on

- Prometheus

Flexibility, resiliency and scalability aspects

The IPMI exporter is designed to expose the power consumption metrics of the machine it is running in. This is achieved at the moment by exporting the measurement given by a physical sensor and obtained via “ipmi_sensor” command. This step might change depending on the machine, as currently there are multiple ways of obtaining power consumption. In case that the environment has other commands for this purpose, the required changes to make it work will be small, being able to implement and test it in a matter of minutes. This gives great flexibility to this exporter to monitor different machines with low effort.

In terms of scalability, the nature of the Prometheus system makes this exporter easy to deploy and function in additional machines if required, as it can be easily integrated in a CI/CD system such as the current one in use in this project, Jenkins. On the other hand, and thanks to its simplicity and modularity, the resiliency is high and, in case of failure, it could trigger an alarm to attract attention and get fixed quickly.

Repositories

https://github.com/SODALITE-EU/ipmi-exporter

Demo

https://www.youtube.com/watch?v=cJUWFeOImOI

This video presents an AOE modelling an AADM on the SODALITE IDE, and how the application can be deployed from the IDE and monitored through a Grafana dashboard.

4.4.2 Skydive Exporter

The Skydive Flow Exporter filters the flow data obtained from Skydive, performs a data transformation, and saves the information to some target (e.g. an object store). This data may then be used to perform various kinds of analyses for security, accounting, or other purposes.

Software dependencies

- Go (built and tested with version 1.11.13)
- Skydive Analyzer

Composed of

- websocket connection to Skydive Analyzer to obtain relevant network flow information;
- a multi-phase pipeline of operations performed on the data: classify, filter, transform, encode, compress, store.

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

- There is no direct interaction of any SODALITE actor with this component.

Depends on

- Prometheus
- Skydive Analyzer

Flexibility, resiliency and scalability aspects

Skydive is a distributed application and is designed to be scalable to large clusters. Skydive agents run on each host to collect the necessary monitoring data. The monitoring data is then channelled to one or more Skydive analyzers. Data may be shared among the analyzers to ensure high availability.

Many different types of infrastructure can be monitored by Skydive, including bare-metal hosts, VMs, Docker or runC containers, Kubernetes entities (cluster, configmap, container, deployment, job, namespace, node, pod, service, etc). For each of these, various types of network interfaces and network flows are identified and can be monitored.

Repositories

https://github.com/SODALITE-EU/monitoring-system

4.5 Development status

SODALITE is a project with its main focus set towards making applications run both in HPC and cloud environments, abstracting that selection from the developer. In order to be able to make those application deployments, a fundamental part is the monitoring of those environments, as it is required to know how the applications are running and how effectively are they utilizing the environments they have been deployed in and if that deployment strategy requires modification to match the performance guidelines.

At the moment of submission of this deliverable, the development has been focused on the components described in the following subsections, whose targets were centered around cloud VMs and infrastructure monitoring, as we established that it was more accessible and stable at this moment, easing the initial set up, testing and development. The initial set up, described in more detail below, consisted in launching the Prometheus server, configuring it to discover all new monitorable machines; configuring new machines with Node Exporter, developing the IPMI exporter for monitoring power consumption in cloud infrastructure and lastly, integrating IBM's network monitoring tool into this system.

A summary of the development status is shown on Figure 16.

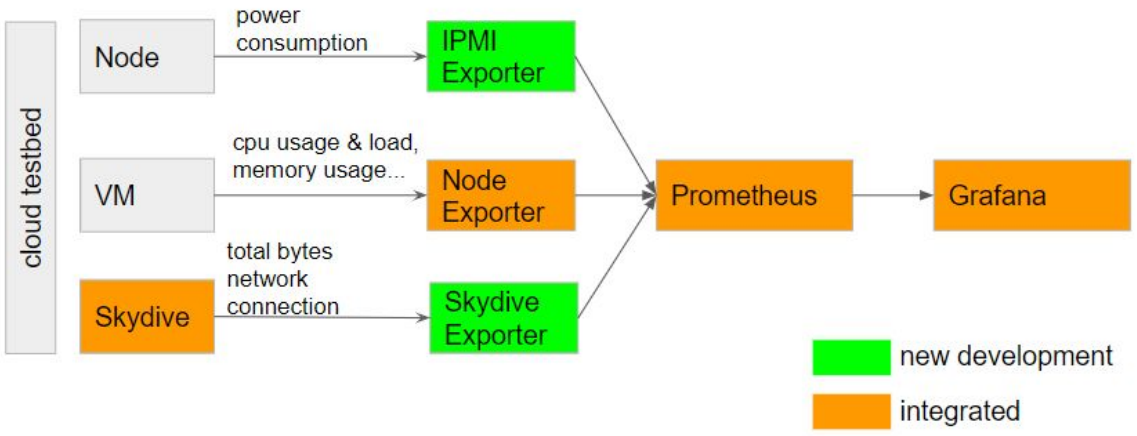

Figure 16 - Monitoring status

4.5.1 Prometheus

In our project, Prometheus is deployed as the main monitoring solution. It is running in a separate VM inside OpenStack, and its config files are the ones hosted in this GitHub repo (https://github.com/SODALITE-EU/monitoring-system). It automatically monitors new OpenStack instances, on top of tracking the power consumption of the physical nodes and metrics exposed by Skydive exporter.

4.5.2 Node exporter

The Node Exporter is open source software, part of the same Prometheus project. It is frequently updated and fairly stable as it is the most installed and used Prometheus Exporter. It offers CPU, memory, network and OS metrics (see Figure 17) and it is automatically deployed with every new virtual machine created. In addition, the central Prometheus instance automatically adds each newly created Node Exporter instance to the monitoring list. It is written in Go and its repository, including all the metrics it can collect can be found in the following link: https://github.com/prometheus/node_exporter

4.5.3 IPMI exporter

This exporter has been developed in SODALITE to expose the power consumption measurement given as an IPMI terminal command. This simplicity means that it can be adapted to expose any metric given as a CLI output. It is currently working on the physical nodes and providing their power consumption to the central Prometheus server. It is written in Go and its repository can be found through the following link: https://github.com/SODALITE-EU/ipmi-exporter

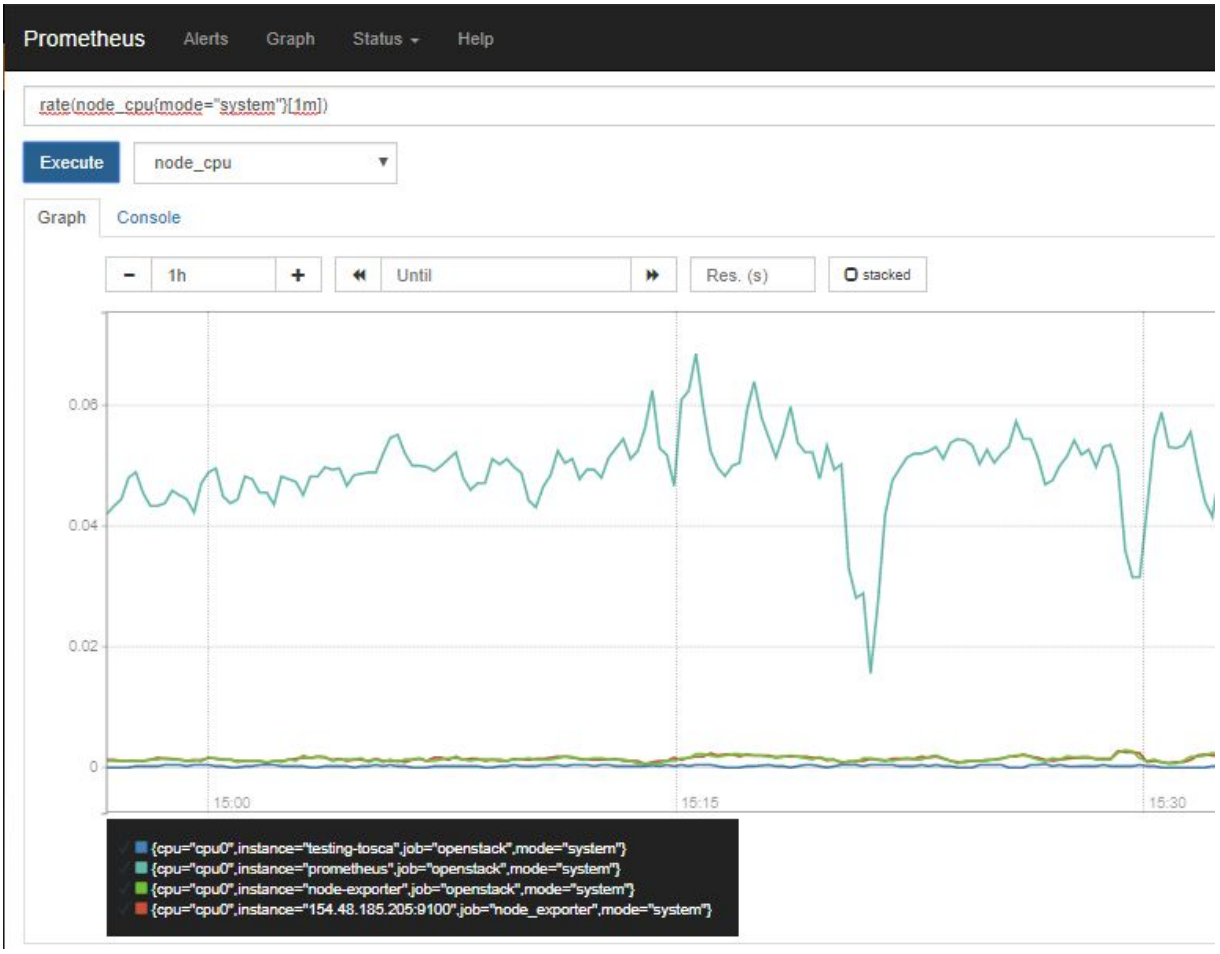

Figure 17 - Example graph showing the CPU consumption of several VMs

4.5.4 Skydive

Skydive is a stable Open Source project with community support. New features are added in an on-going basis. IBM team members are regular contributors to the Skydive community. Over the past year, we implemented a generic flow exporter for Skydive to expose the Skydive flow data in a convenient form to various consumers. This flow exporter has now officially become a project within the Skydive community (https://github.com/skydive-project/skydive-flow-exporter).

In SODALITE, we use Skydive to collect network information, and combine that information with other resource usage gathered from other sources. The data collected from Skydive is collected and forwarded to Prometheus, the monitoring tool for SODALITE. To this end, we implemented a connector to translate data from Skydive flows into a format that can be consumed by Prometheus. The first implementation provides the byte transfer counts for each network connection under observation in the testbed. Figure 18 shows an example of the data produced by the skydive-prometheus connector.

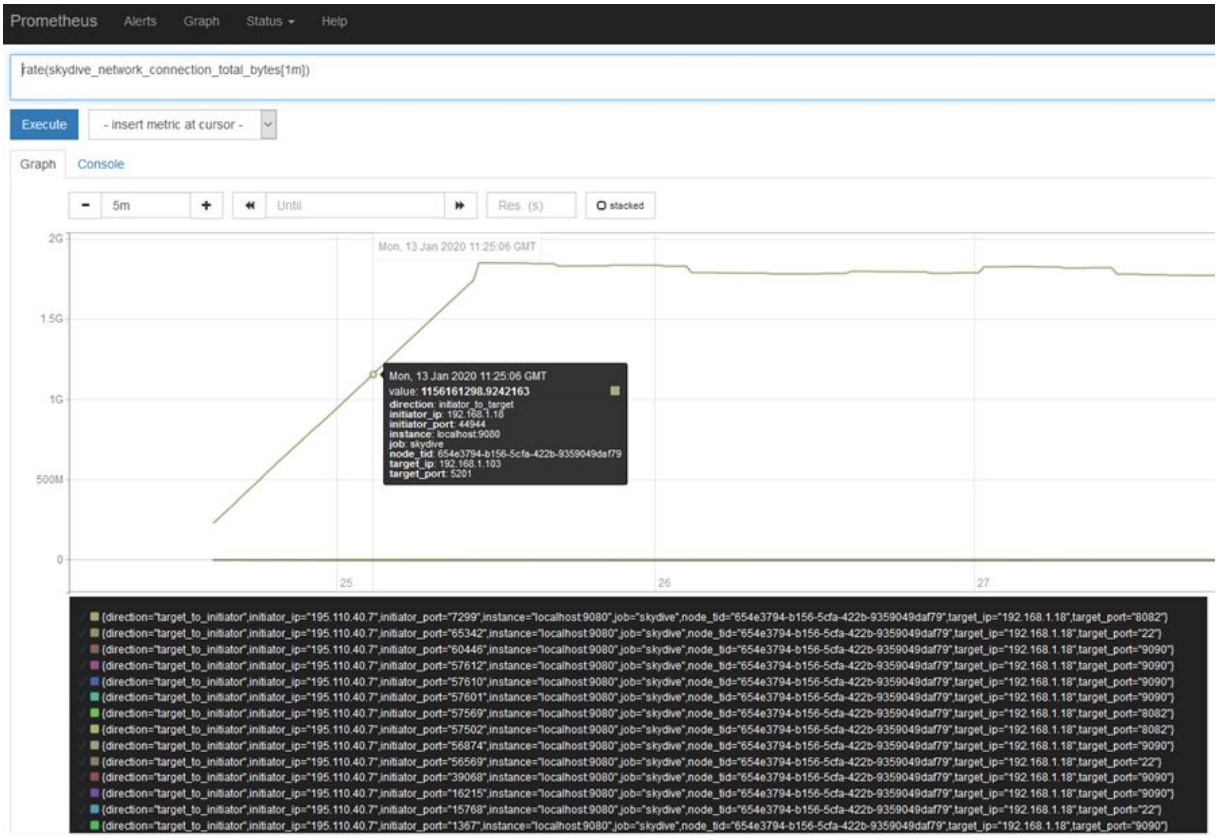

Figure 18 - Skydive-Prometheus connector statistics

The graph shows the rate of data transfer per network connection, as reported by Skydive, in the GUI of Prometheus. This is performed by graphing the rate of change of the connector-produced variable '*skydive_network_connection_total_bytes*', which is accumulated per network connection. Most of the connections have very little traffic, and hover around 0 bytes per second. One connection stands out with a steady rate of about 1.8 GBytes per second for a period of time. We see in the figure the metadata associated with the particular network connection, including initiator and target IP addresses and ports.

This skydive-prometheus connector is work-in-progress and we expect to expand it over the course of the project to provide additional metrics.

4.6 Next steps

As described in the initial paragraph of the previous section, SODALITE has as its goal to be able to deploy and run applications in both HPC and cloud environments. It was also described that the current development has been focused in cloud VM and infrastructure monitoring, leaving the other side of the equation, HPC systems, yet to monitor.

Currently, the *Monitoring* component tracks standard performance statistics in stand-alone mode. In other words, we collect pre-defined statistics, no matter what application is running on the Cloud testbed, and store the data in a database. We do not yet use the data to make autonomous placement decisions. Ultimately, we want to connect the *Monitoring* component with the *Refactoring* component to be able to make autonomous decisions to redeploy applications. Furthermore, it would be most desirable if we could add the ability to selectively monitor particular performance statistics for a specific application. We would want to enable one of the SODALITE actors (Application Ops Expert, Resource Expert, Quality Expert) to be able to specify

particular non-standard resources (perhaps via the IDE) to track in order to debug some anomalous behaviour.

As development progresses, we expect we will discover additional performance parameters that would be useful to track. For example, we currently track via the Skydive-Prometheus connector the total number of bytes transferred per TCP connection. We may discover in the future that some other network related parameters are also useful, and the exportation of those parameters would have to be added to the skydive-prometheus connector.

Specifically, we expect to work on the following items:

- Prometheus:
  - develop an exporter for HPC platform and integrate it with the rest of the system.
- Skydive:
  - improve the implementation of the current Skydive-Prometheus connector;
  - add additional network performance parameters to be exported by the connector;
  - add the ability to specify which particular statistics to collect.

5 Lightweight Runtime Environment (LRE)

The Lightweight Runtime Environment task is responsible for finding the appropriate technology to be used as a runtime environment for the SODALITE applications.

The requirements that the chosen LRE needs to satisfy are:

- it must be able to allow the execution of applications on different platforms in a transparent way;
- it must facilitate the application decoupling.

In this section, we first describe the general concepts of lightweight runtime environments, followed by the related work, which highlights our research contributions. Finally, we present the current development status and the deployment plan for the future.

5.1 Background

Virtualization refers to the act of creating virtual (rather than actual) versions of computer hardware resources including computing, storage and network resources. Hardware virtualisation means the creation of a virtual machine that runs its own operating system but is separated from the underlying hardware resources that are hosting the virtual machine(s). In hardware virtualization the host machine is the one used for the execution of the virtualization, while the guest machine is the actual virtual machine. We need these two terms to distinguish between the software that is running on the host and the software that is running on the guest machine. The firmware that creates the guest virtual machine on the host hardware is the hypervisor.

We distinguish between two types of virtualization:

- full virtualization, with an (almost) complete virtual simulation of the hardware that allow software environments to run unchanged and unmodified, including the guest operating system;
- paravirtualization, where the guest applications are executed in their own individual and isolated domains as if they were each running on their own separate system, but the hardware environment is not simulated thus meaning that the programs and applications have to be specifically modified to run on this guest systems.

Operating-system-level (OS-level) virtualization refers to the operating system paradigm where the kernel allows the creation and existence of multiple isolated userspace instances that each run their individual application software or service. Such instances - commonly known as containers- might look like real computers from the point of view of the applications and programs that are running inside them, however these programs can in fact only see the contents within the container and the resources that have been assigned to the particular container.

There are many various and obvious reasons for virtualisation and containerisation, since this brings many benefits in terms of flexibility, system or application isolation, migration, overhead, availability, fault tolerance and so on.

Within SODALITE we have considered various approaches in terms of virtualization approaches and virtualization levels such as:

- full virtual machines;
- unikernels: Linux containers (LXC)[55] and Kernel Virtual Machines (KVM)[56];

[55] https://linuxcontainers.org/
[56] https://www.linux-kvm.org/page/Main_Page

- containers: where Docker[57] has become the de-facto standard.

5.2 Related work

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.

There is quite a considerable amount of desk and hands-on research that has been done beforehand to support an educated decision on what kinds of technologies should be considered to support the further development of SODALITE. As already mentioned, the main technologies that have been considered were full virtual machines, unikernels and containers.

The virtual machines approach was dismissed since it is in clear controversy with the "light-weight" approach that we are trying to achieve. While the technologies are plentyfull, well known, widely adopted and well supported, the technology itself simply has too much overhead (because of the need to run a full OS, libraries, dependencies …) compared to the other two approaches.

5.2.1 Unikernels

The approach with unikernels was the next in line. This approach is considerably much more light-weight compared to the VM approach. The unikernel approach joins the application and the kernel libraries into a minimum entity that is light-weight, rapidly executable and flexible. Presented below are the unikernel environments that were considered to be implemented or adapted to the needs of SODALITE.

MirageOS[58] is a library operating system that constructs unikernels intended for secure, high-performance network applications capable of running on a variety of cloud computing and mobile platforms. The development of the code can be done on a normal OS such as Linux or MacOS X, and then compiled into a standalone, specialised unikernel that runs under a Xen or KVM hypervisor. Services developed in such a way, run more efficiently, securely and with better control compared to a common software stack.

MirageOS uses the OCaml language, in conjunction with libraries that provide networking and storage support that work under Unix during development, but become operating system drivers when being compiled for production deployment.

IncludeOS[59]: IncludeOS allows you to run applications in the cloud without an operating system. IncludeOS adds operating system functionality to applications allowing to create high-performance, secure and efficient virtual machines. IncludeOS unikernels typically boot in the range of tens of milliseconds and require only a few megabytes of disk and memory.

To run a service with IncludeOS on Linux or macOS there is no need to install IncludeOS, however a few dependencies have to be installed depending on the service that will run. Only explicitly required OS functionalities are compiled into unikernel images. Individual object files are statically linked, and only actually used functions are present in the final binary/VM image.

OSv[60]: OSv was designed to run unmodified Linux applications securely on micro-VMs. It was built from scratch with the intention for effortless deployment and management of microservices. Using

[57] https://www.docker.com/
[58] https://mirage.io/
[59] https://www.includeos.org/
[60] http://osv.io/

OSv it is possible to reduce the memory and CPUoverhead compared to usual OS. Scheduling is lightweight since the application and the kernel cooperate, and memory pools are shared. OSv instances can be deployed directly from a developer IDE or through a continuous integration system. Typically applications can be run unmodified if using OSv. It is possible to access a low level kernel API in order to provide even better performance. OSv is used also in MIKELANGELO[61], a project led by the project partner XLAB where the goal was to bridge the gap between HPC and Cloud, by bringing cloud flexibility to HPC and HPC efficiency and power to the cloud.

Rumprun[62]: Rump solves the problem of the need for driver-like components that are tightly-knit into operating systems by providing free, reusable, componentized, kernel quality drivers such as file systems, POSIX system calls, PCI device drivers and TCP/IP and SCSI protocol stacks.

Rumpkernel is a composition of device drivers. The drivers are taken from existing monolithic OS, in particular from NetBSD. In rumpkernel it is possible to run applications, but libc functionality cannot be used. Rumprun is a rump kernel-based unikernel, basically libc is added to the rumpkernel.

The key concept during rump kernel development is to use existing, unmodified device drivers. This ensures they will work also with broken hardware devices. The unmodified drivers are adopted to the target platform by high-level hypercall interface. The main supported platforms are POSIX style userspace, Xen DomU, and hardware and cloud (x86 bare metal, KVM, VirtualBox).

5.2.2 Containers

Nowadays the complexity of the software applications has become significant. Applications require several dependencies for their compilation, such as specific versions of operating systems, compilers, interpreters, mathematical libraries, and build tools. A well-established way of packing applications with dependencies for easy user-level installation and productivity is achieved via so-called containers. Containers can be used to package entire scientific workflows, software, and libraries, and even data, solving the problem of making the software run reliably when moved from one computing environment to another.

Docker[63] is an open-source technology used mostly for developing, shipping and running applications. With it, applications can be isolated from their underlying infrastructure so that software delivery is faster and independent from specific hardware and environment configurations. Docker's main benefit is to package applications in "containers". By using containers we can create applications that are portable for any system running the Linux operating system (OS) or Windows OS. Although container technology is known already for a considerable time, only the hype around Docker's approach to containers pushed the popularity of containerization into the mainstream.

The main benefit of Docker is that, once an application and all its dependencies are packed into a container, it is ensured that it will run in any environment. Developers using Docker can be sure their applications will not interfere with each other. As a result, containers can be built having different applications installed on it and given to another team, which will then only need to run the container to replicate the initial environment. In this way using Docker tools saves time. A Docker container, as explained above, is a standard unit of software that stores up a code and all its dependencies so the application runs fast and reliably from one computer environment to different ones. A Docker container image is a lightweight, standalone, executable package of

[61] https://www.mikelangelo-project.eu/
[62] http://rumpkernel.org/
[63] https://www.docker.com/

software that has everything needed to run an application – code, runtime, system tools, system libraries, and settings.

The main allure of using containers is the guarantee that containerized software will already have packed the same dependencies and features already in the container and will therefore always function in the same way, regardless of the infrastructure that the container is running on. This can be achieved since containers isolate software from its environment and ensure that it works consistently despite using different underlying systems. VMs are storage intensive since they contain full copies of the operating system, the application itself, all the necessary binaries, and libraries and more. As such they are taking up to tens of GBs of space. VMs can also be slow to start up since all the previously mentioned objects have to be booted. Oppositely containers take up less space (their images are usually only in the range of tens or hundreds of MBs), can handle more applications and use fewer resources. As such, they are more flexible. Moreover, since various applications can run on top of a single OS instance, this is a more effective way to run them.

The additional distinct benefit of Docker containers is the ability to keep application environments isolated both from each other and also from the underlying system. This enables an easy overview and orchestration of the system resources that are hosting the container like its CPU, GPU and network. Not only that, it also makes sure that the data and code remain separate. A Docker container runs on any machine that supports the container's runtime environment. Applications are not tied to the host operating system, in this way the application environment and the operating environment can be kept clean and at the minimum. Container-based apps can be moved from systems to cloud environments or from developers' workstation to server farms.

Udocker[64] is an outcome of the INDIGO DataCloud[65] project. udocker is a basic user tool to execute simple Docker containers in user space without requiring root privileges. Enables download and execution of Docker containers by non-privileged users in Linux systems where Docker is not available. It can be used to pull and execute Docker containers in Linux batch systems and interactive clusters that are managed by other entities such as grid infrastructures or externally managed batch or interactive systems. udocker does not require any type of privileges nor the deployment of services by system administrators. It can be downloaded and executed entirely by the end user. udocker is a wrapper around several tools to mimic a subset of the Docker capabilities including pulling images and running containers with minimal functionality.

Charliecloud[66] uses Linux user namespaces to run containers with no privileged operations or daemons and minimal configuration changes on center resources. This simple approach avoids most security risks while maintaining access to the performance and functionality already on offer. Container images can be built using Docker or anything else that can generate a standard Linux filesystem tree.

Singularity[67] can execute containers like they are native programs or scripts on a host computer. As a result, integration with schedulers is simple and runs exactly as one expects. All standard input, output, error, pipes, IPC, and other communication pathways that locally running programs employ are synchronized with the applications running locally within the container. Additionally, because Singularity is not emulating a full hardware level virtualization paradigm, there is no need to separate out any sandboxed networks or file systems because there is no concept of user-escalation within a container. Users can run Singularity containers just as they run any other program on the HPC resource.

[64] https://indigo-dc.gitbook.io/udocker/
[65] https://www.indigo-datacloud.eu/
[66] https://hpc.github.io/charliecloud/
[67] https://singularity.lbl.gov/docs-hpc

For the SODALITE project, two main requirements are considered for choosing the container technologies to use in the project:

- popularity of the technology, ease of use, and availability of tools and support;
- performance on HPC systems (native support of hardware, for example network and GPUs).

The de-facto standard technology is Docker, which is widely used in Cloud environments as a lightweight virtualization technology. Docker has the ability to share the host's kernel resources to reduce the virtualization overhead. However, it was mainly designed for the deployment of microservices, so it presents some drawbacks when it used on an High Performance Computing (HPC) system, mostly related to security (it relies on a root owned daemon process to build container systems) and performance (it isolates from the host infrastructure, so it becomes complex to take advantage of the host's hardware acceleration)[60, 61]. For these reasons, other technologies have been specifically developed for HPC systems. A comparison of these technologies can be found elsewhere[62, 63]. Giving our requirements, we decided to support Docker for Cloud and Singularity for HPC systems. 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.

5.3 Development status

One of the main goals of development of the runtime environment as well as the whole infrastructure is simplicity and the ability to run similar components in different places on the infrastructure. Since the SODALITE environment is established by two main parts, the HPC and the Cloud infrastructure, we had to adopt an approach that would optimally work on both.

Full virtual machines are not suitable since they are simply too big and require a lot of effort to administer and upkeep, as such they do not really work on the HPC and reconfiguration based on the actual needs is complicated both to execute and to automate. Virtual machines are a viable solution when we are producing fully featured services that have to run in High Availability mode (HA) 24/7.

Unikernels have obvious benefits compared to full VMs. They are much faster and responsive and comparatively have a much smaller attack surface compared to VMs, thus making them a more secure option. Regardless, these benefits do not outweigh the disadvantages that unikernels have. The case against Unikernels is that they are notoriously difficult to build and configure; dependency-wise they are very complicated; making them complicated to upkeep; and lastly the virtualization trends are simply moving away from unikernels in favour of other technologies and approaches. Thus we also abandoned the unikernel approach.

We have therefore decided to go forward with the development of the system using containers. The technology is already at an advanced state, well defined and supported and nonetheless Docker has come to be a de-facto industry standard in the field. In a technological sense, containers offer flexibility, responsiveness, adaptability and applicability to various systems. An additional compelling fact is also that there is a widespread community of users already intensively working with containers that we can rely on.

Docker containers are supported both in Cloud and HPC applications. The SODALITE testbed comprises two main pillars: the Cloud infrastructure and the HPC infrastructure. The Cloud section of the testbed is set up by utilizing Kubernetes for the orchestration of container based

applications and OpenStack for the applications that need to be deployed using VMs. The HPC is based on Singularity in conjunction with the Torque scheduling and management to deploy and run the containers. The main reason to establish the HPC environment using Singularity as the basis for the system that is running the containers is twofold. The first is the simplicity of use and migration. Singularity allows users to pack everything they need in their computing set-up into one single file, this can then be uploaded to a cluster system where it can be easily deployed and run without major hassles. The second and main reason for the use of Singularity is the security aspects that it offers. HPC systems are multi-tenant environments meaning that they are accessed and used by a number of users that are in fact all using the same resources at any given time. Running Docker directly opens the possibility to have a security breach scenario where one user could obtain root privileges on the host itself via the Docker console and run its code as root or even worse, access other users' data. Due to the specific configuration of Singularity, the root filesystem cannot be accessed by the individual user. This practically means that each individual user has root privileges only within its own individual Singularity containers and thus impossible to perform any privileged operation that could compromise any of the security aspects of other users of the system. The other added benefits of Singularity are of course the native support for parallel job management systems like Torque and SLURM and the native ability to use MPI.

The full and in-depth description of the layout and technical specifications of the testbed can be found in Section 3 of Deliverable D6.1[16] so we will offer only an outline of the main components and the reason for choosing the systems running upon them.

As of the time of writing this document all the building blocks of the SODALITE runtime environment have been established at least in its basic arrangement as a prototype environment, thus enabling the development of further components. In the following periods of the project, these developments will continue in order to satisfy the needs of the project.

5.4 Next steps

The future development of the SODALITE LRE depends on the development of the individual components of the environment. The fundamental objective of the future development is to achieve a tightly coherent solution that follows the ideas of practical usability, community acceptance and the development and integration of open source technologies and code. The development must of course be open to integrate and adapt to new technologies which is not trivial since IaC is a new field and still highly volatile in terms of trends and available solutions. That is why a careful analysis of the available technologies and a careful selection is of crucial importance for effective development and long term success of the SODALITE project. According to this we will consider:

- adding application runtime parameters monitoring for both Singularity and Docker (M24),
- address any new container technology that will attract a larger community attention and fit into overall SODALITE plans (M36).

In any case, the development of the SODALITE LRE will follow the overall project plan and the development of the individual components within the testbeds.

6 Predictive Deployment Refactoring

The main objective of the predictive deployment refactoring is to refactor or adapt the deployment model of an application at runtime in order to prevent the violation of the performance goals (e.g. latency, throughput, resource utilization, and cost) of the application. The refactoring opportunities can be explicitly identified and modelled by the software engineer. They can also be discovered at runtime by the refactoring component. In particular, the patterns and anti-patterns (e.g., software system performance patterns and antipatterns) are used to discover the refactoring opportunities.

In this section, we first describe the general concepts of deployment refactoring followed by the related work, which highlights our research contributions. Next, we present the architecture of our deployment refactoring support, followed by a detailed description of each major component. Finally, we present the current development status and the deployment plan for each component.

6.1 Background

According to Martin Fowler[64], *"Refactoring is the systematic process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."* Along the same lines, within the context of SODALITE, we define *deployment refactoring as the systematic process of changing the deployment model/topology of a software system without altering the external behavior of the system*. A key goal of the deployment refactoring is to improve the overall utility and quality of the system by applying a series of small behavior-preserving transformations to the deployment model of the system. The refactoring decisions generally need to strike a balance between different competing quality attributes such as performance, resource usage, security and privacy risks, and cost.

In software development, the code refactoring is commonly performed to remove the "code smells", which are any characteristics in the source code of a system that possibly indicates a deeper problem/quality issue. The deployment models can also have "smells". Thus, the deployment refactoring also aims to detect and remove the smells in the deployment model/topology of a system. Common types of smells that impact on non-functional quality attributes of a system include performance anti-patterns[65], privacy anti-patterns[66], and security anti-patterns[67]. In SODALITE, we define such performance, privacy, and security anti-patterns at the level of a deployment topology/model, and develop techniques to detect them. Different types of anti-patterns need different types of detection methodologies, for example, machine learning based approaches for performance anti-patterns, and knowledge-based reasoning approaches for security and privacy anti-patterns.

One materialization of deployment refactoring is dynamic resource allocation. Deployed resources (VM, containers) can be changed at runtime in order to closely match the incoming workload. Dynamic resource allocation is key to provide a desired quality of service (QoS) to users while optimizing costs. QoS is usually defined as Service Level Agreements (SLA), that are requirements on application-level metrics such response time. Without dynamic resource allocation, resources can be either not enough (under-provisioning) causing violations to Service Level Agreements and degradation of performance, or over-provisioned when the resources are more than needed leading to unnecessary costs.

In the context of SODALITE, dynamic resource allocation must target heterogeneous resources. For cloud deployments CPUs and GPUs must be carefully managed in order to keep QoS under control, on hybrid deployments HPC must be also taken into account. GPUs can boost performance of applications but they can only be allocated at a coarse granularity, while fractions on CPUs can be quickly allocated to each running process (container). To address highly dynamic execution

environments the solutions mainly focus on vertical scalability of containers that allows for fast reconfiguration of resources avoiding the need to boot or reboot any running components.

6.2 Related Work

In this section, we review the existing studies pertaining to the main tasks of our predictive deployment refactoring support: resource elasticity, deployment configuration selection and adaptation, and resource discovery and composition.

6.2.1 Resource Elasticity

In cloud computing, the vertical and horizontal scalability and elasticity of cloud applications have been extensively investigated. Vertical elasticity considers increasing or decreasing computing resources (e.g., CPU, GPU, and memory) used by a single node (physical machine, virtual machine or container). Horizontal elasticity considers adding or removing instances of computing resources associated with an application [13]. In this SODALITE task, we consider a combination of heuristic and control-theoretical models to achieve elasticity for modern containerized software systems such as machine learning systems (e.g., TensorFlow) and heterogeneous resources (e.g., GPUs and CPUs). Given user-defined Service Level Agreements (SLAs), SODALITE supports the control of multiple applications running concurrently on a shared cluster of virtual or physical machines.

In the literature one can find several approaches regarding resource provisioning [19, 20, 21, 32]. Serhani et al. [32] introduced an orchestration architecture that supports self-adaptive IoT workflows, including monitoring the state of the environment (e.g., utilization of a VM, state of a workflow task), detecting abnormalities (e.g., overloaded VM or failed task), and adapting the environment to recover (e.g., replacing a VM). Beloglazov et al. [22] presents a set of tools including heuristics, resource allocation policies, and scheduling algorithms with the goal of satisfying SLA requirements on multiple running cloud applications. Compared to SODALITE, they do not consider GPU executions, neither do they handle vertical scalability. Lakew et al. [23] use control theory to dynamically allocate CPU and memory resources to satisfy SLAs. Similarly to SODALITE, they use containers to wrap cloud applications exploiting their fine-grained means to reconfigure resources at runtime. Compared to SODALITE, this work handles only a single application at a time and they do not consider GPUs.

Several studies (e.g., [24]) highlight the advantages of GPU-based executions motivating our effort in SODALITE of handling heterogeneous resources. GPU executions are usually studied in the context of Machine Learning (ML) applications since they are composed by several matrix-based operations that can be easily parallelized and boosted by graphic-dedicated hardware. These applications are often executed using a dedicated framework such TensorFlow, PyTorch and Keras that allow users for a simplified management of hardware resources. TensorFlow, similarly to other ML frameworks, provides APIs to speed-up applications using GPUs. By default, it schedules requests on GPUs as in SODALITE but CPUs are not managed dynamically and static allocations are only possible. Moveover, TensorFlow is not aware of SLAs and can only speed-up resources according to low-level metrics. To improve TensorFlow default policies users must heavily change their codebase.

Several works in literature handle GPU executions. For example, Xiao et al. [25] propose Gandiva, a tool based on Kubernetes that aims to improve the resource usage of long-lasting deep learning jobs. Gandiva exploits the iterative nature of these kinds of jobs to efficiently allocate GPUs among different concurrent computations. Gandiva only allocates GPUs, while the SODALITE approach exploits heterogeneous resources. Lu et al. [26] present a tool called Augur, that is able to predict the performance of CNNs (Convolutional Neural Networks) in the context of mobile computing. Augur analyses the structure of the CNN and produces a performance model that can predict the

time and memory needed to run the job on a given hardware. As SODALITE, they consider both CPUs and GPUs executions but they do not control the application at runtime.

6.2.2 Deployment Configuration Selection and Adaptation

There has been extensive research on self-adaptive software systems, including self-adaptive cloud applications, over the last few decades [1, 6, 38]. Both proactive and reactive adaptations have been considered. In the proactive mode, the adaptation is performed before the need for adaptation occurs, e.g., based on a predicted drop of a performance metric. The reactive mode is the opposite of the proactive mode. One of the most widely accepted approaches for self-adaptation is *dynamic software product lines* [7], which models an adaptive software system as a set of system variants where the adaptation is performed by switching between the variants as the context and requirements change. This SODALITE task applies and further extends the self-adaptive system research findings for supporting the runtime refactoring of the deployment of Cloud or HPC (high performance computing) or hybrid (Cloud and HPC) applications. In particular, we consider the heterogeneity in the Cloud and HPC resources as well as the heterogeneity in the deployment architecture. The heterogeneity in resources can be exploited to minimize the resource usage cost [8].

In the research literature, there are studies on the optimization of the deployment of different types of applications including embedded systems [9], distributed applications [10], and cloud applications [11, 12, 22]. The key issue is the optimization problem of assigning the software components to the hardware devices. In this SODALITE task, we formulated the deployment optimization problem differently adopting the view of the *dynamic software product lines*. With the explicit modeling of the deployment variability, the deployment optimization problem becomes the selection of the optimal deployment variant among the allowed set of variants. Moreover, we employ a data-driven approach (machine-learning) to model the behavior and performance of the deployment variants, and consider the dynamic discovery of new deployment options based on patterns. Our deployment improvement also considers the detection and removal of performance anti-patterns. Furthermore, the Vehicle-IoT use case of SODALITE requires the coordination between the application-level (functional) adaptation and deployment refactoring.

6.2.3 Resource Discovery and Composition

Resource discovery is a process of finding available computing resources to fulfill the end-user (application) requirements. Resource composition is a process of selecting and allocating/using the discovered resources. Several recent studies [27, 28, 29] have systematically reviewed the existing research on resource discovery and composition. In the research, both centralized and decentralized architectures have been proposed for resource discovery and composition. Compared with the centralized architecture, the decentralized architecture stores the resource information in the participant nodes instead of a central database/knowledgebase. Among the studies on cloud resource discovery and composition, Ting Yu et al. [30] developed a DNS-based method for discovering cloud resources in an inter-cloud environment, where resources among different clouds are shared. Khethavath et al. [31] proposed a mechanism for discovery and allocation of resources in a geographically distributed cloud, whose nodes are the mobile devices of the users. They modeled the cloud as a multi-valued distributed hash table to support efficient discovery of devices, and used an auction model to allocate the device resources optimally to the users. Al-Sayed et al. [39] proposed a cloud service discovery framework that utilizes the ontologies to build standardized semantic specification of cloud services and to provide a natural language based search interface. Djemaa et al. [40] presented a crawler that can discover and categorize Cloud services available on the Web based on semantic similarity between cloud service descriptions and a cloud service ontology. Modica and Tomarchio [41] proposed a framework that can discover security-enabled cloud services by matching the security requirements of the users, and the security capabilities of the services, specified as security policies. A security ontology was

used to annotate the provider-specific policies so that semantic matchmaking can be performed on services, considering their policies.

Among the studies on TOSCA-based resource discovery and composition [33], Brogi and Soldani [34] identified four types of possible matchings between ServiceTemplates and NodeTypes, that can be used to instantiate abstract TOSCA NodeTypes in a deployment topology. Soldani et al. [35] developed the support for matching, adapting, and reusing existing fragments of the application deployment topologies from a shared TOSCA repository to implement the components in a given application topology. Brogi et al. [36] developed a tool that can discover the Docker-based nodes that can host the application components, and use the discovered nodes to complete the partially specified deployment topology (TOSCA) of the application. Antequera et al. [37] presented a middleware that can recommend the custom TOSCA templates from a catalogue of such templates, each modeling different cloud resources and their configurations. It can monitor the application behavior, and trigger adaptation rules as necessary. The adaptation actions can be fine-grained (e.g., changing the size of a VM cluster) as well as coarse-grained (e.g., replacing an existing template with a new custom template).

Compared with the existing works, SODALITE uses semantic web technologies for discovering TOSCA-compliant resources and deployment model fragments. Our semantic matchmaker considers constraints on node attributes, node requirements, node capabilities, and node policies. It also builds a pattern-based high-level abstraction for resource discovery. Furthermore, we apply machine learning to predict the impact of the discovered resources of the performance of the application.

6.3 UML use cases

Predictive Deployment Refactoring task is responsible for the SODALITE UML use case UC9 **Identify Refactoring Options** (see Figure 19). The refactoring of the deployment model of a running application is performed in response to the potential violations of the application goals, which is determined using the monitoring data. The refactoring can find and enact a new deployment model for the application that can resolve the detected goal violations. At design time, the App Ops Expert provides the initial set of refactoring options. At runtime, the new deployment options are discovered. A valid selection of a subset of deployment options results in a valid deployment model variant for the application. In addition to the global optimization of the deployment model, the resources of the individual nodes in the deployment model can also be locally managed.

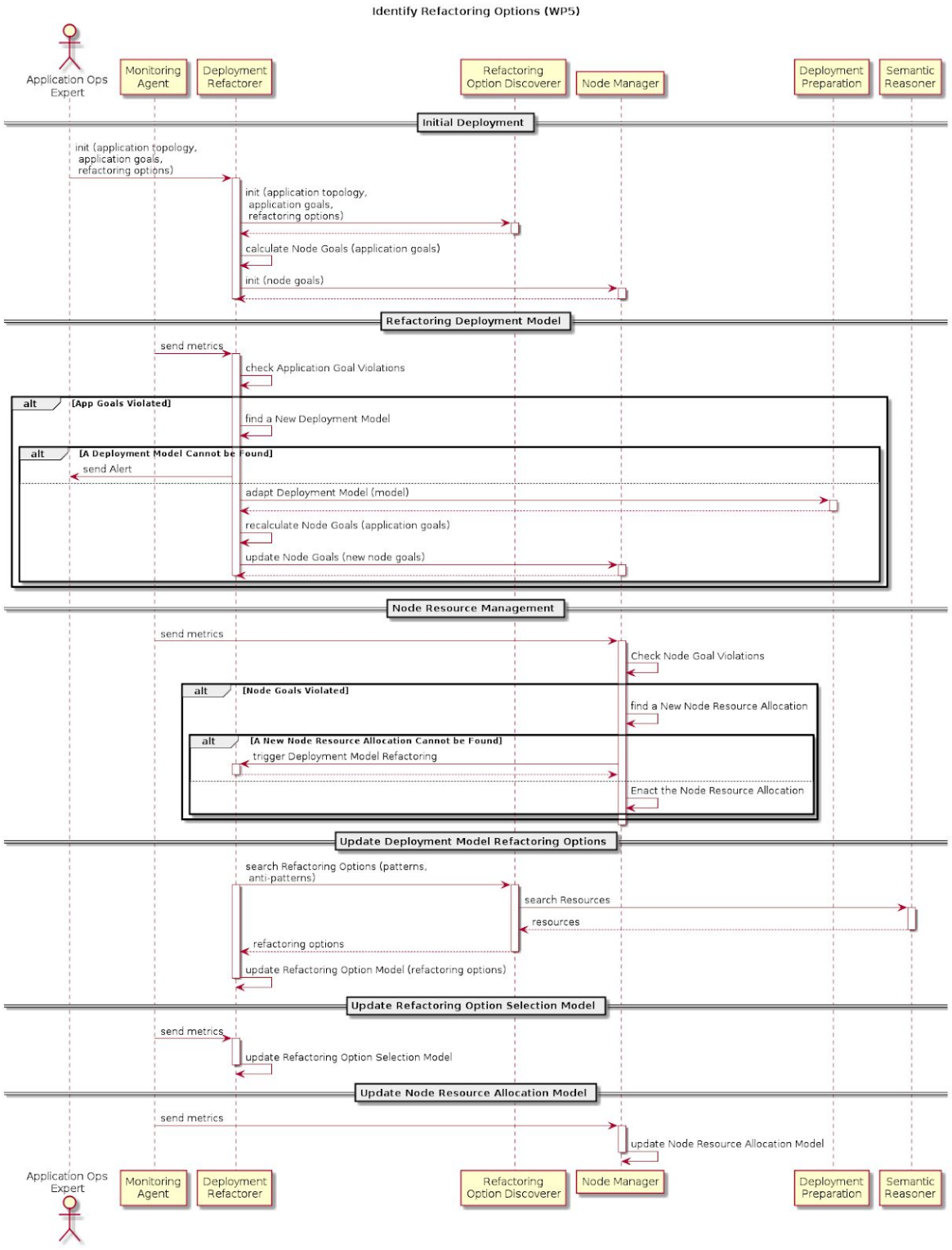

Figure 19 - UC9 Identify Refactoring Options

6.4 Architecture

Figure 20 provides an overview of our predictive deployment refactoring approach. We build our approach on the top of the research works on the data-driven approaches to self-adaptation and performance engineering [1-4].

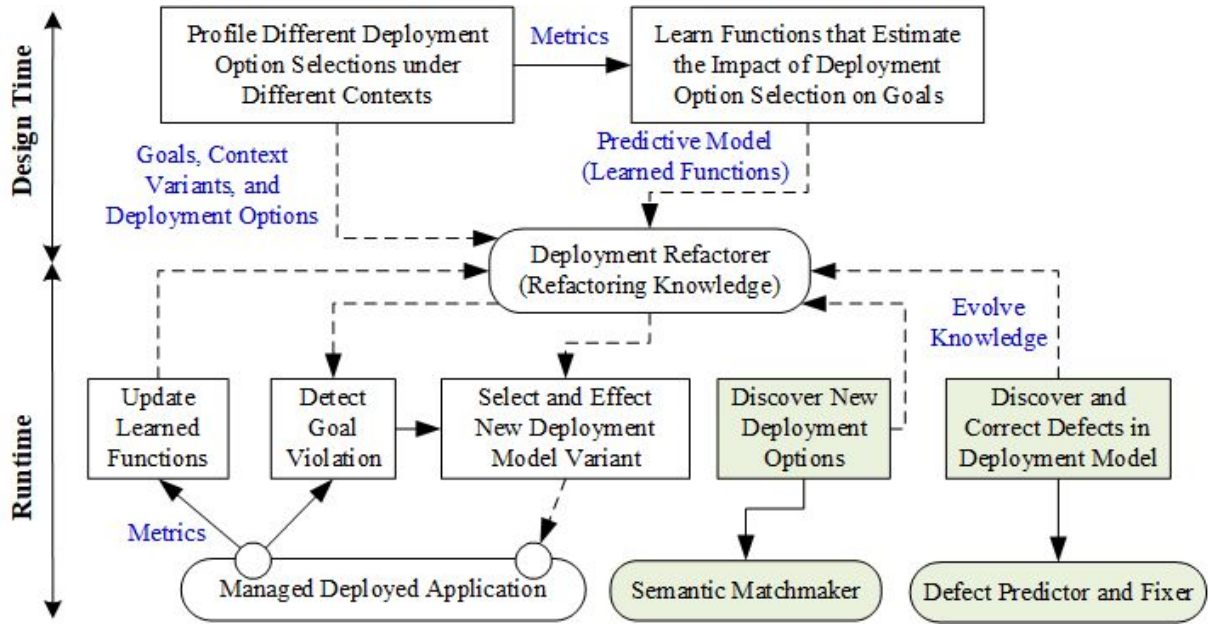

Figure 20 - An overview of SODALITE predictive deployment refactoring

The individual components of an application can be deployed in different ways using different resources (e.g., a small VM and a large VM) and deployment patterns (single node, cluster with load balancer, with or without cache, with or without firewall). We call these deployment possibilities as (application component) deployment options. A valid selection of deployment options results in a valid deployment model variant for the application. The deployment refactoring requires a model that can estimate the impacts of a given deployment option selection on the QoS metrics such as latency and cost, under different contexts such as different workloads. We build a machine learning based predictive model for this purpose. At the design time, we profile the deployment variants to collect the data required to build the machine learning model. At runtime, we use the monitoring of the running application to collect the data and to update the learned model as necessary. The predictive model enables the deployment refactorer to predict the potential violations of the application goals, and consequently to find alternative deployment model variants.

As the deployment environment evolves, the new resources will be added and the existing resources will be removed or updated. Consequently, we need to discover new deployment options as well as changes to the currently used deployment options. WP3 builds the semantic reasoning and matchmaking capabilities. The deployment refactoring adopts and further extends these capabilities to discover/build new deployment options.

The individual components of an application can have the performance goals, which can be potentially derived from the application-level goals. The resources in the node that hosts a component need to be managed dynamically (allocated/deallocated) as necessary to maintain the performance goals of the component.

The modeling of deployment options (generally, the deployment variability) is supported in the SODALITE modeling support (WP3). The deployment options are represented at the TOSCA blueprint level utilizing TOSCA feature called substitution mappings, which supports describing substitutable nodes.

6.4.1 Deployment Refactorer

Deployment Refactorer decides and carries out the refactorings for the deployment model of an application at runtime. To implement the reactoring decision making, it exploits both rule-based approaches and data-driven approaches as appropriate.

The capabilities of Deployment refactorer are:

- monitor QoS (Quality of service) metrics such as performance and cost metrics;
- monitor events (e.g., the change of the location of the user);
- update the functions that estimate the impacts of deployment option selections on metrics;
- detect the potential violations of application goals;
- find and enact a new deployment model variant (a new deployment option selection);
- discover new deployment options or updates to the existing deployment options and update the refactoring knowledge;
- discover the defects in the deployment model (e.g., performance anti-patterns) and if possible correct the defects and otherwise, alert the App Ops Expert;
- assign and update the goals for individual components of a deployed application.

Software dependencies

- Python 3.7
- Java 1.8
- Drools 7.31.0 (rule-engine)

Composed of

- REST API. It offers service operations to be used by App Expert to provide deployment option models and application goals models.
- Backend. It implements the business logic of the component. It consists of:
    - Refactoring Knowledge-base, which contains the knowledge required by the refactoring decision making.
    - Monitor, which collects the data from the running application via the monitoring API.
    - Model Learner, which learns the machine learning model for predicting the impacts of the deployment option selection on the goals.
    - Goal Violation Detector, which detects the violations of application goals.
    - Deployment Variant Generator, which can generate a new deployment model variant.
    - Deployment Variant Effector, which can enact a given deployment model variant via the REST API of the *Orchestrator*.
    - Defect Predictor and Fixer, which can predict the defects (e.g., performance antipatterns) in the running deployment model and suggest fixes for them.

Roles that interact with the component

- Application Ops Expert

Depends on

- Refactoring Option Discoverer
- Node Manager

- Deployment Preparation API
- Orchestrator
- Semantic Knowledge Base
- Semantic Reasoner
- Monitoring

Flexibility, resiliency and scalability aspects

The flexibility of *Deployment Refactorer* is supported through 1) declarative ECA based rule base, 2) black-box performance modeling and prediction with machine learning, 3) dynamic discovery of refactoring options (via *Refactoring Option Discoverer*). The rule-base allows the software engineer to define custom, application-specific rules without altering the implementation of the Refactorer or halting the running Refactorer instance. With the machine learning based approach, Refactorer can model the performance of different applications in an application independent way. The dynamic discovery capability makes the knowledge and capacity of the Refactorer dynamic and evolvable, increasing its flexibility to refactor different applications deployed and operated under different contexts and environments.

As regards to the scalability, *Deployment Refactorer* can use a horizontally scalable machine learning framework for training machine learning models, for example, Apache Spark. The data collected by the *Deployment Refactorer* are stored in a relational database, which is by default, SQLite. This database can be changed without impacting the implementation (source code) of the *Refactorer*, for example, by a MySQL or PostgreSQL cluster. The data can also be stored in a data lake like Apache Spark. The ECA rule engine uses JBoss Drools production rule engine, which can be deployed in a horizontally scalable cluster. The horizontal scalable deployment of the Deployment Refractorer and the persistence of the data also help to make the *Deployment Refactorer* resilient with respect to the changes in the environment, such as monitoring data spikes and node failures.

Repositories

https://github.com/SODALITE-EU/refactoring-ml

Demo

https://www.youtube.com/watch?v=Xl8E8izcFqA

This video presents how the SODALITE Deployment Refactorer uses a rule-based approach to refactor an existing deployment.

6.4.2 Node Manager

The *Node Manager* contains components that allow for the vertical-scalability of resources. Given initial allocations and SLAs for each deployed application, containers are equipped with dedicated control-theoretical planners, implemented as PID (Proportional–Integral–Derivative) controllers, that are in charge of dynamically changing the resources allocated to containers to reach a given set-point usually defined as a desired response-time. PID controllers are able to compute the next allocation in constant time and containers are re-configurable (vertical scalability) in hundreds of milliseconds. This allows for an extremely fast and fine-grained control which is able to react to temporary peaks in the workload or changes in the execution environment.

The *Node Manager* supports three types of (sub-)systems that could potentially compose a complex heterogeneous system: microservices, big-data application, and machine learning. In

particular, while microservices and big-data batch applications usually run on standard hardware, machine learning applications can exploit both GPUs and CPUs to fulfill a given SLA. The *Node Manager* is able to manage all these different types of resources, optimizing their usage.

Since each container deployed in the node has its own independent controller, an additional component called supervisor solves possible situations of resource contention that could appear when the aggregated resource demand requested by controllers is greater than the available resources. For this reason, the supervisor can downscale each controller resource request using different policy, from proportional to priority-based and apply only feasible allocations.

The capabilities of *Node Manager* are:

- vertical scalability/elasticity of a single node (CPU and GPU re-configuration);
- horizontal scalability/elasticity for a cluster of nodes.

Software dependencies

- Python 3.7

Composed of

- REST API (in development). Allows to set new goals (e.g., new SLAs) for the *Node Manager*.
- Load-balancer. It dispatches requests among the different containers running in the node, if an application allows it could send the requests to either CPUs or GPUs.
- Monitoring system (ad-hoc, Prometheus integration in development). It gathers metrics that are used by the Controllers to plan new resource allocations.
- Controllers. They are PID controllers that change the resource allocation in order to fulfill the SLAs. One controller per container is deployed.
- Supervisor. It manages resource contention that can occur in the node.
- TensorFlow Serving. It is used to deploy TensorFlow applications in containers.
- Docker. It is used to enact the resource allocations computed by the Controllers on running containers

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

- There is no direct interaction of any SODALITE actor with this component.

Depends on

- Deployment Refactorer
- Orchestrator
- Monitoring

Flexibility, resiliency and scalability aspects

Node Manager provides a highly scalable architecture. Distributed controllers compute the next resource allocation in constant time and they do not need to be synchronized. This is achieved by fairly distributing the workload to the different distributed containerized copies of the applications. Each container is equipped with a dedicated controller that works only on local data (i.e., only on the portion of workload sent to the controlled container). Controllers dedicated to the same application are instructed to have the same set-point meaning that if locally each controller fulfils the goal, overall the global objective is satisfied.

Being based on control-theory, *Node Manager* is also highly resilient to disturbances as unexpected workload distributions, performance degradation of the cloud. Property like stability, steady-state error and overshooting can be easily measured and tuned.

Node Manager is also flexible since it can control applications of different kinds concurrently without modifications. If the performance of different applications vary a lot an ad-hoc tuning of parameters can be required. This step can be easily automated using state-of-the-art control theoretical techniques.

Repositories

https://github.com/SODALITE-EU/refactoring-ct

Demo

https://www.youtube.com/watch?v=V9iRQIFH5C4

This video presents an extension of TensorFlow based on heuristics, control theory and containerization that allows for the efficient control of heterogeneous resources (CPUs and GPUs) in order to fulfill requirements over the response time (Service Level Agreements).

6.4.3 Refactoring Option Discoverer

Refactoring Option Discoverer can discover new refactoring options as well as the changes to existing refactoring options.

The capabilities of *Refactoring Option Discoverer* are:

- discover the refactoring options that satisfy a given set of constraints (e.g., an OpenStack with 3 CPUs and deployed in a Netherlands data center);
- discover the refactoring options, which are instances of infrastructure design patterns or antipatterns;
- discover the changes to the currently used refactoring options.

Software dependencies

- Python 3.7
- Java 1.8
- Eclipse RDF4J

Composed of

- Semantic Matchmaker. It can perform ontological reasoning to find the (substitutable) deployment options. It connects to the SODALITE semantic knowledge base via the Semantic Reasoner API.
- Deployment Option Diffchecker. It can identify the changes to the currently used deployment options.

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

- There is no direct interaction of any SODALITE actor with this component.

Depends on

- Deployment Refactorer
- Monitoring Agent

- Semantic Reasoner

Repositories

https://github.com/SODALITE-EU/refactoring-option-discoverer

6.5 Development status

Deployment Refactorer adopts both rule-based (event-driven) approach and data-driven approach. The rule-based approach is used to implement the refactoring decision making required by the first version of the Vehicle IoT SODALITE use case. In particular, we applied to an scenario where a driver/user switches from Germany to Italy (location change) and consequently, in order to keep data near to the user, the refactorer changes the current deployment of the application by deploying or undeploying components (and associated VMs) depending on the user location. We have implemented the rule-based refactorer using Drools Rule Engine (https://www.drools.org/). App Expert can define the refactoring policy with the domain specific language (DSL) provided by Drools.

As regards to the data-driven approach to refactoring decision making, we have so far covered the design time part in Figure 20. In order to build our framework, we use the RUBiS benchmark application, which is widely used by cloud benchmark research. To obtain heterogeneity and variations in deployment topology and resources, we modified the application and deployed it in the Google Cloud using different types of virtual machines. Then, we profiled deployment variants under different workload ranges, and we are currently building a machine learning based prediction model for performance using the profiled data. This model will enable to estimate the impact of a given deployment model variant (a particular selection of component deployment options) and thus to select and switch between deployment model variants as the workload varies and the performance goals are violated. Section 4 of Deliverable D3.3 [18] presents our machine learning based model in detail.

We implemented the *Node Manager* as an extension of Kubernetes. It consists of a special Kubernetes pod that contains the control-theoretical planners that re-configure (i.e., vertical scalability) other containers dynamically. It currently supports TensorFlow applications in inference mode that can exploit both GPUs and CPUs. The *Node Manager* is able to continuously change their resource allocation in order to efficiently fulfill requirements over the response time (e.g., response < 0.5s). We tested the implementation on an Azure cluster and five different benchmark applications: Skyline Extractor from the Snow SODALITE Use Case, GoogLeNet, AlexNet, ResNet and VGG-16.

Kubernetes itself provides means to automatically scale containers/pods. In particular, the Horizontal Pod Autoscaler (HPA) is a component that is able to change at runtime the number of pod instances according to predefined and custom metrics. SODALITE components change resource allocation at a fine-granular level by reconfiguring existing containers according to SLAs (vertical scalability). HPA is complementary with the NodeManager architecture. We plan to integrate Kubernetes HPA as part of the SODALITE horizontal scaling system.

Control-theoretical planners, distributed in the cluster (one per container), compute the optimal (according to their model) resource demand of each application. Resource demands can be either actuated as is by the supervisor or downscaled in case of resource contention. Computed resource demands can be aggregated at an application-level, machine-level or system-level. These three metrics are KPIs that can be exploited by horizontal auto-scaling systems to both autoscale container instances (as the HPA) or even to change the number of cluster nodes when needed. This

way, the complexity of computing the actual resource need of the system is segregated at the NodeManager level and simpler systems can exploit its work to refine the resource allocation.

Kubernetes Vertical Pod Autoscaler (VPA) provides a reconfiguration mechanism for containers already in execution as in SODALITE. VPA (currently available in beta version) requires that containers be restarted to enact reconfiguration, and thus the result is inherently slower than SODALITE that allocates resources while applications are running. Moreover, VPA cannot be always used in conjunction with HPA while SODALITE is designed to support both horizontal and vertical autoscaling. Finally, both HPA and VPA do not directly manage GPUs as SODALITE does.

As regards to *Refactoring Option Discoverer*, we have implemented the basic support for the discovery of the refactoring options that satisfy a given set of constraints (e.g., an OpenStack VM with X number of CPUs, and deployed in a data center in Germany). The implementation uses the semantic knowledge-base and reasoning capabilities developed in WP3.

6.6 Next steps

The following are the next steps for the Deployment Refactorer.

- *Complete the runtime part of the Deployment Refactorer (see Figure 20).* This requires the integration of a *Deployment Refactorer* with *Monitoring Agents*. The refactoring decisions need to be made at runtime based on the monitored data. The performance models built offline using profiled data need to be updated based on the feedback on the impacts of the refactoring decisions.

- *Complete the integration with Node Manager. Node Manager* performs fine-grained resource management decisions at the level of a cluster or a single node, hosting a component of the application deployment topology. *Deployment Refactorer* makes controlled changes to the application deployment topology itself such as replacement of a node or a fragment of the topology, and undeployment of one or more nodes. By combining these two deployment adaptation strategies, SODALITE develops a novel multi-level deployment adaptation scheme for cloud and HPC applications.

- *Complete the refactoring logic required for Vehicle IoT Use Case.* Vehicle IoT Use Case provides unique deployment adaptation use cases, and thus the potential for research innovation. The use case is deployed over cloud, edge, and HPC environments. Both event-driven ad-hoc adaptation and data-driven (ML) adaptation are necessary. The adaptations are driven by different quality attributes such as performance, energy (thermal data of devices), security (GDPR). Moreover, the application features can also be adapted, potentially without changing the deployment topology.

- *Implement the defect prediction and correction for the deployment model instance of the running application*. The semantic decision support and static optimization tasks of SODALITE (WP4) ensure that the initial deployment of the application is defect-free and optimized. However, as the deployment model of the application evolves over time, new defects such as performance and security anti-patterns can be introduced.

The next steps for the *Node Manager* are:

- Complete the integration with *Monitoring Agent* and *Deployment Refactorer*. The *Node Manager* requires to be properly integrated into the SODALITE infrastructure. On the one hand, *Node Manager* will exploit monitoring data collected by *Monitoring Agent* to properly schedule requests on heterogeneous devices and optimize resource allocation. On the

other, it must cooperate with the *Deployment Refactorer*, which can modify the available resources and change the set points of the control-theoretical planners.

- Fully integrate control-theoretical planners into Kubernetes and other orchestrators. The *Node Manager* can be seen as an advanced controller for containerized applications running on heterogeneous hardware. The integration with Kubernetes will enhance the usability of *Node Manager.* Kubernetes pods will be automatically controlled by *NodeManager* given user-defined requirements.

The following are the next steps for the *Refactoring Option Discoverer.*

- *Improve the constraints-base discovery of new deployment options*. In addition to node attributes, the selection of deployment options or resources needs to consider more constraints such as node capabilities, requirements, and policies. Thus, the required semantic matching capabilities should be developed.

- *Support pattern-based discovery of new deployment options*. In order to perform pattern based refactoring of a deployment topology, we need to be able to discover the available instances of infrastructure design patterns.

- *Support discovery of changes to the existing deployment options*. The structure and properties of the currently used deployment options can change overtime, which should trigger refactoring decision making.

7 Conclusions

This deliverable D5.1 has reported on the status of the SODALITE Runtime Layer at the end of the first year of the project. We have given an overview of the approach taken in SODALITE with regards to the deployment and runtime, and then detailed the individual components and their current status.

We summarize below the general status of the activities and the next steps.

SODALITE decided to take the approach of distributing the software to be deployed taking advantage of **Lightweight Runtime Environments** (LRE), like unikernels or containers. For this reason, we reviewed appropriate LRE technology to use in the SODALITE framework. As a result of this thorough review, we decided to use Singularity for HPC infrastructure and Docker for the rest of infrastructures.

For the deployment task, we have selected xOpera due to its free distribution and native support of TOSCA standard. Using xOpera as a base **orchestrator**, we can deploy on a number of infrastructures that it initially provided (e.g. OpenStack). Additionally, as part of the SODALITE activities, xOpera has been improved:

- It now offers a REST API , which makes it easier to be invoked by third party components, like the SODALITE IDE;
- We are able to deploy applications on HPC infrastructures accessing the HPC through SSH, thus providing the support of the hybrid infrastructure - Cloud and HPC;
- As part of SODALITE the xOpera orchestrator has been set up to run in docker containers. An instance of xOpera (3 VMs running the REST API and core + Image Registry + database) has been installed on the cloud testbed to be used by the consortium;
- xOpera now offers separate input yaml files which can be used to store secrets without adding them to the blueprint itself. This functionality has been also included in the REST API calls;
- xOpera is in an active development phase, meaning that missing features are constantly being added and patches released. The missing SODALITE features are planned in the development of the REST API and xOpera as needed.

Prometheus has been selected as the base **monitoring** platform, which incorporates agents (exporters, in Prometheus terminology) for a large number of monitoring targets. We have also decided to use Skydive for monitoring network infrastructures. During the first year, we have focused on monitoring the infrastructure of the cloud testbed. For this reason, we have implemented or integrated the following exporters that allow Prometheus to scrape monitoring information from VMs, from the physical nodes and from the network infrastructure:

- *Node Exporter*. We use this exporter provided by Prometheus to monitor (CPU, memory, etc) the VMs created by the *Orchestrator*.
- *IPMI Exporter*. We have implemented and deployed the *IPMI exporter* on the physical nodes of the cloud testbed (see D6.2[17] for the nodes' specification), which uses the IPMI interface to retrieve monitoring information from the nodes. We are using it to obtain power consumption metrics.
- *Skydive Exporter*. We have installed the Skydive analyzer on the cloud testbed. We have implemented a Skydive exporter to feed Prometheus with the metrics provided by Skydive.

The objective of the **predictive deployment refactoring** is to refactor or adapt the deployment model of an application at runtime in order to prevent the violation of the performance goals of the application. We have put three mechanisms in place:

- Deployment Refactoring. Analysing the performance of the deployment, the *Deployment Refactorer* will suggest a deployment alternative, using a machine learning model, which is currently being built. The design-time part of the component has been finalized. It also provides a rule-based refactoring support that enables codifying the refactoring decisions using declarative rules.
- Vertical Scalability. The *Node Manager* manages the vertical scalability of the controlled containers using a control theoretic approach.
- Deployment Option Discovery. The new deployment choices are discovered and used to improve the deployment model at runtime. A preliminary support for semantic reasoning based discovery has been implemented.

On the second year, we will be focusing on adding more functionalities and integrating these developments with the rest of the architecture and the testbeds:

- xOpera will add support for TOSCA workflows and integrate authentication/security features into the deployment phase;
- ALDE will support Torque as underlying workload manager and add a security layer;
- an HPC exporter will be implemented to provide monitoring information to the Prometheus server from the executions of applications on the HPC testbed;
- additional metrics will be provided by the Skydive exporter;
- complete the runtime part of the Deployment Refactorer
- implement the pattern-based discovery of deployment options;
- improve the integration of the NodeManager with Kubernetes and possibly extend to other orchestrators.

References

1. Esfahani, Naeem, Ahmed Elkhodary, and Sam Malek. "A learning-based framework for engineering feature-oriented self-adaptive software systems." IEEE transactions on software engineering 39.11 (2013): 1467-1493.
2. Yadwadkar, Neeraja J., et al. "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach." Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017.
3. Klimovic, Ana, Heiner Litz, and Christos Kozyrakis. "Selecta: heterogeneous cloud storage configuration for data analytics." 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18). 2018.
4. Trubiani, Catia, et al. "Exploiting load testing and profiling for Performance Antipattern Detection." Information and Software Technology 95 (2018): 329-345.
5. D2.1 Requirements, KPIs, evaluation plan and architecture. SODALITE Technical Deliverable 2019.
6. Krupitzer, Christian, et al. "A survey on engineering approaches for self-adaptive systems." *Pervasive and Mobile Computing* 17 (2015): 184-206.
7. Hallsteinsen, Svein, et al. "Dynamic software product lines." *Computer* 41.4 (2008): 93-95.
8. Chhetri, Mohan Baruwal, et al. "Exploiting Heterogeneity for Opportunistic Resource Scaling in Cloud-hosted Applications." *IEEE Transactions on Services Computing* (2019).
9. Aleti, Aldeida, et al. "Software architecture optimization methods: A systematic literature review." *IEEE Transactions on Software Engineering* 39.5 (2012): 658-683.
10. Malek, Sam, Nenad Medvidovic, and Marija Mikic-Rakic. "An extensible framework for improving a distributed software system's deployment architecture." *IEEE Transactions on Software Engineering* 38.1 (2011): 73-100.
11. Frey, Sören, Florian Fittkau, and Wilhelm Hasselbring. "Search-based genetic optimization for deployment and reconfiguration of software in the cloud." *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013.
12. Andrikopoulos, Vasilios, et al. "Optimal distribution of applications in the cloud." *International Conference on Advanced Information Systems Engineering*. Springer, Cham, 2014.
13. Al-Dhuraibi, Yahya, et al. "Elasticity in cloud computing: state of the art and research challenges." *IEEE Transactions on Services Computing* 11.2 (2017): 430-447.
14. D3.1 - First version of ontologies and semantic repository. SODALITE Technical Deliverable 2020.
15. D4.1 - IaC Management - initial version. SODALITE Technical Deliverable 2019.
16. D6.1 - SODALITE platform and use cases implementation plan. SODALITE Technical Deliverable 2020.
17. D6.2 - Initial implementation and evaluation of the SODALITE platform and use cases. SODALITE Technical Deliverable 2020.
18. D3.3 - Initial prototype of application and infrastructure performance models SODALITE Technical Deliverable 2020.
19. Gandhi, Anshul, et al. "Adaptive, model-driven autoscaling for cloud applications." 11th International Conference on Autonomic Computing ({ICAC} 14). 2014.

20. Nikravesh, Ali Yadavar, Samuel A. Ajila, and Chung-Horng Lung. "Towards an autonomic auto-scaling prediction system for cloud resource provisioning." 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE, 2015.
21. Hu, Ye, et al. "Resource provisioning for cloud computing." Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research. 2009.
22. Beloglazov, Anton, Jemal Abawajy, and Rajkumar Buyya. "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing." Future generation computer systems 28.5 (2012): 755-768.
23. Lakew, Ewnetu Bayuh, et al. "Kpi-agnostic control for fine-grained vertical elasticity." 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). IEEE, 2017.
24. Kindratenko, Volodymyr V., et al. "GPU clusters for high-performance computing." 2009 IEEE International Conference on Cluster Computing and Workshops. IEEE, 2009.
25. Xiao, Wencong, et al. "Gandiva: Introspective cluster scheduling for deep learning." 13th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018.
26. Lu, Zongqing, et al. "Augur: Modeling the resource requirements of ConvNets on mobile devices." IEEE Transactions on Mobile Computing (2019).
27. Liaqat, Misbah, et al. "Federated cloud resource management: Review and discussion." Journal of Network and Computer Applications 77 (2017): 87-105.
28. Chauhan, Sameer Singh, et al. "Brokering in interconnected cloud computing environments: A survey." Journal of Parallel and Distributed Computing 133 (2019): 193-209.
29. Zarrin, Javad, Rui L. Aguiar, and Joao Paulo Barraca. "Resource discovery for distributed computing systems: A comprehensive survey." Journal of parallel and distributed computing 113 (2018): 127-166.
30. Yu, Chin Ting, Henry CB Chan, and Daniel Wai Kei Kwong. "Discovering resources in an intercloud environment." GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, 2017.
31. Khethavath, Praveen, et al. "Introducing a distributed cloud architecture with efficient resource discovery and optimal resource allocation." 2013 IEEE Ninth World Congress on Services. IEEE, 2013.
32. Serhani, M. Adel, et al. "Self-adapting cloud services orchestration for fulfilling intensive sensory data-driven IoT workflows." Future Generation Computer Systems (2020).
33. Bellendorf, Julian, and Zoltán Ádám Mann. "Cloud topology and orchestration using TOSCA: A systematic literature review." European Conference on Service-Oriented and Cloud Computing. Springer, Cham, 2018.
34. Brogi, Antonio, and Jacopo Soldani. "Finding available services in TOSCA-compliant clouds." Science of Computer Programming 115 (2016): 177-198.
35. Soldani, Jacopo, et al. "ToscaMart: A method for adapting and reusing cloud applications." Journal of Systems and Software 113 (2016): 395-406.
36. Brogi, Antonio, et al. "Orchestrating incomplete TOSCA applications with Docker." Science of Computer Programming 166 (2018): 194-213.

37. Antequera, Ronny Bazan, et al. "Recommending heterogeneous resources for science gateway applications based on custom templates composition." Future Generation Computer Systems 100 (2019): 281-297.
38. Kritikos, Kyriakos, et al. "Evolving Adaptation Rules at Runtime for Multi-cloud Applications." International Conference on Cloud Computing and Services Science. Springer, Cham, 2019.
39. Al-Sayed, Mustafa M., Hesham A. Hassan, and Fatma A. Omara. "An intelligent cloud service discovery framework." Future Generation Computer Systems 106 (2020): 438-466.
40. Ben Djemaa, Raoudha, Hajer Nabli, and Ikram Amous Ben Amor. "Enhanced semantic similarity measure based on two-level retrieval model." Concurrency and Computation: Practice and Experience 31.15 (2019): e5135.
41. Di Modica, Giuseppe, and Orazio Tomarchio. "Matchmaking semantic security policies in heterogeneous clouds." Future Generation Computer Systems 55 (2016): 176-185.
42. Kochovski, Petar, Pavel D. Drobintsev, and Vlado Stankovski. "Formal Quality of Service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method." Information and Software Technology 109 (2019): 14-25.
43. Amedro, Brian, et al. "An efficient framework for running applications on clusters, grids, and clouds." Cloud Computing. Springer, London, 2010. 163-178.
44. Kim, Hyunjoo, et al. "Autonomic management of application workflows on hybrid computing infrastructure." Scientific Programming 19.2-3 (2011): 75-89.
45. Mateescu, Gabriel, Wolfgang Gentzsch, and Calvin J. Ribbens. "Hybrid computing—where HPC meets grid and cloud computing." Future Generation Computer Systems 27.5 (2011): 440-453.
46. Barika, Mutaz, et al. "Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions." ACM Computing Surveys (CSUR) 52.5 (2019): 1-41.
47. Taherizadeh, Salman, Vlado Stankovski, and Marko Grobelnik. "A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers." Sensors 18.9 (2018): 2938.
48. Sotiriadis, Stelios, et al. "From meta-computing to interoperable infrastructures: A review of meta-schedulers for HPC, grid and cloud." 2012 IEEE 26th International Conference on Advanced Information Networking and Applications. IEEE, 2012.
49. Mandal, Nandita, et al. "Integrating existing scientific workflow systems: the Kepler/Pegasus example." Proceedings of the 2nd workshop on Workflows in support of large-scale science. 2007.
50. Qasha, Rawaa, Jacek Cala, and Paul Watson. "Towards automated workflow deployment in the cloud using tosca." 2015 IEEE 8th International Conference on Cloud Computing. IEEE, 2015.
51. Qasha, Rawaa, Jacek Cała, and Paul Watson. "A framework for scientific workflow reproducibility in the cloud." 2016 IEEE 12th International Conference on e-Science (e-Science). IEEE, 2016.
52. Carlini, Emanuele, et al. "Basmati: Cloud brokerage across borders for mobile users and applications." European Conference on Service-Oriented and Cloud Computing. Springer, Cham, 2017.

53. Rossini, A., Kritikos, K., Nikolov, N., Domaschka, J., Griesinger, F., Seybold, D., ... & Achilleos, A. (2017). The cloud application modelling and execution language (CAMEL).
54. Ferry, Nicolas, et al. "CloudMF: applying MDE to tame the complexity of managing multi-cloud applications." 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. IEEE, 2014.
55. Bergmayr, Alexander, et al. "A systematic review of cloud modeling languages." ACM Computing Surveys (CSUR) 51.1 (2018): 1-38.
56. Binz, Tobias, et al. "OpenTOSCA–a runtime for TOSCA-based cloud applications." International Conference on Service-Oriented Computing. Springer, Berlin, Heidelberg, 2013.
57. Brogi, Antonio & Fazzolari, Michela & Ibrahim, Ahmad & Soldani, Jacopo & Wang, Pengwei & Carrasco, Jose & Cubo, Javier & Durán, Francisco & Pimentel, Ernesto & Di Nitto, Elisabetta & D'Andria, Francesco. (2015). Adaptive management of applications across multiple clouds: The SeaClouds Approach. CLEI Electronic Journal. 18. 2-2. 10.19153/cleiej.18.1.1.
58. Štefanic, Polona, et al. "TOSCA-based SWITCH Workbench for application composition and infrastructure planning of time-critical applications." (2018).
59. Štefanič, Polona, et al. "SWITCH workbench: A novel approach for the development and deployment of time-critical microservice-based cloud-native applications." Future Generation Computer Systems 99 (2019): 197-212.
60. J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the HPC environment," in International Conference on High Perfor- mance Computing. Springer, 2015, pp. 506–513.
61. M. de Bayser and R. Cerqueira, "Integrating MPI with Docker for HPC," in Cloud Engineering (IC2E), 2017 IEEE International Conference on. IEEE, 2017, pp. 259–265.
62. O. Rudyy, M. Garcia-Gasulla, F. Mantovani, A. Santiago, R. Sirvent and M. Vázquez, Containers in HPC: A Scalability and Portability Study in Production Biological Simulations,; 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 2019, pp. 567-577.
63. Benedičič, Lucas & Cruz, Felipe & Madonna, Alberto & Mariotti, Kean. (2019). Sarus: Highly Scalable Docker Containers for HPC Systems. 10.1007/978-3-030-34356-9_5.
64. Fowler, Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.
65. Arcelli, Davide, Vittorio Cortellessa, and Daniele Di Pompeo. "Performance-driven software model refactoring." Information and Software Technology 95 (2018): 366-397.
66. Shastri, Supreeth, Melissa Wasserman, and Vijay Chidambaram. "GDPR Anti-Patterns: How Design and Operation of Modern Cloud-scale Systems Conflict with GDPR." arXiv preprint arXiv:1911.00498 (2019).
67. Tuma, Katja, et al. "Inspection guidelines to identify security design flaws." Proceedings of the 13th European Conference on Software Architecture-Volume 2. 2019.
68. Fatema, Kaniz & Emeakaroha, Vincent & Healy, Philip & Morrison, John & Lynn, Theodore. (2014). A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives. Journal of Parallel and Distributed Computing. 74. 10.1016/j.jpdc.2014.06.007.

69. Schonenberg, Helen, et al. "Process flexibility: A survey of contemporary approaches." Advances in enterprise engineering I. Springer, Berlin, Heidelberg, 2008. 16-30.
70. D2.4 - Guidelines for Contributors to the SODALITE Framework. SODALITE Technical Deliverable 2020

A Appendix

This appendix section will include a list of metrics that are currently being monitored in our system. It will be divided into three subsections corresponding to the three types of exporters currently utilised: node exporter, IPMI exporter and Skydive exporter.

A.1 Node exporter

Node exporter is developed by the Prometheus team and it covers mostly all relevant hardware aspects, such as CPU, I/O and network utilization, among others. In its repository[68] there is a list of all the included collectors. Each one of those collectors will expose a series of metrics, and listed in the below included table there is a subset of the most relevant to SODALITE.

Metric	Description
node_cpu{cpu=x,mode=y}	This metric represents the number of seconds that the CPU "x" has spent in mode "y". These modes can be idle, system usage, I/O, CPU stolen by another VM, etc.. This is perhaps one of the most important metrics as it gives vast and valuable information about how the CPU is being used by the VM.
process_cpu_seconds_total	Another variable that can be used for measuring CPU usage, as it joins the total number of seconds spent by the CPU in system and user tasks.
go_memstats_alloc_bytes	Total number of allocated bytes. Gives a general overview of how much memory is being used.
go_memstats_alloc_bytes_total	Only increasing memory counter. Useful for finding the maximum amount of memory consumed by an application.
node_network_receive_bytes	Received bytes. There are similar variables for sent, dropped, etc.
node_disk_read_time_ms	Total number of milliseconds spent reading from memory. Similar variables exist for write times.
node_load1	CPU load average during the last minute. Also available for the last 15 and 5 minutes.

[68] Node exporter GitHub repository containing all collectors: https://github.com/prometheus/node_exporter

A.2 IPMI exporter

This exporter was created only with the idea of exposing the physical nodes power measurement given by an IPMI command that utilises a physical sensor to obtain it, but it could be modified to include more metrics given by command outputs or variations of the already measured power consumption, such as average consumption for the last minutes.

Metric	Description
power_consumption	Gives the power consumption measurement in Watts.

A.3 Skydive exporter

The Skydive exporter forwards network metrics from Skydive to the Prometheus server used in SODALITE. This allows to combine network information with other resource usage gathered from other sources. The first implementation provides the byte transfer counts, but more metrics will be improved during the rest of the project.

Metric	Description
skydive_network_connection_total_bytes	Total number of bytes transferred between 2 network endpoints, qualified by source address, source port, target address, target port.

B Appendix

To achieve the functionalities of the Runtime Layer, it makes use of several Open Source solutions. These are described below.

B.1 xOpera

xOpera (https://github.com/xlab-si/xopera-opera) is a lightweight orchestrator compliant with the TOSCA simple YAML Profile v1.2.

xOpera allows for simple and straightforward orchestration of containerized computing environments with Docker[69] containers and Ansible automation. The only prerequisites for the operation of xOpera is to have Python 3 installed and a virtual environment set up and furthermore the installation can be made even more simple by using Ansible itself to install everything needed for operation of xOpera in conjunction with OpenStack.

xOpera is open source and available on GitHub under the Apache 2.0 license. The documentation and example use cases are also available on GitHub.

B.2 Prometheus

Prometheus (https://prometheus.io/) is an open-source software developed for monitoring and alerting. It “scrapes” metrics and their values from exporters, pieces of software created for providing information related to a specific field, topic or subject. This information is collected using an HTTP pull model at specified intervals and is logged in a time-series database, allowing easy representation and displaying using its web platform or the well-known tool Grafana. Thanks to a wide set of already existing exporters and the simplicity of their functioning it is fairly simple to find or develop one to cover the required metrics.

B.3 Skydive

Skydive (http://skydive.network) is a real-time network topology and protocol analyzer that provides detailed network topology and performance information. Skydive agents collect topology information and flows and forward them to a central agent for further analysis.

Network topology support of Skydive allows one to view relationships between hosts (both physical and virtual), containers (network namespace), and network entities (device, bridge, Veth, TUN, Macvlan, etc).

Skydive is extensible and allows the development of probes for new kinds of environments. Probes already exist for Kubernetes, Istio, NSM, OVN. For example, the Kubernetes probe provides information on clusters, namespaces, nodes, pods, containers, services, network policies, volumes, and deployments.

The extensible Skydive framework allows a developer to implement probes for new kinds of topology entities and flow types.

Detailed documentation of Skydive can be found at http://skydive.network/documentation. Additional insights can be gleaned by looking through source code, especially the configuration possibilities defined in the skydive.yml.default file.

See https://github.com/skydive-project/skydive/blob/master/etc/skydive.yml.default.

[69] https://www.docker.com/