Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

Application deployment and dynamic runtime - intermediate version

D5.2

ATOS

31.1.2021

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

Deliverable data			
Deliverable	D5.2 - Application deployment and dynamic runtime - intermediate version		
Authors	Jesús Gorroñogoitia (Atos), Jorge Fernández Fabeiro (Atos), Lucas Pelegrin Caparrós (Atos), Indika Kumara (JADS/UVT), Dragan Radolović (XLAB), Nejc Bat (XLAB), Kamil Tokmakov (USTUTT), Kalman Meth (IBM), Giovanni Quattrocchi (POLIMI), Paul Mundt (ADPT)		
Reviewers	Dennis Hoppe (USTUTT), Zoe Vasileiou (CERTH)		
Dissemination level	Public		
History of changes	Name	Change	Date
	Atos	Outline created	18.10.2020
	Atos	Section 2. Sections 4.5, 4.8, 4.9, 4.1, 5.2	8.01.2021
	ALL	Partners' contribution to sections 4, 5	13.01.2021
	ALL	Partners' contribution to sections 1, 3, 4, 5, 6	15.01.2021
	ALL	Final contributions	18.01.2021
	Atos	Version for internal review	18.01.2021
	ALL	Addressing internal review	25.01.2021
	Atos, XLAB	Final version	31.01.2021

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

List of tables

List Of Tables

Executive Summary

This deliverable reports on the status of the development, at M24, of the SODALITE *Runtime Layer* and the integration of its components with the rest of the SODALITE platform. This is the second of three deliverables in this series, to be released annually during the project lifetime. This deliverable complements D3.1[1] and D4.2[2], 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 *Runtime Layer* offers three main features: (1) the orchestration of the deployment of applications on heterogeneous infrastructures, (2) the collection of runtime monitoring information, and (3) the adaptation of applications for performance improvement.

The main focus of the deliverable is to present the new features that have been incorporated into the *Runtime Layer* since the last release in M12 D5.1[3] , with the focus on the innovation they bring, their internal architecture within the *Runtime Layer*, the main functional aspects they offer, the current status of their development, the analysis of their QA assessment, and the planned developments for next releases in M30 and M36.

- ***The Orchestration Layer***: The M24 *Runtime Layer* release supports the deployment of orchestrated, containerized applications in Cloud infrastructures managed by AWS, OpenStack or Kubernetes, as well as on HPC clusters managed by SLURM, TORQUE/PBS Pro schedulers. The access to the Orchestration layer is protected by the adoption of the SODALITE *IAM Authentication*. **Orchestration has been extended to support multiplatform, hybrid data management**, by adopting stream-driven, data transfer technology adopted from the RADON project[4], as a result of our mutual collaboration. This **IaC data management feature will be extended to support HPC environments** in SODALITE.
- ***The monitoring layer***: has been significantly redesigned to support **dynamic monitoring of targets on both Cloud infrastructures, such as OpenStack, and on HPC clusters**, such as those managed by TORQUE/PBS Pro and SLURM schedulers, on Edge and also on their interconnecting network. Moreover, Monitoring supports the **broadcasting of alert notifications** to subscribers, such as those in Refactoring, upon the detection of QoS violations.
- ***The Refactoring Layer***: **Deployment refactorer** was integrated with the SODALITE monitoring infrastructure to support the adaptation of the deployment topology of an application in response to monitoring data and alerts. The **machine learning (ML) pipeline for building ML models for predicting the performance of many deployment alternatives** of an application has been implemented and evaluated. These predictiction models enable the **selection and switching among deployment variants** at runtime. The policy-based deployment adaptation was improved to support the various event-based adaptation use cases. The dynamic discovery of nodes has been improved to support node (TOSCA) policies. Node Manager implementation and evaluation was completed. Node Manager provides **runtime resource management at three levels: cluster-level smart load balancing, machine-level supervision of resource contention, and container-level control theoretical vertical scalability**.

Partial integration of the *Runtime Layer* components, mostly for orchestration drivers, some monitoring and refactoring components have been completed in M24 release.

Next steps towards the release of the final version of the *Runtime Layer* (M30, M36) are focusing on the complete integration of the entire *Runtime Layer,* the support of deployment to additional target infrastructures such as OpenFaaS and Google Cloud, the automation of dynamic monitoring configuration upon deployment, the complete implementation of the alerting management, the

implementation of specialized monitoring dashboards, the support of all redeployment adaptation scenarios and several improvements in refactoring features.

Glossary

Acronym	Explanation
AAI	Authentication and Authorization Infrastructure
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
AWS	Amazon Web Services
CI/CD	Continuous Integration / Continuous Deployment
CPU	Central Processing Unit
CSAR	Cloud Service Archive
DI	Deployment Instance
DTR	Decision Tree Regression
ECA	Event Condition Action
FaaS	Function as a Service
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
GB	Gigabyte
GCS	Google Cloud Storage
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IAM	Identity and Access Management
ID	Identifier

IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IoT	Internet of Things
IPMI	Intelligent Platform Management Interface
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
K8S	Kubernetes
LRE	Lightweight Runtime Environment
ML	Machine Learning
MLOps	Machine Learning operations
MLP-NN	Multilayer Perceptron Neural Network
MQTT	Message Queue Telemetry Transport
M<X>	Month <X>
N/A	Not Available
QA	Quality Assurance
QE	Quality Expert The equivalent process from ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes: Infrastructure management and Configuration management processes
QoS	Quality of Service
QoE	Quality of Experience
Ops	Operations
OS	Operating System
PBS	Portable Batch System
PoC	Proof of Concept
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
RFR	Random Forest Regression
S3	Amazon Simple Storage Service
SLA	Service Level Agreement
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SRM	Storage Resource Manager
SSH	Secure SHell
TB	TOSCA Blueprint
TOSCA	Topology and Orchestration Specification for Cloud Applications
UC	(UML) Use Case
UI	User Interface
UML	Unified Modeling Language
VM	Virtual Machine
WebDAV	Web Distributed Authoring and Versioning
WP<X>	Work Package <X>
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language
Y<X>	Year <X>

1 Introduction

Modern software coexists with heterogeneous, software-defined, high-performance computing environments and resources, including cloud servers, GPUs, FPGAs, Kubernetes, FaaS, etc. Advanced applications require complex and heterogeneous deployments that match their components with the infrastructure that offers the best performance fulfilling their requirements. In this context, SODALITE aims to address this heterogeneity by providing a toolset that enable developers and infrastructure operators to achieve faster development, deployment and execution of applications on different heterogeneous infrastructures as also presented in deliverable D5.1:

- a pattern-based abstraction library with support for application, infrastructure and performance and operation abstractions;
- a metamodel 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.

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[5] blueprint of an application and deploys each application component on the specified targets, which may be heterogeneous.
- **Collecting runtime monitoring information at different levels**: application, runtime environment and infrastructure. With this information, it is possible to analyse the application's performance, and apply corrective strategies upon the detection of QoE violations.
- **Enabling adaptation of the application to improve its performance**. In order to fulfil the application's performance goals, different mechanisms could be applied at runtime.

1.1 Deliverable goal

This deliverable presents the functional specification and the technical implementation of the intermediate release (M24) of the SODALITE *Runtime Layer*, with the focus on the main features that have been incorporated to this layer since the last release at M12, reported in D5.1. For each new feature, this document emphasizes its innovative aspect, architecture, functional characteristics, status of its current implementation and planned work for the next development phases, namely M30 and M36.

This document also reports on the progress achieved on existing features of the *Runtime Layer* reported in D5.1 and on the adoption of the automated SODALITE development methodology and toolset, which relies on the usage of public code repositories, CI/CD pipelines and software QA assessment.

This deliverable has been developed in parallel and coherently to WP2, WP5 and WP6 deliverables D2.2, D4.2, D6.3, D6.6 and to the work developed in WP3 as part of the second project year.

Throughout the document, we are using the terms Application Ops Experts (AOE), Resource Experts (RE) and Quality Experts (QE), as described in D5.1.

1.2 Structure of the document

The document is structured as follows: Section 2 gives an updated overview of the Runtime Layer architecture, highlighting the main changes introduced since the last reported version of D5.1. Section 3 introduces the development DevOps toolset and methodology adopted for the building and delivering of the *Runtime layer* as well as the locations of the source code repositories of its different components. It also describes the adopted software QA process and tool set, and pointers to QA assessment results for the components of the *Runtime layer*. Section 4 provides a detailed description of the main features incorporated to the *Runtime layer* since the last release like presented in D5.1, with the emphasis on their innovation, current development status and plans for next releases. Section 5 outlines the development changes for those features of the *Runtime layer* that were already released in the M12 release. Section 6 summarizes the feature development plan for next releases (M30, M36). Section 7 concludes the document, summarising the current status of the *Runtime Layer* and the next steps towards the final release.

2 Runtime Layer Architecture

The architecture of the *Runtime Layer* was introduced in [D5.1]. This section describes the changes that have been adopted in this architecture since then.

As explained in D5.1, this layer is composed of three main building blocks (see Figure 1), corresponding to each of the layer 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 current architecture has been extended to include the drivers supported and integrated within the orchestrator have been included:

- *OS driver*: the orchestrator uses this driver to interface with the shell of the VM in Cloud infrastructures such as OpenStack or AWS,
- *TORQUE driver*: the orchestrator uses this driver to interface with the HPC TORQUE scheduler in order to support the deployment of batch jobs in HPC clusters. Other similar drivers for HPC schedulers could be developed based on this one (e.g. SLURM, TORQUE/PBS Pro),
- *K8S driver*: this driver enables the orchestrator to interface Kubernetes to support the deployment of container based application components in pods,
- *OpenFaaS driver*: to support the deployment of serverless functions into OpenFaaS - to be supported in Y3 of the project,
- *AWS driver*: to support the deployment into AWS.

It has also been extended to introduce the new required interfaces for Identity and Access Management (IAM) (e.g. *IAMIntrospectionAPI*) and for the retrieval of deployment secrets (e.g. *SecretVaultAPI*).

The ***Monitoring*** block is composed of a monitoring server, a dashboard and a set of probes that retrieve metrics from the different monitoring targets: VMs, HPC schedulers and nodes, runtime environment, application components, etc. This block includes new components not previously included in the former architecture:

- *Monitoring Dashboard*: this frontend provides visualization of monitoring information (using different graphs and gauges) obtained for selected target application components and execution environments. This dashboard offers individual views for each user's deployed application, which are accessible through the *SODALITE IDE*. This is the standard access point for Application Ops Expert (AOEs) to browse their apps monitoring data. Dashboard uses the *MonitoringAPI* REST interfaces to query monitoring data.

- *Monitoring Alert Manager*: manages defined alerting rules to trigger notifications (and the associated monitoring data) to subscribers (e.g. orchestration and refactoring components) when the rule condition holds for the target monitoring metrics. This is the standard way for some SODALITE platform components to be notified with monitoring data. The *AlertingAPI* REST interface is used by the *Orchestrator, Node Manager* and *Deployment Refactorer* components to subscribe themselves to concrete alerts.

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 internal interactions among these block components have been made explicit in the architecture, through the REST API exposed by each component.

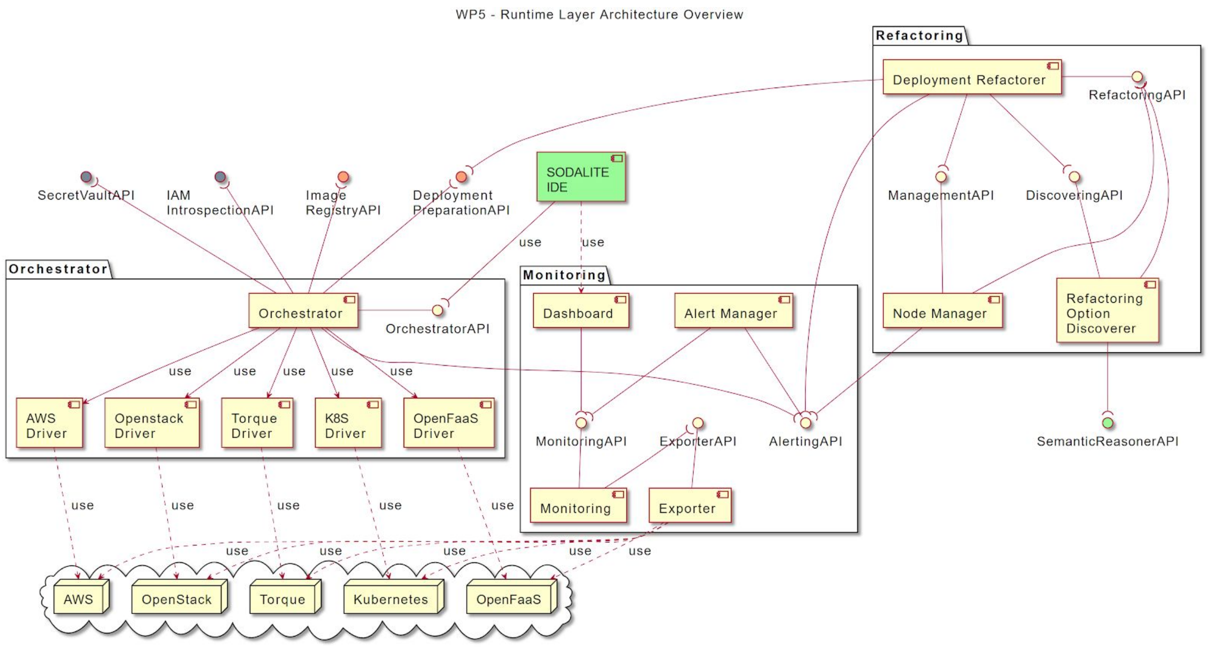


Figure 1 - Runtime Layer Architecture

As shown in the general SODALITE architecture [D5.1], the Runtime Layer components need to interact with other components in the SODALITE platform to fulfill their objectives. The following describes the changes in the interactions of the *Runtime layer* with components of the other layers:

- **IDE** [D3.1]. The IaC blueprint is sent to the *Orchestrator* via the *IDE*, where the AOE 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 from the *IDE* for each deployed application by accessing its associated dashboard in the *Monitoring Dashboard*.
- **Deployment Preparation** [D4.2]. 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** [D3.1]. 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*.

3 Project development toolset

This section introduces the methods and toolsets adopted within SODALITE to produce and deliver a high quality of code. Details on these aspects are reported in D1.4[6]. Same content is already introduced in Section 3 of D4.2 but replicated and adapted here for the sake of self-consistency.

3.1 GitHub Repositories

SODALITE chose *GitHub* as its primary publicly available development version control system. *GitHub* acts as the open source community not only as a version control system but also as a developers collaboration platform by offering many available tools, such as an issue tracker, a project management tool, a wiki etc., and further introducing means to intensify collaboration.

All the available open-sourced code produced in SODALITE can be accessed through SODALITE's *GitHub* organization: https://github.com/SODALITE-EU.

SODALITE utilizes different project development collaboration features provided by *GitHub* such as: project boards, teams, discussions through issues, pull requests, and peer reviews of code.

3.2 Continuous Integration and Continuous Delivery CI/CD

SODALITE uses the *Jenkins*[7] to support automated testing, versioning and publishing processes for SODALITE components. To improve the quality and the automation of the CI/CD process, a convention for the development process has been set up with specific examples of usage described in detail in D6.3[8].

3.3 Software QA

SODALITE is bound to produce mostly open source code on a publicly available version control system. The SODALITE consortium also recognizes the high impact of developing excellent quality of code of its software components as a high priority task. For this reason the free and open online *SonarCloud*[9] utility is used to assess the quality of the code developed. To enable developers to deliver better code quality, *SonarCloud* shows various dashboards and enables for a streamlined integration with *GitHub*, providing developers with a good estimate of code quality even before merging the code into the master/main branch. This feature among many others is extensively used in the SODALITE CI/CD pipeline providing both the developer and the reviewer of the code with significant and important insights about the quality of developed code, as well as providing useful suggestions on how to improve the code.

All of the repositories of the SODALITE components were integrated with *SonarCloud* during the second year of the project. The main metrics collected concern the following aspects:

- The number of bugs: bugs in *SonarCloud* are identified exploiting various static analysis tools specific to the supported languages.
- The number of security vulnerabilities and hotspots. As highlighted in the *SonarCloud* manual[10], "*with a Hotspot, a security-sensitive piece of code is highlighted, but the overall application security may not be impacted. It's up to the developer to review the code to determine whether or not a fix is needed to secure the code. With a vulnerability, a problem that impacts the application's security has been discovered that needs to be fixed immediately*".
- The number of code smells.
- The code coverage defined in terms of lines of code that are exercised by automated test cases.
- The amount of replicated code.

The general goal of SODALITE with respect to these metrics is to continuously keep them under control and improve them from release to release. As for the components belonging to the *IaC Management Layer*, we expect the *Runtime Layer* metrics to show relatively high values considering that it is used by most of the other components of the platform.

IWe provide for each stable component an overview of its current status in terms of the *SonarCloud* metrics in the following sections.

3.4 Runtime Layer artifacts

The following Table 1 provides links to the GitHub repositories, SonarCloud analysis reports and Dockerized images of each of the components of the *Runtime Layer* reported in this deliverable. Details on the QA assessment results reported by SonarCloud for each component of the *Runtime Layer* and how they were addressed are reported in Sections 4 and 5.

Component	Github Repository	SonarCloud Dashboard	Image on Docker Hub
Orchestrator	https://github.com/SODALITE-EU/xopera-rest-api	https://sonarcloud.io/dashboard?id=SODALITE-EU_xopera-rest-api	1.https://hub.docker.com/r/sodaliteh2020/xopera-rest-api 2.https://hub.docker.com/r/sodaliteh2020/xopera-flask 3.https://hub.docker.com/r/sodaliteh2020/xopera-nginx
HPC Driver	https://github.com/SODALITE-EU/alde	https://sonarcloud.io/dashboard?id=SODALITE-EU_alde	N/A[11]
IaC Data Management	https://github.com/RADON-SODALITE	N/A	N/A
Monitoring	https://github.com/SODALITE-EU/monitoring-system	https://sonarcloud.io/dashboard?id=SODALITE-EU_monitoring-system	1.https://hub.docker.com/r/prom/prometheus 2.https://hub.docker.com/_/consul
Alert Manager	https://github.com/SODALITE-EU/monitoring-system	https://sonarcloud.io/dashboard?id=SODALITE-EU_monitoring-system	N/A
HPC exporter	https://github.com/SODALITE-EU/hpc-exporter	https://sonarcloud.io/dashboard?id=SODALITE-EU_hpc-exporter	N/A
Intel NCS2 exporter	https://github.com/adaptant-labs/prometheus_ncs2_exporter	https://sonarcloud.io/dashboard?id=adaptant-labs_prometheus_ncs2_exporter	https://hub.docker.com/repository/docker/adaptant/prometheus-ncs2-exporter
EdgeTPU	https://github.co	https://sonarcloud.io/d	https://hub.docker.com/reposit

exporter	m/adaptant-labs/edgetpu-exporter	ashboard?id=adaptant-labs_edgetpu-exporter	ory/docker/adaptant/edgetpu-exporter
SkyDive exporter	https://github.com/skydive-project/skydive-flow-exporter/blob/master/prom_sky_con	N/A	https://hub.docker.com/r/sodaliteh2020/prometheus-skydive-connector
IPMI exporter	https://github.com/SODALITE-EU/ipmi-exporter	https://sonarcloud.io/dashboard?id=SODALITE-EU_ipmi-exporter	N/A
Deployment Refactorer	https://github.com/SODALITE-EU/refactoring-ml	https://sonarcloud.io/dashboard?id=SODALITE-EU_rule-ml-based https://sonarcloud.io/dashboard?id=SODALITE-EU_perf-predictor-api	https://hub.docker.com/r/sodaliteh2020/rule_based_refactorer https://hub.docker.com/r/sodaliteh2020/fo_perf_predictor_api
Node Manager	https://github.com/SODALITE-EU/refactoring-ct	https://sonarcloud.io/dashboard?id=SODALITE-EU_refactoring-ct	N/A
Refactoring Option Discoverer	https://github.com/SODALITE-EU/refactoring-option-discoverer	https://sonarcloud.io/dashboard?id=SODALITE-EU_refactoring-option-discoverer	https://hub.docker.com/r/sodaliteh2020/refactoring_option_discoverer

Table 1 - Runtime Layer artifacts

4 New features developed in the second project year

This section presents the main new features included within the M24 release of the *Runtime Layer*. Some of these features were conceptually anticipated in [D5.1], despite them not delivered within the M12 release. For each feature, it is described the innovation it brings, its architecture (within the overall *Runtime Layer*), its main functional characteristics, its current implementation status, the analysis of the code quality and its planned development steps for next releases (M30 and M36). Table 2 summarizes the main new features introduced in M24 for the main *Runtime Layer* components, namely the orchestration, monitoring and refactoring components, which are further described in following subsections.

Component	Feature	Status
Orchestration	HPC Driver	HPC driver implementation for SLURM/TORQUE schedulers. Integration with the orchestrator.
	IaC Data Management	Partial implementation of TOSCA libraries that support publishing to and consuming data from local filesystem, S3, GCS and GridFTP
Monitoring	Dynamic Monitoring	Dynamic registry of monitoring exporters. Integration with the main monitoring engine. Ansible playbooks to register/deregister exporters.
	HPC Monitoring	Jobs status and consumed resources metrics for SLURM/TORQUE/PBS Pro schedulers.
	EDGE Monitoring	Exporters for Edge TPUs and Intel NCS2 Neural Compute Sticks developed and integrated. Partial implementation of others (e.g. integrated NVIDIA Tegra GPUs)
	Skydive Monitoring	Skydive-prometheus-connector developed and integrated.
	Alerting	Rule File Server and API implemented and integrated with the monitoring engine. Partial implementation of Alert Manager and integration with the monitoring engine.
Refactoring	Deployment Refactorer	Architecture design. Initial implementation of the policy-based deployment adaptation, validated in with three key scenarios from the *Vehicle IoT use case.* Implementation of the building performance models for enabling deployment configuration.
	Node Manager Refactoring	A prototype of Node Manager was implemented and evaluated on the Azure public cloud using four benchmark applications: Skyline Extraction from Snow UC, ResNet, GoogLeNet, and VGG16.
	Refactoring Option Discoverer	Resource matchmaking: logical expressions on the constraints on node properties can be used to select the nodes, nodes are discovered by matching the requirements of the source node with the capabilities provided by the candidate target nodes, resource matchmaking based on policies.

Table 2 - Runtime Layer main new features

4.1 Orchestration - HPC Driver (ALDE)

4.1.1 Innovation

ALDE (Application Lifecycle Deployment Engine) is a component of the Orchestrator which functions as a driver for HPC environments. Whenever a project requirement is in need to execute HPC workloads, *ALDE* is the component that drives to execute the operations. Since *ALDE* it is an outcome of the TANGO[12] project, it was required to adapt the code to the specific requirements of SODALITE and that has been the one main key aspect about *ALDE* that has been innovated along the year. Before the fork creation from the TANGO project repository[13], *ALDE* could deploy SINGULARITY containers in SLURM-based HPC environments already, but SODALITE required to extend the support to TORQUE-based. TORQUE is a scheduler used to orchestrate and administrate the jobs queued in HCP clusters and from now one *ALDE* supports both schedulers engines: TORQUE and SLURM.

Moreover, new auto-discovery features have since been included in *ALDE*. Since the support for different sorts of environments has increased in scope, it was required to adapt the discovery of the resources available in new HPC environments.

4.1.2 Architecture

Because *ALDE* requires an initialization setup in order to have administrative permissions to execute workload in the HCP environments. An *ALDE* server instance is being deployed as is shown in Figure 2. Once the initial configuration is done *ALDE* is fully operative using its REST API interface.

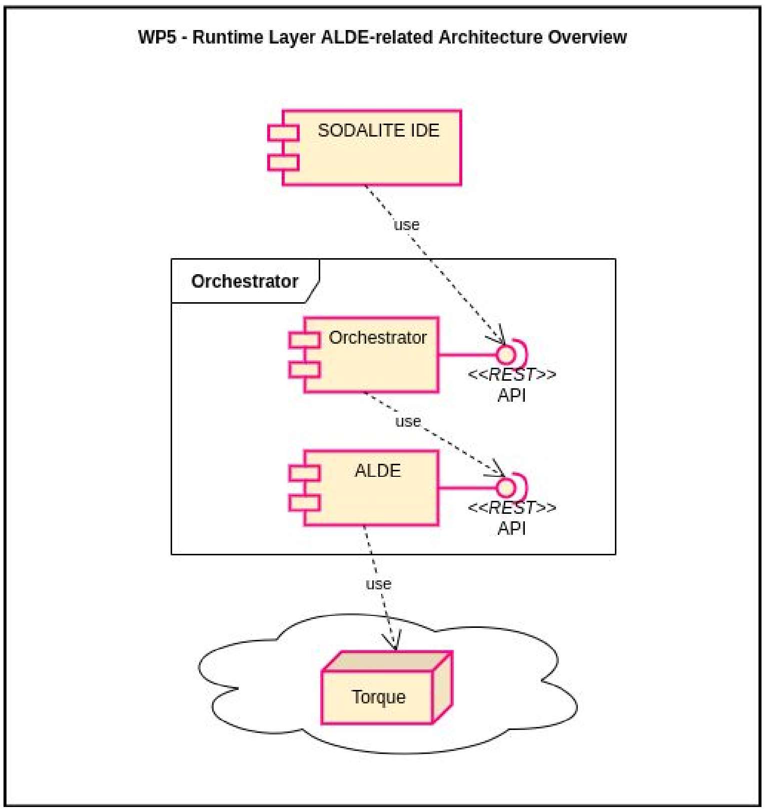


Figure 2 - ALDE component within the orchestrator architecture

4.1.3 Features

ALDE capabilities or features haven't changed in scope nor its core architecture inside the *Orchestrator*. As it was written in the D5.1 documentation version, the main *ALDE* capability is to deploy and execute applications onto the supported HPC workload environments. Based on this point, functionalities of the *ALDE* software have been extended to support TORQUE schedulers and are presented in the following status section. Nonetheless, as an outcome of this implementation, *ALDE* extended a feature that wasn't present before. New auto-discovery features have been included in order to adapt to the new upcoming challenges considering multiple HPC Schedulers setups.
At the beginning of the *ALDE* setup. It executes a resource inventory procedure to identify the accessible resources available. This information could be required for the scheduler as a preventive measure to not overbook the resource capacity of the HPC environment during the scheduling time.

4.1.4 Status

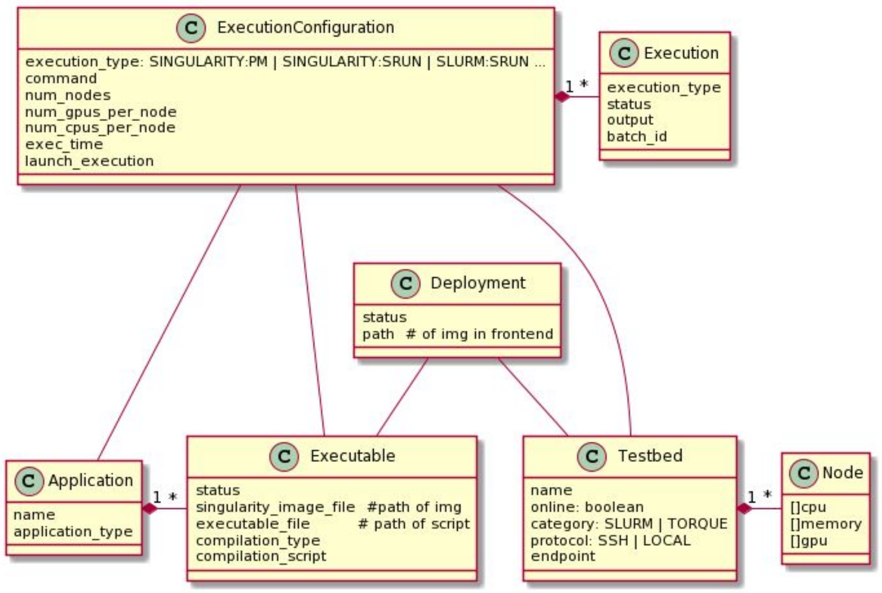


Figure 3 - ALDE data model structure

To adapt *ALDE* to the features required in SODALITE it was needed to modify *ALDE* data model (Figure 3) and include the following changes in the software:

Testbed: An HPC testbed, managed by SLURM or TORQUE workload manager. By enabling this, *ALDE* can select the functionality to interface the administration of the workload to the Testbed environment.

Node: Definition of nodes in the testbed. *ALDE* gets the nodes from the workload manager and can connect to the nodes to get additional information. It was extended to auto-discover new resources such as new types of GPU cards and processors.

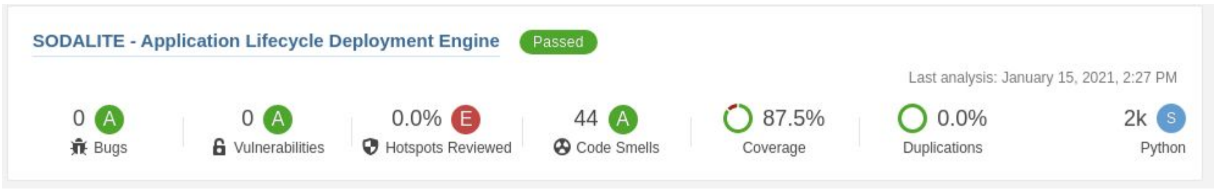
Deployment: Creating a deployment triggers deployment of an Executable (i.e., the SINGULARITY image in the *ALDE* server) to the frontend node. The result is that the image in the *ALDE* server is copied to the frontend node. As with the Executable, the Deployment can be created posting all the information, so the file is not copied to the frontend node, but the request already contains the path in the node where an image is stored. Moreover, the interface of the deployment is adapted based on the Testbed (HPC environment) where applications are being executed, that is, to extend the functionality of the workload administration based on the Testbed category (TORQUE or SLURM).

On the other hand, *ALDE* project has been integrated with the CI (Continuous Integration) methodology to automatically integrate new versions to the *DockerHub* repository whenever newer versions are available. By using *Jenkins*, a script located inside the *ALDE* repository builds, tests and publishes a *Docker* version whenever a release is being made in the *ALDE* repository. The innovation is to feature container-based deployments of *ALDE* and to shorten the development cycle of the component by ramping up the integration software stage.

Lastly, a *xOpera* blueprint has been made to interface the *Orchestrator* with *ALDE*. The blueprint can deploy new applications setups easily by including the required configuration steps when a new application setup is made in *ALDE*.

4.1.5 Code Quality

Code quality report for the *ALDE* code is shown in Figure 4. This report shows some issues that will be investigated and removed in the next release.


SODALITE - Application Lifecycle Deployment Engine Passed

Last analysis: January 15, 2021, 2:27 PM

Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Python
0 A	0 A	0.0% E	44 A	87.5%	0.0%	2k S

Figure 4 - Code quality report for ALDE

4.1.6 Next steps

ALDE and the *Orchestrator* are fully integrated via a TOSCA blueprint. In a PoC performed the blueprint was executed with the *Orchestrator* and it had proved to deploy a *SINGULARITY* container to the HPC environment, *ALDE* did manage to schedule the workload until its completeness.

4.2 IaC Data Management

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

4.2.1 Innovation

From the requirement elicitation done during Y2[14], there is a need for a portable and scalable way to connect loosely coupled application components deployed on various platforms and perform data movement between them. As an example, consider a production HPC infrastructure, where strict firewall rules are applied and a number of available ingress endpoints for data management is often limited to GridFTP file transfer protocol. Moreover, egress connectivity from the infrastructure to the Internet is often blocked for security reasons, hence directly obtaining data from external repositories, data services or cloud storages is not possible. For such cases, there exist state-of-the-art technologies and tools that provide adaptivity between various interfaces for data management.

Scientific and research communities have developed services for unified data management as a consequence of a federation of infrastructure and service providers that offer different cloud storage technologies and different file transfer protocols. Services, such as *Globus*[15], *Onedata*[16] and *DynaFed*[17], provide a unified data access for federated providers as well as external providers, e.g. *Amazon S3* and *Microsoft Azure*. While *Onedata* and *DynaFed* are open source and freely extendable, *Globus'* support for connecting with Cloud storage backends is a premium feature.

Furthermore, there has been a development for the multiprotocol data management services. *FTS3*[18] is a data movement service for data transfers between various storage systems. It is based on *GFAL2*[19] multiprotocol data management library and any combination of the protocols, such as GridFTP, SRM, S3, HTTP(S), WebDAV and XrootD, can be used. This allows to schedule a file transfer between cloud storages, Grid and HPC systems. *Rucio*[20] is a framework for data management across globally distributed locations and across heterogeneous data centers. It unifies different storage technologies into a single federated entity and utilises FTS3 as a middleware for multiprotocol connections.

With the respect to data management for multi-cloud storage, rclone[21] provides a command line tool for synchronisation of files across different cloud storage providers, such as S3, Dropbox, GCS as well as different protocols, such as HTTP, (S)FTP, WebDAV. *Zenko*[22] is a multi-cloud storage controller, which exposes an S3-compatible API for data management across private and public clouds, such as S3, Azure, GCS and Ceph. SODA foundation[23] provides a cloud vendor agnostic data management for hybrid cloud, intercloud or intracloud and also exposes S3-compatible API for management of various cloud providers, such as Azure, GCS, S3, as well as Ceph.

With the emergence of Edge computing, IoT and serverless platforms, data processing shifted towards handling data streams, and thus several data stream platforms and data flow services have been developed. *MQTT*[24] is an OASIS standard messaging protocol for the IoT. It is designed as a lightweight publish/subscribe messaging transport for connecting remote devices with constrained resources. *Mosquitto*[25] and *HiveMQ*[26] are some of the implementations of *MQTT*. *Apache Kafka*[27] is a distributed stream processing platform that allows to publish/subscribe to and persist streams of events, including continuous import/export of data from other systems (such as S3, Azure, FTP, GCS, SQL, HDFS, etc.) or real-time applications via *Kafka Connect*. *Fledge*[28] is an IoT platform, and it acts as a gateway between edge devices and cloud storage systems. It currently provides plugins for some platforms and protocols, such as Apache Kafka, GCS, HTTP, EdgeX. *MinIO*[29] is a Kubernetes native, S3-compatible object storage. It provides an alternative version that can be deployed on edge. Data flow services, such as *Apache NiFi*[30] and *StreamSets*[31], also provide rich support for data management between various cloud storage vendors (S3, GCS, Azure, etc.) and streaming platforms (*Kafka, MQTT*).

From the current SOTA, it can be seen that the current data management services found in scientific communities mostly focus on HPC and cloud storage platforms and do not cover edge, IoT and serverless platforms. Inversely, it is true that edge and serverless platforms target stream and cloud platforms, but do not target HPC platforms. Therefore, in order to satisfy the requirements of use case owners and to meet the SODALITE objectives for supporting heterogeneous infrastructures, data management for mentioned platforms shall be provided. As part of the collaboration with RADON[32] project, we study the feasibility of using data pipelines as components to unify data management between various heterogeneous platforms. **As an**

innovation, SODALITE extends RADON's TOSCA and IaC libraries for data pipeline management, which currently target multi-cloud storage and serverless platforms, with GridFTP support – a common file transfer protocol used in HPC. This enables an interoperability between HPC, Cloud storage types and data streams.

4.2.2 Architecture

The RADON project has developed a set of standard TOSCA libraries[33] for lifecycle management of data pipelines, which is inline with IaC-based orchestration in SODALITE. The concept of data pipeline allows composition of application components (e.g. microservices, serverless functions or self-contained components) as independently deployable and scalable pipeline tasks with the data movement and possible data transformation between the components[34]. As an underlying technology for data pipelines, Apache NiFi service is used. It exposes a REST API for data flow management between pipeline elements (blocks) and also provides connectors to various platforms and storage systems, such as S3, GCS, Azure, Apache Kafka, HDFS, MQTT, HTTP, (S)FTP, etc. This enables fetching data from one storage provider and pushing data to another provider as a pipeline task.

In the context of the SODALITE and RADON collaboration D7.4[35], SODALITE reuses RADON's libraries for data management and extends them to support data transfer protocols common in HPC, such as GridFTP. Figure 5 depicts an overview on data pipeline management from SODALITE's perspective. The Orchestrator exposes a REST API for deployment of a CSAR (Cloud Service Archive), which contains a TOSCA application topology description along with TOSCA node types, IaC and other dependencies. The application topology must specify pipeline blocks and connections between them, as well as specify which NiFi instance to use and whether the orchestrator must create a new instance or use the existing instance of NiFi. On the lower level, to instantiate a pipeline block, the orchestrator uses NiFi REST API to upload a NiFi XML template that describes the pipeline block. NiFi then registers the template and returns the ID of the pipeline, which in turn is used by the Orchestrator to request NiFi for the pipeline execution. Same happens for every pipeline block in the application topology. At this point, the registered pipeline blocks are established and functional, and the Orchestrator relies on NiFi instances to perform data movements between the pipeline blocks or move data to a certain storage system as a pipeline task.

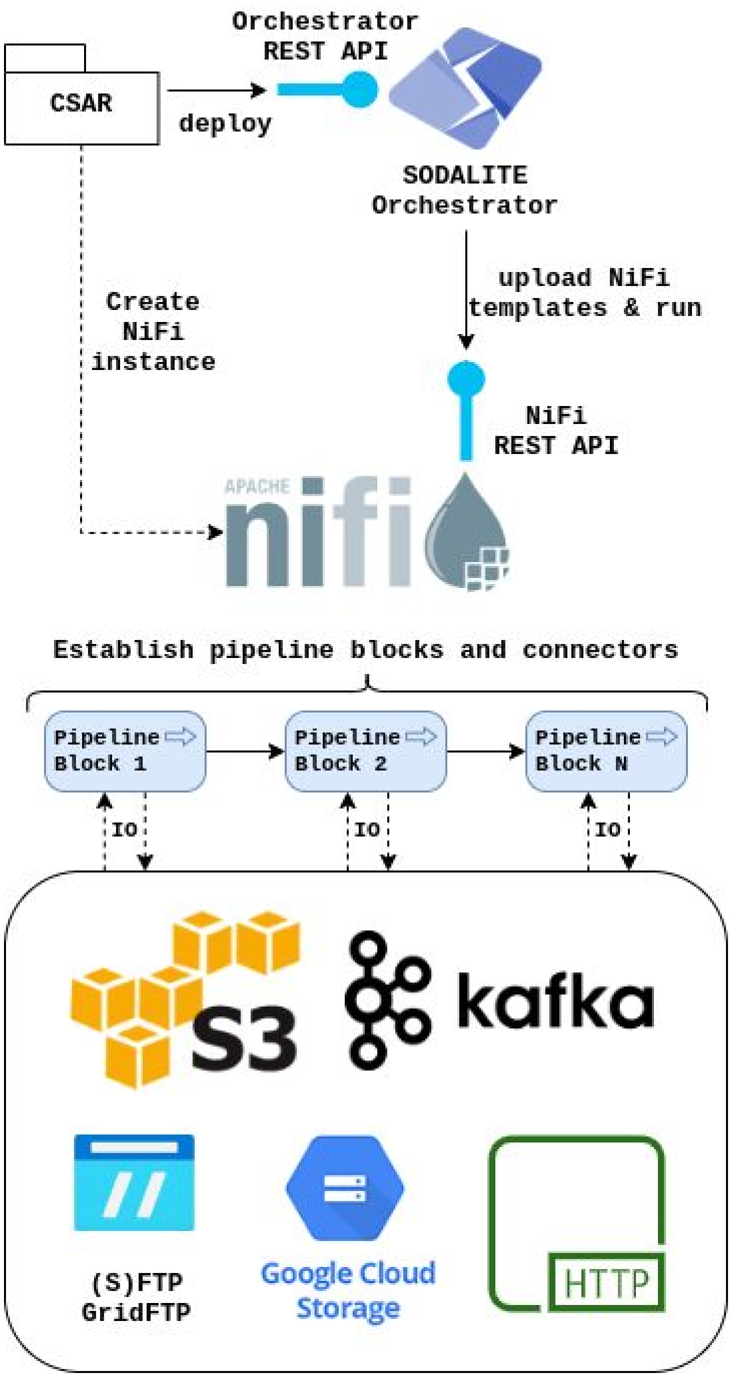


Figure 5 - IaC data pipeline management architecture

As a brief background to the data pipeline approach used in RADON, an outline of its architecture and components is presented in Figure 6. A PipelineBlock is an entity that executes pipeline tasks, such as data processing, API calls invocation, fetching data from or pushing data to remote storage systems or stream platforms, etc. The PipelineBlock may contain input (DataIngestionQueue) and output (DataEmissionQueue) queues for buffering input and resultant data. Using these queues, multiple PipelineBlocks can be connected sequentially, forming a group of PipelineBlocks. Similarly, multiple groups can also be connected using InputPipes - gateways for receiving input data from the previous group or external data source, and OutputPipes - for forwarding resultant data to the next group or external data sink.

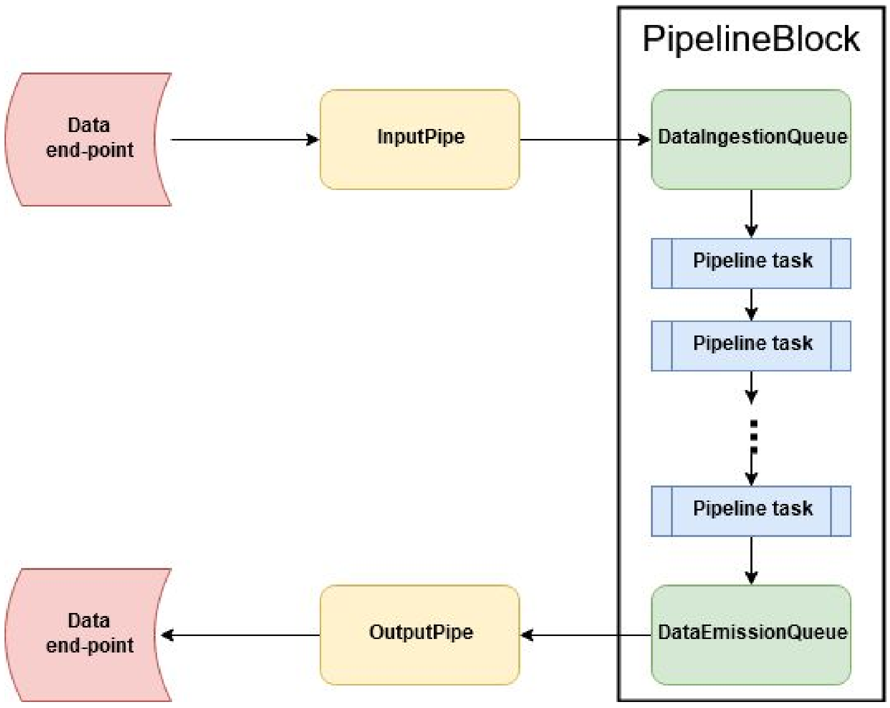


Figure 6 - Architecture of data pipeline [D5.5]

4.2.3 Features

The IaC data management features are limited to the capabilities and functionalities offered by *Apache NiFi*. We mainly focus on utilising NiFi for multi-protocol and multi-platform data movement, abstracted in TOSCA and IaC. Current structure of featured TOSCA node types for data pipeline blocks is presented in Figure 7, and they can be categorised into four classes of pipeline blocks and can be extended:

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

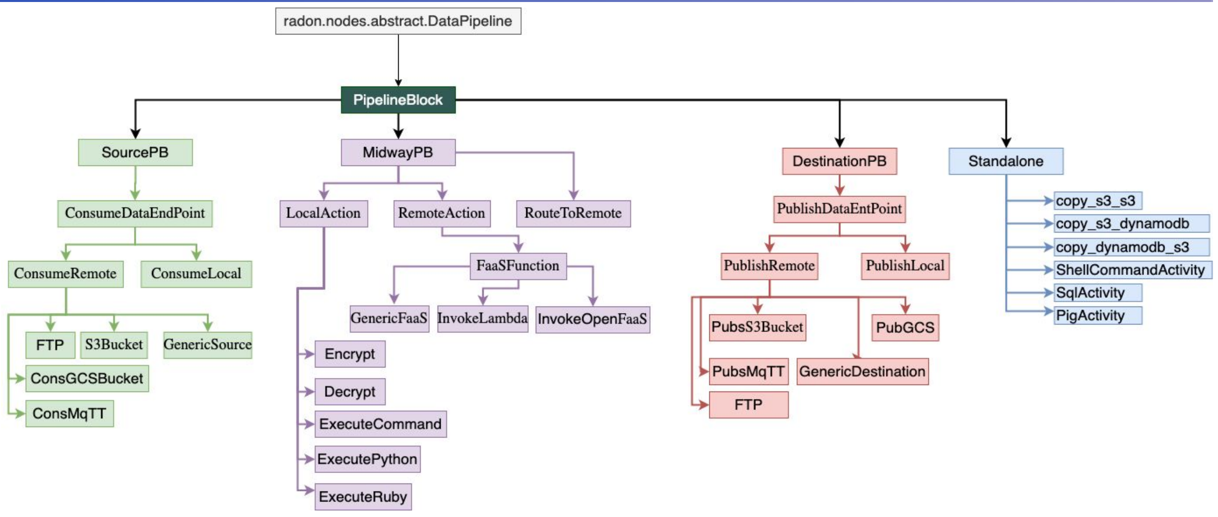


Figure 7 - A hierarchy of featured TOSCA node types

4.2.4 Status

Currently, TOSCA libraries that support publishing to and consuming data from local filesystem, S3, GCS and GridFTP are under development, and can be found in the joint RADON-SODALITE organisation: https://github.com/RADON-SODALITE.

4.2.5 Code Quality

The repositories contain IaC, which is not currently covered by *SonarCloud*, therefore code quality cannot be reported.

4.2.6 Next steps

As a next step, further development and integration into the SODALITE platform is planned. As such, we plan:

- to extend the TOSCA libraries to further support other platforms and protocols, such as (S)FTP, HTTP(S) and Apache Kafka, and evaluate data pipeline approach with use cases;
- to have an instance of Apache NiFi running on the SODALITE Cloud testbed as a data management service;
- to integrate Apache NiFi instances with the IAM and Secret Management components.

4.3 Dynamic Monitoring

4.3.1 Innovation

SODALITE supports the execution of complex applications, consisting of components and/or batch processes, which are deployed into a mixture of infrastructures, including Cloud, HPC or EDGE. One of the defining features of these applications is their ability to dynamically acquire and release computing and networking resources, depending on their specific needs during their runtime. The runtime adaptation requires the analysis of the metrics that describes the behaviour of the application components and their consumed infrastructure resources, which are dynamically allocated/released at runtime. These metrics as collected by probes (or exporters in monitoring terminology), which must be instantiated and configured with the same dynamicity as the application components and associated resources are.

The purpose of dynamic monitoring is the runtime allocation of monitoring probes that collect runtime metrics of target application components, their execution environments and allocated infrastructure resources, hence. As the application deployment may be refactored at runtime, resulting in dynamic redeployment and reallocation of resources, dynamic monitoring is not only required at deployment time, but eventually at any point of the application execution at runtime.

The SODALITE monitoring system, based on *Prometheus*[37] as the main monitoring engine, does not support this level of dynamical allocation and release of probes (see [D5.1] for a state of the art analysis of monitoring approaches). Although *Prometheus* supports the specification of probes that collect runtime data from their associated targets, this needs to be pre-configured before enabling monitoring.

Therefore, a technical innovative redesign of the monitoring system is required. This is achieved by adopting *Consul*[38] (see following paragraphs for further description of this component) as a dynamic registry of monitoring probes, a registry that is consulted by *Prometheus* when it is notified of probe changes. This approach is complemented with a mechanism that permits exporters to register themselves dynamically in the monitoring registry when their activation is required by the orchestrator.

4.3.2 Architecture

The dynamicity of the monitoring system relies on its capability of getting automatically aware of any change in the execution environment and infrastructure resources used by an application. This behaviour is achieved by making the different exporters supported by the monitoring system (*IPMI exporters, Node exporters, Skydive exporters and HPC exporters*) to register itself in a *Consul* server using the API it offers for that, as Figure 8, which depicts the complete SODALITE monitoring layer architecture, shows.

The Prometheus component also makes use of the Consul API, since it is configured to scrape metrics from any exporter of the aforementioned types that have been previously registered in Consul. Let us note that Consul[39] describes itself as a mesh solution for monitoring services in clusters and, as a mesh solution, it was thought to work as a federation of servers and agents running in the distributed premises of the underlying infrastructure. However, just a single Consul instance acting simultaneously as server, local agent and cluster leader is enough for our requirements. Regarding the metrics scraping itself, all the exporters offer an API to which the Prometheus component launches HTTP requests to collect data.

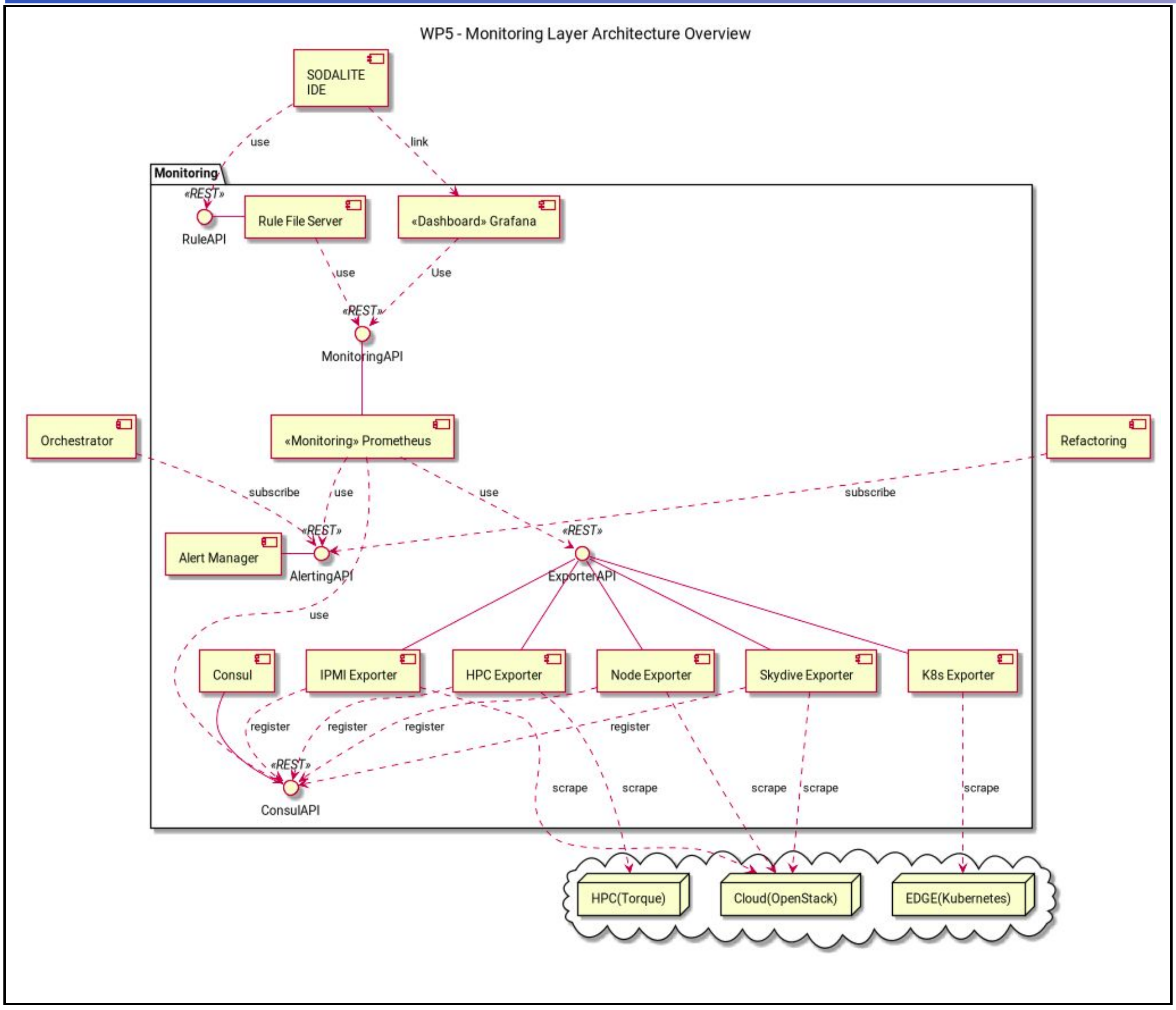


Figure 8 - Monitoring Y2 architecture

4.3.3 Features

Dynamic monitoring adapts the number, type and configuration of probes (i.e. exporters) required to collect monitoring metrics from a target deployed application, the execution environment where it is being executed and the infrastructure resources it uses. Current implementation of dynamic monitoring supports the dynamic allocation of exporters and their registration within the monitoring system upon their initialization. It also supports them to decomiss themselves when they cease on their activity (e.g. they may quit collecting metrics when they detect the target application has been terminated).

4.3.4 Status

Figure 9 shows the status of main Monitoring components, compared to the situation reported in D5.1. This figure distinguishes among:

- components not updated since D5.1, in grey,
- components updated since D5.1, in orange,
- new components developed, in green.

What concerns this feature, a new component, *Consul*, has been adopted and incorporated to the SODALITE architecture. The main monitoring component, Prometheus, has been reconfigured to obtain the list of exporters from *Consul*, and these exporters have been reconfigured to register themselves, upon their initialization, into the *Consul* instance. The *K8s Exporter* for EDGE can optionally register itself in *Consul* (or be discovered indirectly, through the *Consul-K8s Service Sync*[40]), although it is not strictly required as Kubernetes-based exporters are automatically detected and scraped via pod annotation.

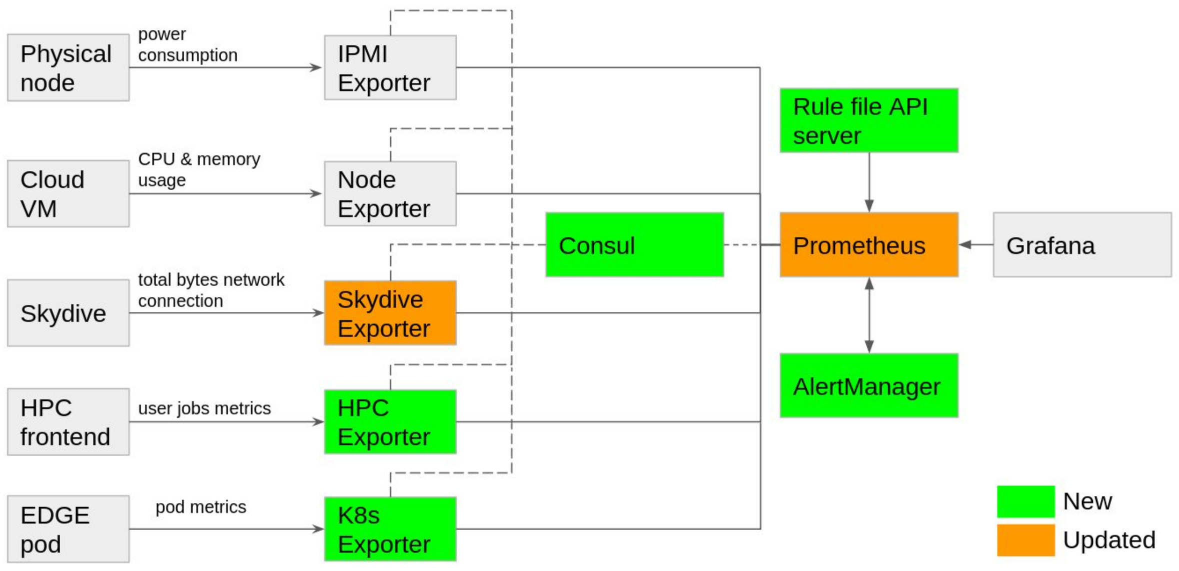


Figure 9 - Monitoring Y2 status (extension of Figure 16 from [D5.1])

In D5.1, the Prometheus instance of the SODALITE architecture was deployed in a VM inside the OpenStack cloud testbed. It was prepared to automatically discover and monitor OpenStack resources thanks to a specific configuration hook[41] of *Prometheus*. Endpoints for other kinds of exporters had to be previously known and set in the configuration file. This was the case of the IPMI and *Skydive* exporters also supported in that initial version. In this current version the *Prometheus* is not responsible anymore of keeping a list of monitoring targets, this task being delegated to the *Consul* instance.

Consul offers a REST API through which services can be registered, the properties of each service being specified in a JSON file passed along with the registration request. *Prometheus* can be configured to retrieve exporter endpoints seamlessly from the *Consul* service registry. These exporters must have been previously registered, what can be done by the exporters themselves after they are started. This can be done at this point since just a service name, a unique identifier, the address and the port of the exporter endpoint has to be included in the JSON file[42] in order to properly register the exporter as a service. Analogously, exporters can remove themselves when stopped. In both cases, the *Prometheus* instance is kept aware of any change with neither server restarting nor configuration file reloading.

4.3.5 Code Quality

Code Quality cannot be reported neither for *Prometheus* nor for *Consul*, as they are third-party components, developed by external teams, which have been incorporated to the SODALITE monitoring layer and integrated to each other. The developments done by the SODALITE team in order to integrate and deploy them within the *Runtime Layer* are mostly related to their mutual integration, and their automatic configuration and delivery, based on TOSCA blueprints and Ansible playbooks, which are not analysed by *SonarCloud*. They are part of the *monitoring-system repository*[43], whose QA report is presented in Figure 10.

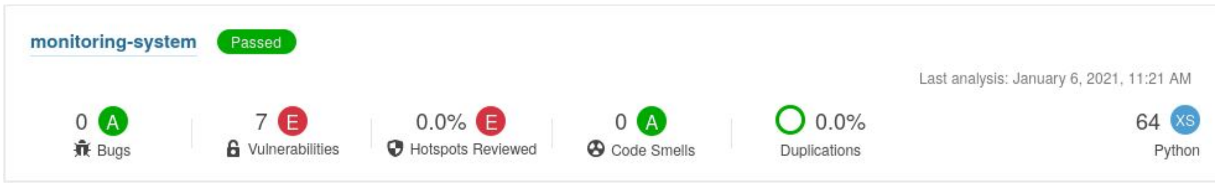


Figure 10 - Code quality report for Monitoring

4.3.6 Next steps

Consul and *Prometheus* instances are fully integrated with the other components of the *Monitoring Layer*, and they can be deployed from TOSCA blueprints by the *Orchestrator*. The feasibility of this approach has been confirmed in an proof of concept (PoC) implemented in the OpenStack testbed, which installs in a VM (by using the *orchestrator*, configured with TOSCA blueprints and Ansible playbooks) instances of *Consul*, *Prometheus*, and a *Node Exporter* that registers itself upon initialization. All the blueprints and playbooks needed for deploying that PoC are available in the *consul-registration-poc/openstack* folder of the *SODALITE-EU/monitoring-system*[44] repository.

However, full automation of the registering and deregistering processes for all the exporters available (IPMI exporters, Node exporters, HPC exporters and Skydive exporters) is still ongoing:

- Registration of *Skydive* exporters is tentatively supported by means of a TOSCA blueprint[45] run by the *Orchestrator* to deploy the SODALITE platform stack.
- Regarding the *Node exporters*, the current playbooks for creation and destruction of VMs of different cloud providers must be updated to install/register and to deregister the exporter, following the example[46] included in the aforementioned PoC.
- Several integration issues related to the lifetime, scope, number and containerization of HPC exporter instances that must run for each application are preventing for the moment the automation of its registration process.

4.4 HPC Monitoring

4.4.1 Innovation

Monitoring the execution of HPC jobs is required to tune the behaviour of HPC-based applications on the pursuit of maximizing their performance and throughput. In the HPC realm, job monitoring is supported by the specific scheduler, but restricted to be manually consulted by the job owner or the administrator of the HPC cluster[47] [48]. As the main objective of HPC environments is to achieve the maximum job performance with the existing computational resources, no monitoring services are installed in the background. Moreover, there are no installed dashboard-based solutions to monitor the status of submitted jobs and visualize their allocation and consumption of computational resources, aggregated (or filtered) by job, by composition of jobs, by user, by queue, or by another view that could be required by the user.

The innovative aspect of this approach is the adoption of external[49], dynamical[50] HPC probes, integrated within a multi-infrastructure target monitoring layer, that collect and aggregate metrics about the execution status and consumed resources of target jobs. Other similar external monitoring approaches (e.g. as presented by Röhl[51]) neither dynamically configure HPC probes for multi-targeted HPC infrastructures nor are integrated within a hybrid monitoring approach that aggregates metrics from HPC, Cloud or EDGE.

The ability of monitoring multiple HPC clusters, regardless of the adopted scheduling technology, is another innovative feature offered by this approach.

4.4.2 Architecture

As Figure 8 shows, the HPC monitoring capabilities rely on HPC exporters able to scrape metrics from HPC infrastructures. These exporters query the monitoring utilities of the schedulers of the target HPC infrastructures and use the information obtained to compose and expose metrics. Once registered as services in the *Consul* component, the *Prometheus* server can scrape metrics from them.

4.4.3 Features

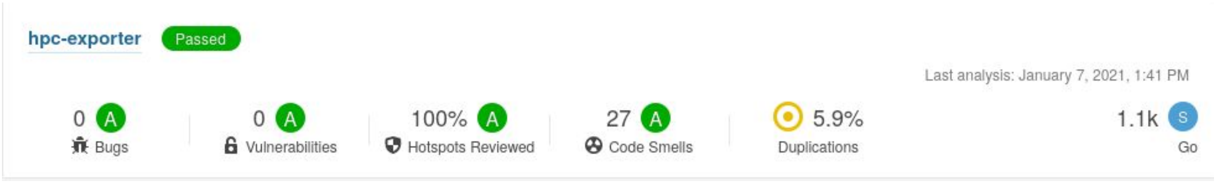
The HPC exporter collects metrics about different aspects of HPC infrastructure usage from both performance and cost viewpoints: job status, CPU and wall time consumption, physical and virtual memory consumption, energy consumption, traffic for I/O and network communications, etc. Different types of HPC job schedulers are supported, starting with PBS Professional and SLURM in the current implementation.

4.4.4 Status

The current *HPC exporter* (available in *SODALITE-EU/hpc-exporter*[52] repository) is a tool implemented in Go which is able to connect to frontends of HPC infrastructures and give information about the status, the consumption of time (CPU, wall) and memory (virtual, physical) for a list of jobs launched by a given user in a given HPC infrastructure. The tool opens an SSH connection to the corresponding frontend, launches *PBS Professional* or *SLURM* commands (depending on the scheduler available in the monitored HPC infrastructure) commands through that connection and parses their output to obtain the desired metrics. By means of the Go client library for *Prometheus*, these metrics are formatted and exported through a valid endpoint. Separate instances of the exporter can run directly by launching different instances of the tool executable, or by building the Docker image specified in the *Dockerfile*[53] provided and running multiple instances of it in several containers.

4.4.5 Code Quality

Code quality report for the *HPC exporter* code is shown below. This report (Figure 11) spots some code duplication that will be investigated and reduced for the next release.


hpc-exporter Passed

Last analysis: January 7, 2021, 1:41 PM

0 A Bugs | 0 A Vulnerabilities | 100% A Hotspots Reviewed | 27 A Code Smells | 5.9% Duplications | 1.1k S Go

Figure 11 - Code quality report for HPC exporter

4.4.6 Next steps

In its current state, each instance of the *HPC exporter* is able to monitor a given list of target jobs, submitted by a user in a HPC infrastructure. Further research is needed in relation to the number of the *HPC exporters* each application needs, and also about their lifetime and scope. Such decisions will affect how this tool is deployed (blueprints, playbooks, proper Docker containerization, etc.) by the *Orchestrator*.

Regarding the metrics collected by the *HPC exporters*, by now only metrics about job status, CPU and memory consumption are provided. Current exporter implementations for TORQUE/PBS Pro and SLURM can be extended to query those schedulers about the number of jobs submitted to queues, which will provide a way to estimate queues congestion.

Moreover, job status, CPU and memory consumption metrics are easy to obtain since they are directly provided by the information commands of the HPC schedulers with no collateral effects on the execution of the jobs themselves. Having access to metrics like I/O and network usage or power consumption is not so trivial, since runtime interaction with the computing nodes is needed up to some extent, which is not usually allowed by the HPC infrastructure administrators and may also affect the performance of the jobs being monitored. Further research is needed about how these metrics can be retrieved, including the development of an ad-hoc exporter able to collect them without affecting such a sensitive aspect of HPC infrastructure operations.

It would be also interesting to do some research about how end users could benefit from *Grafana* specialized dashboards for monitoring the state of their applications running on HPC premises, for example with different views per job, per queue, etc.

4.5 Edge Monitoring

4.5.1 Innovation

Monitoring of Edge Gateways is critical for ensuring their continued operability and to reduce the risk of service failure arising from environmental factors such as resource limits or safe operating thermal ranges being exceeded. As Edge Gateways are often multi-user systems, and are often passively cooled, it is integral to have a complete overview of the system's operating parameters at run-time, regardless of whether these are caused directly or indirectly.

As resources in an Edge Gateway may become periodically unavailable (e.g. a resource is removed from the cluster, or required by a high-priority application), the monitoring stack must be capable of dynamically reconfiguring itself to match the current system state. To this extent, the current monitoring solution integrates tightly with Kubernetes, relying on dynamic device and feature discovery to label nodes with node characteristics that are used as a basis for determining Pod scheduling affinity. For example, an *EdgeTPU exporter* may be registered once at the cluster level, and will automatically be scheduled on any node in which an EdgeTPU device becomes available - whether through static platform discovery (in the case of integrated devices), or dynamically at run-time via USB hotplug events.

4.5.2 Architecture

The *Edge Monitoring* architecture is briefly described in Figure 12 below:

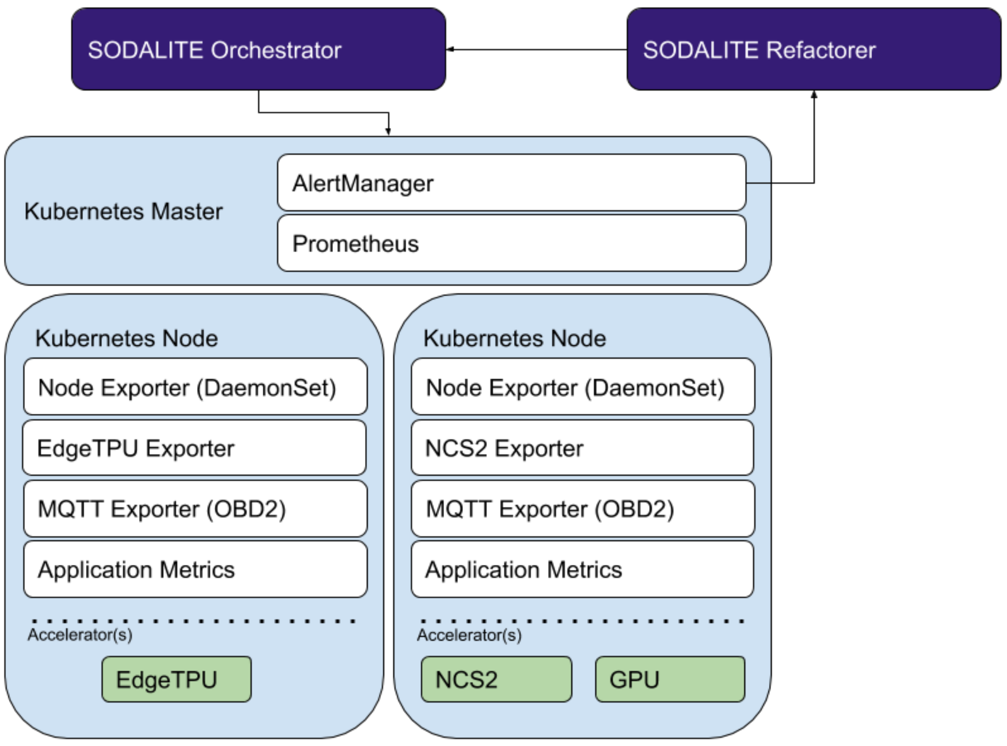


Figure 12 - Edge Monitoring architecture

In this configuration, a single *Prometheus* and *Alertmanager* instance is provided within the cluster, and each Edge node is automatically instantiated with a *Node Exporter* instance by the Kubernetes master. Individual exporters (e.g. for device-specific metrics) are registered at the cluster level, and are automatically scheduled on any Edge node matching the scheduling constraints.

4.5.3 Features

The first version of the *Edge Monitoring* stack supports the dynamic registration of device-specific exporters across Kubernetes nodes, and is able to handle the dynamic insertion/removal of heterogeneous accelerators at run-time. Annotated pods are automatically scraped by the cluster-level *Prometheus* instance, typically running on the Kubernetes master node, and require no specific configuration.

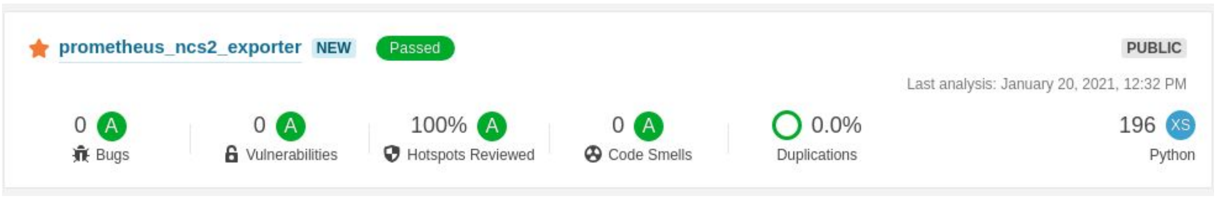
4.5.4 Status

Exporters for a number of heterogeneous accelerators used by the Vehicle IoT use case have been developed (specifically, exporters for *Edge TPUs* and *Intel NCS2 Neural Compute Sticks*) and integrated. Others (e.g. integrated *NVIDIA Tegra GPUs*) require modification to existing upstream device plugins, and the definition of customized platform-specific configurations. This work is on-going.

4.5.5 Code Quality

The *Edge Monitoring* stack is composed of a number of different components, and does not provide a single integrated package. As most of the components in the *Edge Monitoring stack* are written in different languages (Golang, Python, C++), there is no single comprehensive quality metric applied - each component is ultimately responsible for monitoring its own quality. Components that do provide quality metrics (e.g. the Go Report Card for Golang-based components) provide this as part of their documentation in their respective repositories.

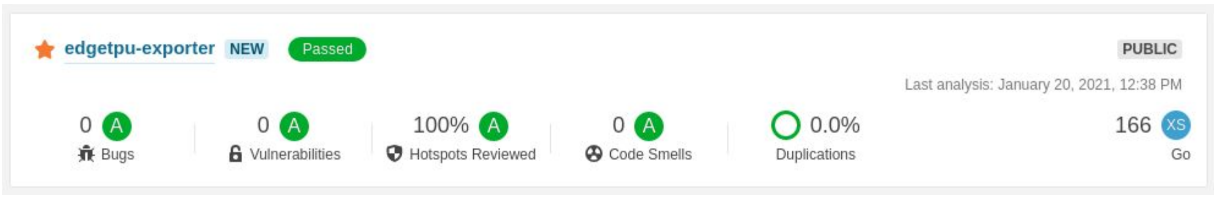
The *SonarCloud* code quality report for the corresponding exporters are provided in Figure 13:


prometheus_ncs2_exporter NEW Passed PUBLIC

Last analysis: January 20, 2021, 12:32 PM

0 A Bugs | 0 A Vulnerabilities | 100% A Hotspots Reviewed | 0 A Code Smells | 0.0% Duplications | 196 XS Python

Figure 13 - Code quality report for the corresponding exporters


edgetpu-exporter NEW Passed PUBLIC

Last analysis: January 20, 2021, 12:38 PM

0 A Bugs | 0 A Vulnerabilities | 100% A Hotspots Reviewed | 0 A Code Smells | 0.0% Duplications | 166 XS Go

Figure 14 - Code quality report for Edge Monitoring components

4.5.6 Next Steps

The next steps will be to: (1) create node-local instances of *Prometheus/Alertmanager* to allow for finer-grained alerting and monitoring, while allowing a cluster-level *Prometheus/Alertmanager* instance to federate the node instances for cluster-wide dashboards and monitoring; and (2) allow for the dynamic discovery and registration of platform-specific alerting rules (e.g. different Edge Gateways may expose CPU or GPU monitoring data via different hard-coded thermal zones, which should be transparent to the monitor).

4.6 SkyDive Network Monitoring

4.6.1 Innovation

Various monitoring tools provide network metrics[54]. One of the distinguishing features of *Skydive* is that it understands the topology of how entities are connected in the network and it has the capability to identify individual network flows from specific network connections. This enables a user to trace how a flow travels through the network topology and to identify where there might be potential problems that require attention. The *skydive-prometheus-connector*[55] is a recent add-on to *Skydive* to export data from captured flows for allowing easy consumption by *Prometheus*. The *skydive-prometheus-connector* is built upon the skydive-flow-exporter[56] infrastructure.

4.6.2 Architecture

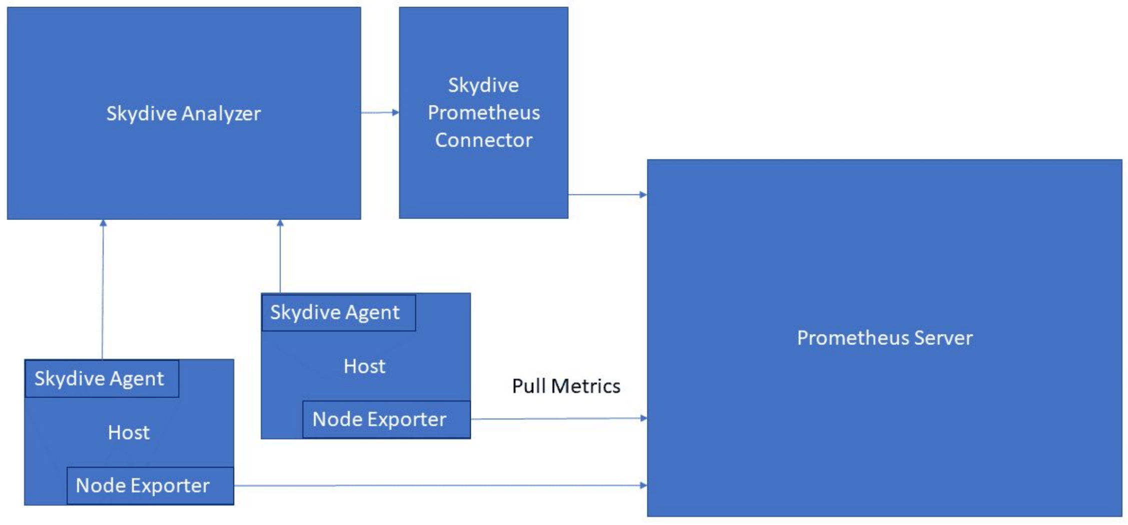


Figure 15 - Skydive-Prometheus Connector

A Skydive agent runs on each host that is being monitored to collect the network monitoring information that is requested. The data from the various *Skydive* agents is sent to the *Skydive Analyzer* to be collected and managed. The skydive-prometheus-connector extracts the relevant information from the *Skydive Analyzer* and converts it into a format that is consumable by Prometheus.

4.6.3 Features

The first implementation of the *skydive-prometheus-connector* provides the byte transfer counts plus metadata for each network connection under observation. This information, probed periodically, can be used to calculate the rate of data transfer per network connection. The code can be easily extended to provide additional metrics that are found to be useful.

4.6.4 Status

The *skydive-prometheus-connector* was recently pushed upstream to the *Skydive Github* repository[57]. A blog[58] was prepared with detailed instructions and examples of using the *skydive-prometheus-connector*. A blog[59] was also published describing the more general usage of the skydive-flow-exporter. The *skydive-prometheus-connector* and *Skydive* analyzer were recently incorporated into the SODALITE platform deployment blueprint[60].

4.6.5 Code Quality

Code Quality for *Skydive* and its extensions cannot be reported, as they are third-party components, and they do not provide the *SonarCloud* statistics.

4.6.6 Next steps

At present, the network metrics are simply reported, but no action is taken based on the network metrics. We aim to be able to recognize some network anomaly or bottleneck, and to be able to recommend a change in the configuration to improve performance. In addition, data is currently captured uniformly from all hosts in our configuration. The collection of such data requires some overhead. We aim to be able to (dynamically) selectively choose where and when to collect network data, based on detection of some anomalous behaviour, thus avoiding unnecessary collection of most of the network data.

4.7 Alerting

4.7.1 Innovation

According to section 4.1 (*Monitoring, tracing and alerting - Background*) of [D5.1], the alerting part of a system is essential because it notifies critical changes or situations, assists the operators to have a good overview of the state of all the infrastructure, and being on top of the monitoring and tracing parts helps to spot, mitigate and prevent problems. Despite the adopted monitoring engine, *Prometheus*, enables the notification of alerts to its registered *Alert Managers*, upon the detection of conditions that triggers the provided alerting rules. Then, a proper management of those alerts and their dispatching to subscribers within the *Runtime layer* (notably the *Orchestrator* and the *refactoring* components) is required.

No alerting components were described in sections 4.4 (*Monitoring, tracing and alerting - Architecture*) and 4.5 (*Monitoring, tracing and alerting - Development status*) of D5.1, so in this intermediate version of the runtime, the foundations of such an alerting system are established.

4.7.2 Architecture

The alerting part of the monitoring system has three main components: the *AlertManager*, the *Rule File API Server* and the *Grafana* dashboard. As Figure 8 shows, these three components interact with the *Prometheus* server, since it is the central repository of all the monitoring information collected from the exporters. The *Orchestrator* and *Refactoring* parts are subscribed to the *AlertManager* in order to be notified about detected changes or malfunctions in the runtime target application behaviour so that they can analyse the need of redeployment and/or reconfiguration. Moreover, the *Grafana* dashboard provides end-users with human-readable alerting information.

4.7.3 Features

The alerting part of the monitoring system is composed of an *AlertManager* that handles the alerts triggered by the *Prometheus* server and notifies any component or actor subscribed to them. In this case both the *Orchestrator* and the *Refactoring* part are interested in alerts, so that these components are responsible for adapting the resources available according to any relevant change on the infrastructure derived from the alerts triggered. These alerts are defined for each application on rule files which are kept by the *Rule File Server*. This server offers a REST API that allows the applications to register rule files in order to be evaluated by the Prometheus component, which is properly configured to access these files. The *Grafana* dashboard, used to plot and summarise the information collected in the *Prometheus* server, also offers its own alert management features.

4.7.4 Status

The *Prometheus* and the *Rule File Server* are deployed on Docker containers that share a volume on which the alerting rule files are stored. Applications register their rule files through the Rule File API, and the server saves them in the shared volume. A path to that volume is kept in the *Prometheus* configuration file. The *Rule File API Server* component is intended to loosen the coupling among the rule files lifetime and its actual enforcement in the *Prometheus* instance. The aforementioned API is built on top of Flask (a lightweight WSGI web application framework), with the requests being received and redirected to *Flask*[61] by a multithreaded instance of *Gunicorn*[62] (a WSGI production-ready HTTP server). Since both *Flask* and *Gunicorn* are written in Python, the component is deployed using a custom Docker image based on Python3.8 one and with additional layers for *Flask* and *Gunicorn*. This image has been uploaded as *monitoring-system-ruleserver*[63] to the SODALITE public image registry on Docker Hub. The Dockerfile used to build the image along with the implementation of the Flask application and its requirements are available in the folder ruleserver of the *SODALITE-EU/monitoring-system*[64] repo, along with a README file describing the usage of this component REST API for adding and removing rule files.

Prometheus may be configured to periodically send information about alert states to an *AlertManager* instance, which then takes care of dispatching the right notifications. This *AlertManager* instance is deployed in its own container using the official Docker image provided by the *Prometheus* development team. This container offers an endpoint that is included in the *Prometheus* configuration file in order to make the *Prometheus* send the alerts triggered to the *AlertManager*. This reported version of the monitoring system implements the foundations of the alerting feature, but no concrete behaviour is supported regarding notifications nor alert operations.

4.7.5 Code quality

The code quality report for *AlertManager*, included as part of the *monitoring-system repository*[65], is reported in Figure 16. As this component is currently under development it still faces some vulnerabilities that will be fixed for next release, once the component functionality is complete.

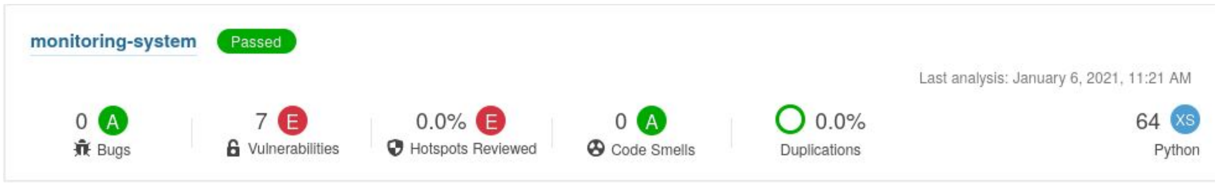


Figure 16 - Code quality report for Monitoring Alerting

4.7.5 Next steps

By now just the foundations of the alerting system are set, since there is an *AlertManager* that is able to receive alerts from the *Prometheus*, which, in turn, is able to enforce the alerting rule files kept by the *Rule File Server*. However, no real behaviour is yet implemented.

As a starting point, it would be interesting to prepare a PoC to assess up to which extent this alerting system could support the implementation of scaling or reconfiguration policies. A feasible experiment could be: (1) defining an alert that is triggered when the CPU of a VM is overloaded, (2) subscribe the *Orchestrator* and the *Refactoring* parts to such an alert through the *AlertManager*, (3) force that overload in the VM (4) , and make the aforementioned parts of the system to react to the problem by instantiating a new VM that will share the workload with the first one. If such a PoC is successful, then real alerting files, subscriptions and reactions could be implemented for the different use cases. With relation to that, further research is also needed about how the alerting capabilities of the *Grafana* dashboard can be integrated in the alerting part of the monitoring system.

4.8 Deployment Refactorer

According to Martin Fowler [66], “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. We aim to provide the system and language support that are tailored to deployment model refactoring (See *Policy-based Deployment Adaptation*).

A configurable system provides a number of options that enable users to generate many variants of the system where each variant is tailored to support a given set of functional and quality requirements[67]. In the context of the SODALITE, we consider the systems whose components can be deployed in various ways (many *deployment options*), which is a type of a configurable system. The configurable system engineering is a common approach to building self-adaptive systems, where the adaptation is modeled as switching between system variants [De Lemos 2013]. Along the same lines, we model the safe adaptation of a deployment model of an application as switching between allowed deployment model variants, where each deployment model is a subset of deployment options. To enable deployment model switching, an efficient approach to estimating the performance metrics for many deployment model variants is needed (see *Alternative Deployment Configuration Selection*).

In software development, the code refactoring is commonly performed to remove the "code bugs and smells"[57] . The deployment models can also have "bugs and smells". Thus, the deployment refactoring also aims to detect and remove the bugs and smells in the deployment model/topology of a system. In particular, within the context of the SODALITE, we focus on detecting and removing performance bugs/anomalies (see *Data-driven Performance Anomaly Detection*) and deployment policy violations (See *Policy-based Deployment Adaptation*)

4.8.1 Innovation

Policy-based Deployment Adaptation

The rule-based (reactive) systems are one of the common approaches used by self-adaptive system research literature to enable capturing and enacting adaptation decisions[68]. While there exist rule-based systems for common software applications, there is a lack of such systems tailored to runtime management of heterogeneous application deployments. The management system needs to support different classes of runtime changes that can potentially occur to a deployment model of a heterogeneous application. The changes need to be realized and managed at runtime without disturbing the operations of those users unaffected by the change. A software engineer should be able to declaratively specify changes and enact the change specification at runtime to modify the application. SODALITE aims to provide a policy-based deployment adaptation support that meets these requirements.

Alternative Deployment Configuration Selection

The increasing heterogeneity of computing resources gives rise to a very large number of deployment options for constructing distributed multi-component applications. In principle, application providers can leverage this heterogeneity to optimize the application performance and cost. However, in practice, selecting the most appropriate combination of deployment options is hard because of the large deployment space[69] [70]. One way to address this problem is to build a comprehensive performance model for the application by measuring the performance of the different deployment variants, i.e., valid combinations of deployment options, upfront. However, such an exercise can be extremely costly and time-consuming. Therefore, we propose a novel approach for modeling and predicting the performance of valid deployment variants of a distributed application – based on the observed performance of a small subset of the variants. We first model the valid and practical set of deployment variants for a given application by leveraging domain knowledge about the commonalities and variations in the deployment options. We then measure the performance of a selected subset of these deployment variants to create a suitable performance model that we use to predict the performance of the remaining deployment variants.

The performance models are built by applying machine learning and deep learning by using the deployment options as features (independent variables) of the models.

Data-driven Performance Anomaly Detection

An anomaly can be defined as a rare event where the system behavior deviates from what is standard, normal, or expected[71]. For example, a service could use an anomalous amount of resources, or the service network exhibits traffic anomalies. The ability to detect anomalies and trigger corrective actions is critical to maintain the quality of service, and to prevent runtime service failures and undue usage of resources. While anomaly detection has been a popular topic in the research literature, there are only few recent studies on microservice performance anomalies[72]. The microservice based applications are complex, interconnected web of services. There are a lack of labeled datasets reporting different types of anomalies occur in microservice-based applications, in particular, service network anomalies. Consequently, there is also a lack of a machine learning based framework for detecting such anomalies.

We aim to provide a dataset of service network anomalies for the microservice-based applications. A machine learning based framework is also developed to support continuous performance anomaly detection in microservices. By utilizing various capabilities of SODALITE (e.g., monitoring and alerting, platform and resource discovery, policy based adaptation, and redeployment), the runtime detection and correction of anomaly behaviors are supported.

4.8.2 Architecture

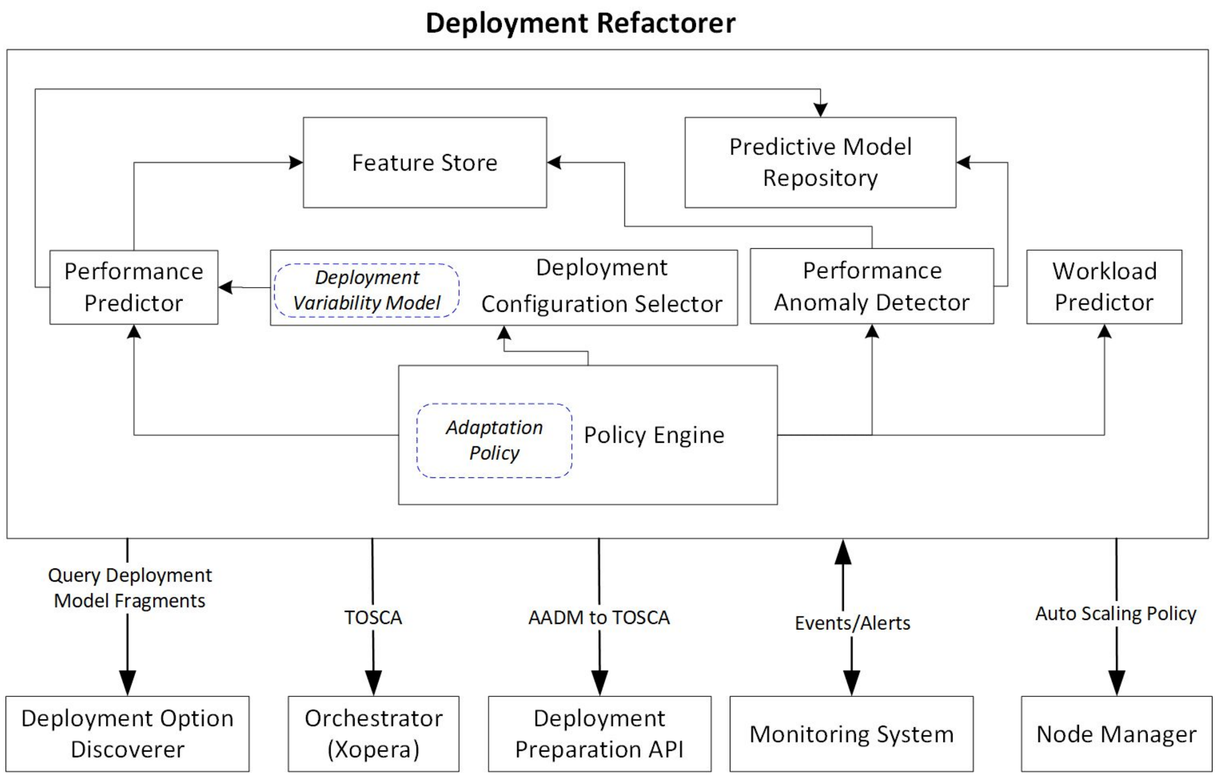


Figure 17 - Overall Architecture of Deployment Refactorer

Figure 17 shows the overview of the detailed architecture of *Deployment Refactorer*. The overall deployment adaptation logic can be codified as an ECA (Event-Condition-Action) policy. *Policy Engine* can enact and manage such policies. In order to build complex policies, *Deployment Refactorer* provides a set of utilities: *Workload Predictor*, *Performance Predictor*, *Deployment Configuration Selector*, and *Performance Anomaly Detector*. *Workload Predictor* uses linear and polynomial regression models to forecast the workload (the number of requests for the next period). Given the predicted workload and the deployment options used, *Performance Predictor* can predict the performance metrics. If the current deployment model variant cannot meet the performance goals, *Deployment Configuration Selector* can be used to find an alternative deployment model from the allowed set of deployment model varnats (expressed in the deployment variability model). *Performance Anomaly Detector* can be used to continuously monitor the current deployment for anomaly behaviours, and generate alerts. The predictive ML models used by each of these components are stored in the *Predictive Model Repository*. The features used by such models are stored in the *Feature Store*. In the rest of this section, the support for the key capabilities of *Deployment Refactorer* are presented.

Policy based Deployment Adaptation

In response to the events generated by the components such as *Monitoring System*, *Node Manager*, *Performance Predictor*, and *Performance Anomaly Detector*, the *Deployment Refactorer* carries out the desired changes to the current deployment of a given application. To allow a software engineer to define the deployment adaptation decisions, we provide an ECA (event-condition-action) based policy language. Figure 18 presents the key concepts of the policy language. A policy consists of a set of ECA rules.

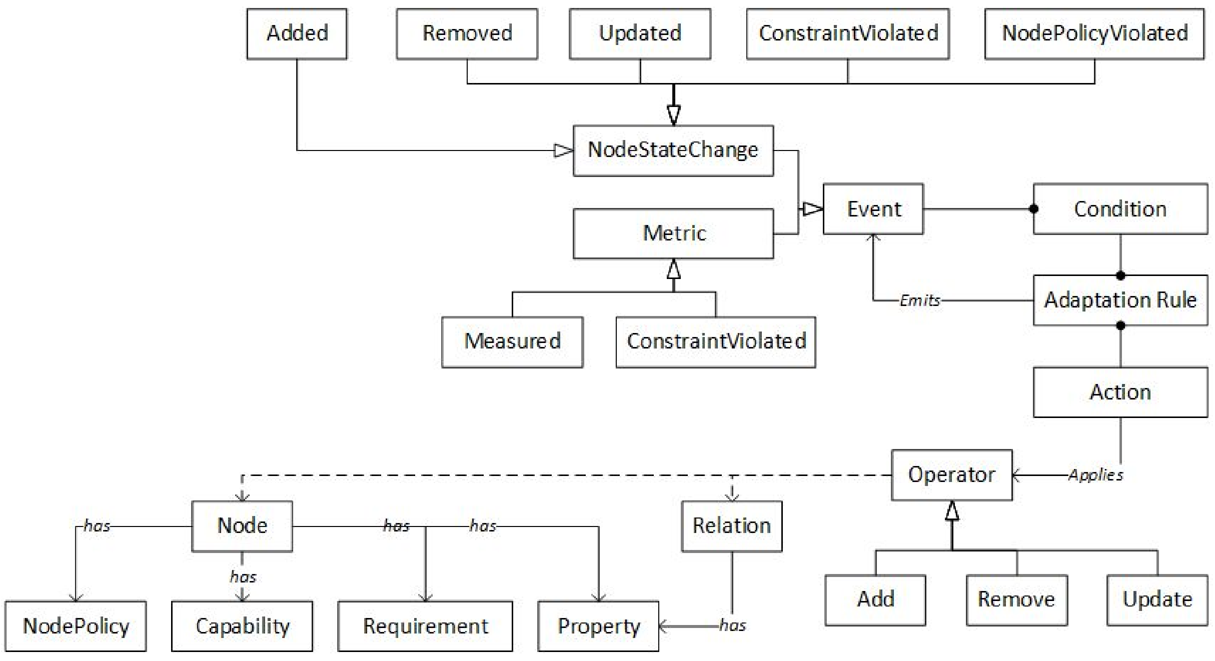


Figure 18 - Meta-model of Deployment Adaptation Policy Language

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

Actions. The actions primarily include the common change operations (Add, Remove, and Update) and the common search operations (*Find* and *EvalPredicate*) on nodes and relations in a deployment model. Additionally, the custom actions can be implemented and then used in the deployment adaptation rules, for example, actions for predicting performance of a particular deployment model instance or predicting workload. To ensure the safe and consistent changes to the deployment model instance, *Deployment Refactorer* makes the change operations to a local representation (a Java Object model) of the deployment model (represented using the concept of models@runtime[73]). Once the adaptation rules in a rule session are executed, *Deployment Refactorer* translates the current local object model to the TOSCA file, and calls the relevant API operation of the Orchestrator with the generated file to enact the updated deployment model.

Execution. The correct ordering of the rules as well as that of the actions within each rule are required to achieve a desired outcome. The rules are independent and are activated based on their conditions. When multiple rules are activated at the same time, the priorities of the rules can be used to resolve any conflicts. Within a rule, if-then-else conditional constructs can be used to order the actions.

Figure 19 show an example of a deployment adaptation rule that reacts to the event *LocationChangedEvent* by undeploying a data processing service deployed in a VM located in a data center at the previous location (de-Germany), and deploying the same service in a VM from a data center at the new location (it-Italy). A predicate over the TOSCA node properties location service name is used to find the correct TOSCA node template.

```
import eu.sodalite.TOSCARepositoryAPI repoAPI;

rule "location_change_from_de_to_it"
  when
    $f1 : LocationChangedEvent(preLoc == "de", currentLoc == "it") and
    $dm : DeploymentModel()
  then
    $dm.removeNode("(?location = \"" + $f1.getPreLoc() + "\" )
                 && (?service-name = \"" + $f1.getServiceName() + "\" )");
    $dm.addNode(repoAPI.find("(?location = \"" + $f1.getCurrentLoc() + "\" )
                 && (?service-name = \"" + $f1.getServiceName() + "\" )"));
    emit NodeReplaced($f1.getServiceName());
  end
```

Figure 19 - A snippet of a Deployment Adaptation Rule

Alternative deployment configuration selection

Figure 20 depicts the workflow of our approach for building a performance model that can predict the performance of each of the alertiave deployment model variants by using the performance measurements from only a subset of variants. The initial performance models are built offline, and at runtime, based on the monitored data, the models are retrained as necessary, for example, if the model accuracy drops below a predefined threshold.

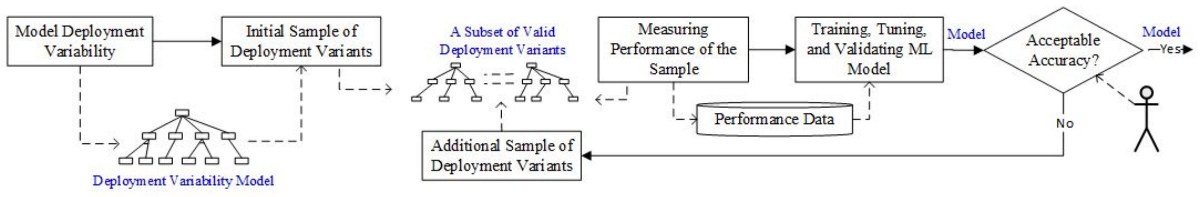

Model Deployment Variability
Initial Sample of Deployment Variants
A Subset of Valid Deployment Variants
Measuring Performance of the Sample
Training, Tuning, and Validating ML Model
Model
Acceptable Accuracy?
Model
Yes
No
Performance Data
Additional Sample of Deployment Variants
Deployment Variability Model

Figure 20 - An Overview of Our Approach to Building A Performance Model for a set of Alternative Deployment Model Variants

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

To model the allowed variations in the deployment topology of an application, we use feature modeling technique, which is a widely-used variability modeling technique[74] [Berger 2013], and is also supported by open source and commercial tools. We used *FeatureIDE*[75], which is an Eclipse plugin that can be installed into the *SODALITE IDE*. A feature model can represent the commonalities and variations in a family of artifact variants as configuration options and their inter-dependencies and other constraints. An artifact can be a software system, application, design model, and more. By respecting all the constraints defined by the feature model, we can select a subset of configuration options (called a *feature configuration*), which represents a valid artifact variant or a family member. Figure 21 shows a feature model for a benchmark web application (used in the evaluation of our approach). The feature *leaf* nodes represent the component deployment options, which are the unique assignments of application components to VMs. Similarly, the feature group nodes (non-leaf nodes) and their hierarchical organization captures the logical decomposition of the deployment decisions. For example, the *web server* and the *database cache* can be deployed together (*co-deployment*) or separately (*separate-deployment*), which can be modelled using a *XOR* feature group.

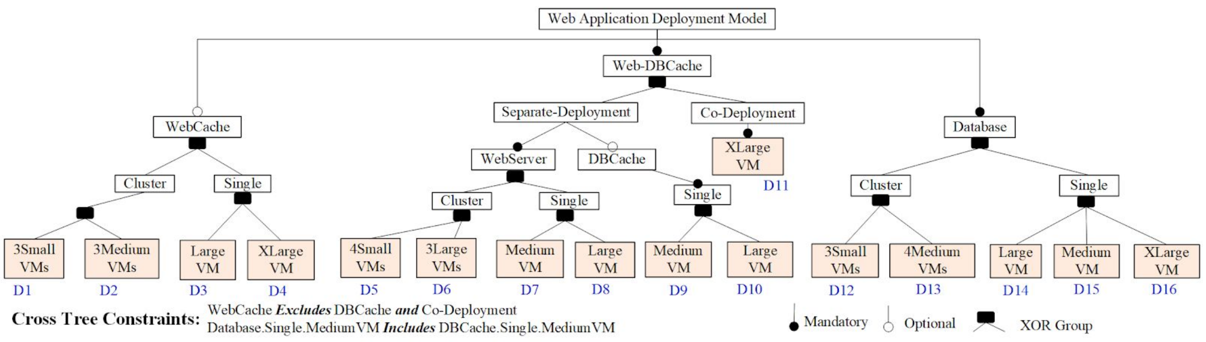


Figure 21 - A Feature Model, Capturing Deployment Variability in the RuBiS Web Application

To sample a variability model (i.e., to select a subset of deployment model variants), there exist many sampling strategies proposed by the research literature in the performance modeling of configurable systems[76]. In our current implementation, we experimented with three sampling techniques: random sampling, T-wise sampling, and dissimilarity sampling.

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

To build the predictive models, the current literature in configurable systems has used many different learning algorithms, including traditional machine learning algorithms as well as deep learning models. In our current implementation, we used the following three models: *Decision Tree Regression (DTR), Random Forest Regression (RFR), Multilayer Perceptron Neural Network (MLP).*

The prediction of the performance (a continuous value) of individual deployment variants is a regression problem. It essentially learns a mapping function (f) from input variables (X) to a continuous output variable (y). The input variables are the leaf nodes in the feature model, i.e., the component deployment options, and the workload levels, and the output variable is response time.

The performance prediction model is used at runtime to predict the performance of a given deployment variant for a given workload. If the current deployment model cannot satisfy the performance goals, then, a deployment model variant that can meet the performance goals is selected.

Data-Driven Performance Anomaly Detection

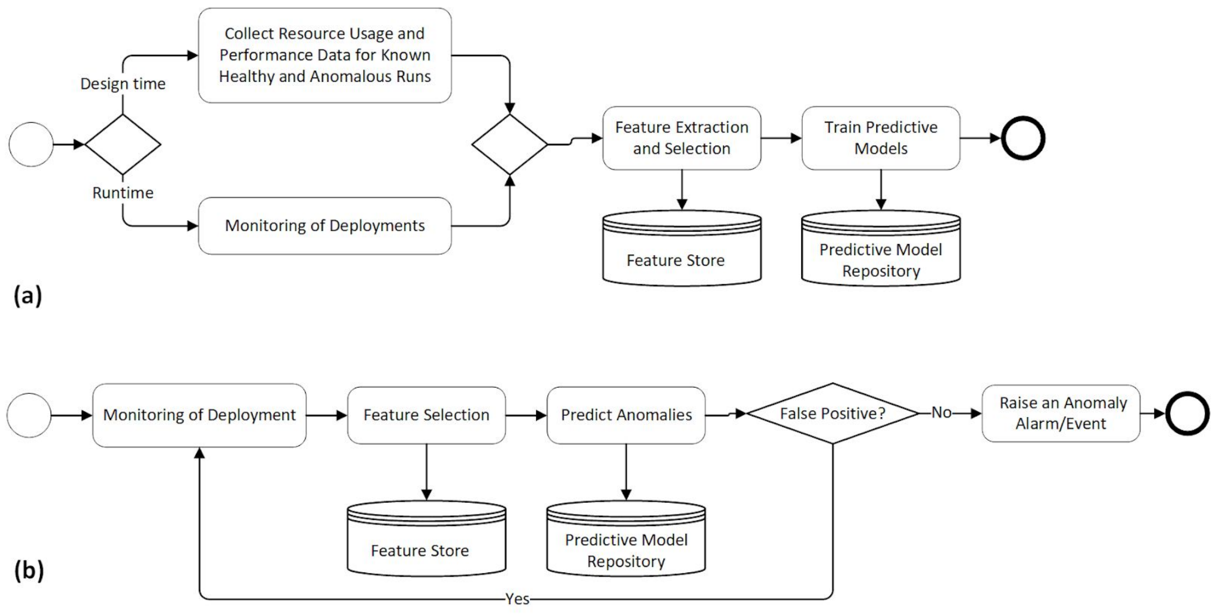

Design time
Collect Resource Usage and Performance Data for Known Healthy and Anomalous Runs
Runtime
Monitoring of Deployments
Feature Extraction and Selection
Train Predictive Models
Feature Store
Predictive Model Repository
(a)
Monitoring of Deployment
Feature Selection
Predict Anomalies
False Positive?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

Figure 22 - An Overview of our Anomaly Detection Approach : (a) Training Workflow (Offline and Runtime), (b) Prediction Workflow (Runtime)

We aim to detect whether a compute node or a cluster is anomalous and classify the type of the anomaly at runtime, independent of the microservice application that is running on the compute node or the cluster. To detect and classify anomalies, we propose a machine learning based approach (see Figure 22). We first build the machine learning models at the design time by utilizing historical resource usage and performance data that are collected from healthy and anomalous situations. Then, at runtime, we apply these models to the monitoring data from the application deployment to detect anomalies. The monitoring data is also used to update and adapt the models.

4.8.3 Features

- **Alternative deployment configuration selection**
    - A machine learning based framework to build the predictive models for predicting the performance of a set of alternative deployment models
    - Select and switch between alternative deployment models at runtime
- **Policy based Deployment Adaptation**
    - A high-level support for expressing arbitrary deployment adaptation policies
- **Data-Driven Performance Anomaly Detection**

- A dataset of microservice network anomalies
- A machine learning based framework for detecting performance anomalies in microservice deployments.

4.8.4 Status

Overall Architecture of Deployment Refactorer

The overall architecture of the deployment refactoring was published in our *ESOCC 2020 paper*[77]. A detailed discussion of the architecture was included in a journal paper submitted to *Journal of Grid Computing*. A literature review on policy-aware deployment and management of cloud applications was published as a contribution to a journal publication in the Sensors journal[78].

Policy based Deployment Adaptation

We submitted a journal paper on our approach and the initial implementation of the policy-based deployment adaptation support to *Journal of Grid Computing*.

We validated our approach with three key scenarios from the *Vehicle IoT use case*: location-aware redeployment, alert-driven redeployment: Cloud alerts, alert-driven redeployment: Edge alerts. The selected scenarios demonstrated deployment, monitoring, location-aware redeployment, and alert-driven redeployment. Each scenario covers deployment modeling, actual deployment, monitoring, and deployment adaptation. The recorded demonstration videos of the three scenarios are also available in the *GitHub*[79].

Alternative deployment configuration selection

We submitted a journal paper on our approach to building performance models for enabling deployment configuration section to *IEEE Transactions on Services Computing*.

We demonstrated the practicality and feasibility of our proposed approach by applying it to a) an extension of the *RuBiS benchmark*[80] application deployed on *Google's Compute Engine* (92 deployment alternatives, shown in Figure 23), b) *Teastore*[81] microservice benchmark application on *Google Kubernetes Engine* (78 deployment alternatives). The experimental results using the predictive algorithms demonstrated the effectiveness of our proposed approach, i.e., the ability to accurately predict the performance of deployment configurable cloud applications.

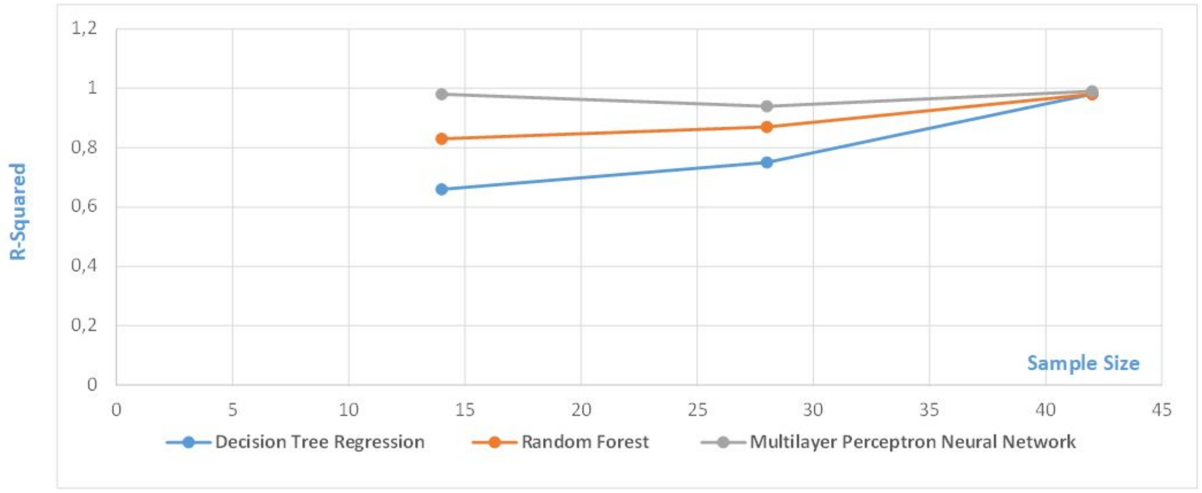


Figure 23 - performance of the models with respect to the sample size used for training

Figure 23 - shows the performance of the models with respect to the sample size used for training the models (for the RuBiS benchmark application). From the experimental results, the first

observation is that all three models perform better with model tuning. The second observation is that all three models show improvement in their performance with iterative model training using sampling. DTR, RFR and MLP-NN return a R2 score of 98%, 96% and 99% respectively. Most importantly, the results confirm that our approach is feasible by demonstrating that it is possible to learn a good function that can predict the performance of unseen deployment variants based on training using the performance data of a subset of the valid deployment variants (14-28 of the 93 valid variants).

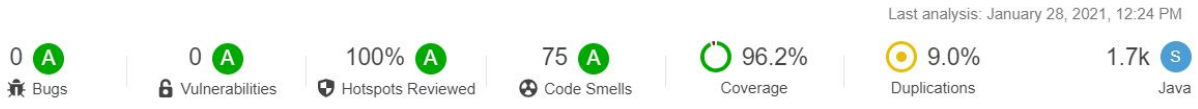
Data-Driven Performance Anomaly Detection

- We have started to experiment with a recent microservice memory anomaly dataset[63] using three predictive algorithms *Random Forest, Decision Tree and Deep Learning with AdaBoost*.
- We have started to collect a service network anomaly dataset with *TeaStore microservice benchmark*[82], *Google Kubernetes Engine*, and SODALITE *SkyDive* monitoring support.

4.8.5 Code quality

This component is written in Java and Python, and it is integrated into *SonarCloud* for quality assessment and obtained the following quality score for each sub project in the *Github* repository. Note that the extended RUBiS benchmark application is implemented in PHP, which is not included in the *SonarCloud* reports.

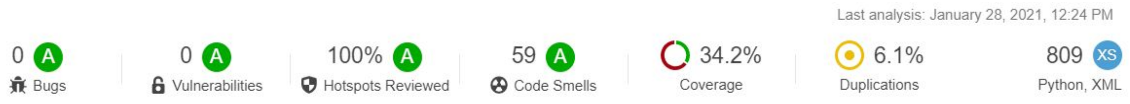
- **Policy-based Deployment Adaptation (Java):** The Java-based sub-component has 96.2% code coverage, 75 code smells and 9.0% code duplications. In the next releases we will focus on reducing the code smells, removing duplicate code by refactoring the code base, and improving code coverage by adding more unit tests.


Last analysis: January 28, 2021, 12:24 PM

0 A	0 A	100% A	75 A	96.2%	9.0%	1.7k S
Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Java

Figure 24 - Code quality report for Policy-based Deployment Adaptation

- **Machine Learning Pipelines (Python):** The Python-based sub-component has 34.2% code coverage, 59 code smells and 6.1% code duplications. In the next releases we will focus on reducing the code smells, removing duplicate code, and improving code coverage.


Last analysis: January 28, 2021, 12:24 PM

0 A	0 A	100% A	59 A	34.2%	6.1%	809 XS
Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Coverage	Duplications	Python, XML

Figure 25 - Code quality report for Machine Learning Pipelines

4.8.5 Next steps

- **Policy based Deployment Adaptation**
  - Improve the policy language to support all deployment adaptation scenarios of SODALITE use cases
  - Improve the integration between the policy engine and other components (e.g, *Deployment Preparation API* and *Node Manager*).
- **Alternative Deployment Configuration Selection**
  - Model the deployment configuration selection as an optimization problem that maximizes the overall utility of the system while considering cost/performance tradeoffs.
  - Support support all deployment model switching scenarios of SODALITE use cases.
  - Changes in the infrastructure (e.g., adding new resources or retirement of existing resources) change the deployment options, which in turn alters the number and importance of the features used by the machine learning (ML) models. Hence, the different versions for the ML models should be maintained. Also, the accuracy and performance of the machine learning models can decrease overtime. Thus, the models need to be monitored for their performance and restrained as appropriate. The machine pipeline for building performance prediction models needs to be improved by applying MLOps (Machine Learning Operations) principles and techniques.
- **Data-Driven Performance Anomaly Detection**
  - Complete the collection of service network anomaly dataset and the implementation of the ML models for anomaly detection
  - Complete the integration with the policy based adaptation support

4.9 Node Manager Refactoring

Component Node Manager is in charge of managing and optimizing the usage of existing resources allocated by the SODALITE users or by other SODALITE components (e.g., *Deployment Refactorer*). The *Node Manager oversees the execution of multiple concurrent applications deployed on a shared cluster of* virtual or physical machines using containers. The cluster is assumed to be heterogeneous meaning that nodes can be equipped with both CPUs and GPUs. The goal of the *Node Manager* is to fulfil given application requirements on the response time while minimizing resource usage.

4.9.1 Innovation

The main innovation of *Node Manager* lies in its ability to provide a coordinated management of heterogeneous resources through both smart load balancing and fine-grained resource management. In heterogeneous clusters traditional cloud-based solutions[83] [84], cannot be reused because they do not consider the heterogeneity introduced by GPUs but, usually, only different VM configurations. CPUs and GPUs are interdependent resources while different virtual machines are not. GPUs are more computationally powerful than CPUs but they also use CPUs to load and write data, and to be activated. Moreover, they also have different scaling capabilities: CPUs can be scaled by allocating with high precision quotas of cores; GPUs can only be time-shared among different applications. Solutions that combine the management of CPUs and GPUs[85] [86] [87] mostly focus only on Machine Learning training, and only exploit scheduling and load balancing algorithms and not dynamic resource provisioning[88].

An architecture for the deployment and management of applications that can be run on a heterogeneous infrastructure

Node Manager facilitates the initial deployment and runtime management of applications with the capabilities of running on a heterogeneous infrastructure (e.g., Machine Learning applications). Users just have to provide a TOSCA blueprint with metadata regarding container information (e.g., container image) and the initial amount of resources to allocate for each application. *Node Manager* automatically deploys the application in a set of containers that are optimized for the runtime control.

Node Manager exploits three level of controls: a centralized load balancing algorithm that efficiently schedules requests to CPUs or GPUs according to applications' needs, a *Supervisor* on each cluster node to solve local resource contention scenarios, and one *CTController (*based on control theory*)* for each instance of application deployed on each node to continuously optimize CPU allocation (vertical scalability).

A smart load-balancing algorithm for heterogeneous resources

Node Manager provides a centralized load balancer that exposes APIs that can be exploited by users to utilize their applications. Users' requests are temporarily stored in dedicated queues (one for each application). *Node Manager* uses two schedulers to extract requests from queues and execute them on fast GPUs or CPUs.

GPUScheduler extracts requests from the queue of the application with the greatest difference between expected and measured performance to boost executions. *CPUScheduler* works together with *CTControllers.* It uses a fair round-robin policy and forward requests to available CPU devices. Locally at the node level, *CTControllers* accelerate or decelerate the executions by continuously modifying the CPU cores allocated to containers. When GPUScheduler schedules a request for GPU execution, the average response time of the application significantly decreases. In this case, *CTControllers* react quickly to this sudden change by decreasing the number of allocated CPU cores. Note that allocated cores could not be lowered even when GPUs are used because of other exogenous factors (e.g., workload fluctuations).

Control-theoretical planners and Supervisor for the coordinated management of CPU resources considering GPU utilization

Node Manager optimizes the scheduling of the centralized load balancer by dynamically reconfiguring running containers (vertical scalability). Given N application to deploy on a node with G GPUs, *Node Manager* deploys, N containers for CPU executions, G containers containing all the application for GPU executions, and one container running all the *CTControllers* (one per application) and the *Supervisor*. This deployment allows the independent management of CPU cores and memory for each application since they run segregated. Moreover, it allows to separate CPU executions from GPU ones and to share a single GPU among different applications (at the time of writing it is technically not possible to mount the same GPU onto two containers).

CTControllers are implemented as PI controllers and they are set to have a control period of 1 second. Each *CTController* oversees the execution of a single container (among the ones dedicated to CPU executions), it computes the optimal resource allocation to reach a given set-point, considering the workload and the utilization of the GPU. Before enacting the resource allocation, the computed value is passed to the *Supervisor* that gathers all the computations of *CTControllers* at each control period. If the sum of the allocations is greater than the available cores *Supervisor* downscales the allocations using a configurable heuristic (the default strategy is proportional). After the *Supervisor* computes the allocations, the state of *CTControllers* is updated and the allocation quickly enacted using *cpu-quotas* (vertical scalability).

4.9.2 Architecture

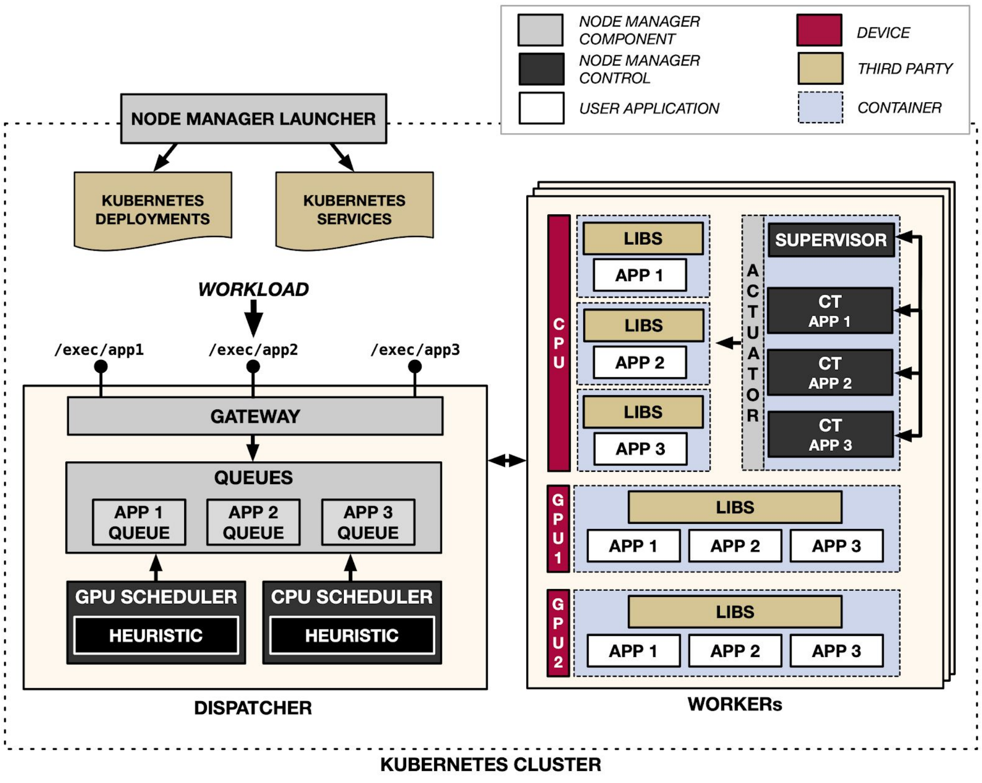


Figure 26 - Node Manager Architecture

Node Manager exploits a distributed architecture and a hierarchical control-strategy as shown in Figure 26 where three applications are in execution. *Node Manager* exploits Kubernetes to ease application deployment. Users can submit applications descriptions using TOSCA blueprints to component *Node Manager Launcher.* This component automatically generates Kubernetes Deployment and Service files and deploys the applications as described in the previous section. The deployment step returns a set of endpoints that are exposed by component *Gateway* and that can be used by the user to submit requests to each application. Requests are stored in queues so that the *GPU* and *CPU Schedulers* prioritize the executions according to application needs. In the distributed nodes (i.e., workers) applications along with dependency libraries are deployed onto containers. Moreover a special container called *Actuator* wraps all the *CTControllers* (CT in the figure) and the *Supervisor. Actuator* exploits Docker-out-of-Docker to reconfigure containers at runtime without the need of centralized updates. This enables short control periods (e.g., 1 second).

4.9.3 Features

The main feature of *Node Manager* is the automated resource management for heterogeneous infrastructures. *Node Manager* is able to keep the response time of multiple concurrent applications under control (i.e., below the SLA). It coordinates smart load balancing and resource allocation in order to optimize the usage of GPUs and CPUs. Moreover, *Node Manager* automates the deployment of the applications using standard TOSCA inputs.

4.9.4 Status

A prototype of *Node Manager* was implemented and evaluated on the Azure public cloud using four benchmark applications: *Skyline Extraction* from Snow UC, *ResNet*, *GoogLeNet*, and *VGG16*. *Node Manager* was compared with a rule-based system obtaining overall 96% fewer SLA violations while using 15% fewer resources. A detailed description of the results can be found in deliverable D6.3 at Section 5.2.

4.9.5 Code quality

The component is written in Python and it is integrated in *SonarCloud* for quality assessment and obtained the following quality scores as depicted in Figure 27.

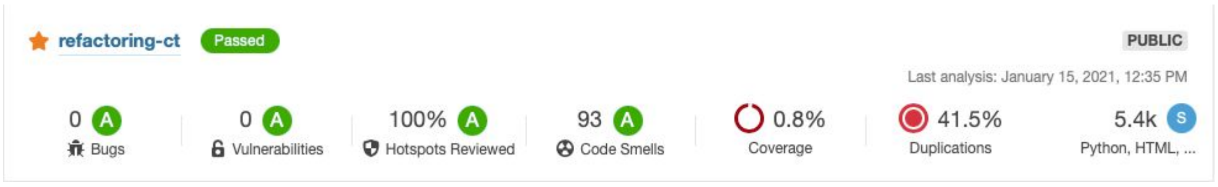


Figure 27 - Code quality report for Node Manager

4.9.5 Next steps

- Better integration with the SODALITE infrastructure (monitoring)
- Integration with *Deployment Refactorer*

4.10 Refactoring Option Discoverer

A refactoring option represents one or more nodes in a deployment model/topology. The objective of *Refactoring Option Discoverer* is to discover new refactoring options as well as the changes to existing refactoring options. *Deployment Refactorer* uses *Refactoring Option Discoverer* to find the alternative nodes to be used for adapting the current deployment model.

4.10.1 Innovation

TOSCA Compliant Resource Discovery using Semantic Matchmaking.

While there exist many studies on the discovery and composition of services and resources, only a few recent studies have considered discovery and composition of standardized resources, for example, TOSCA based compute nodes[89]. These approaches are also limited to matching between the requirements and capabilities of nodes. In particular, the matching is mainly based on syntactic properties, and there is also a lack of consideration for policies attached to resources (e.g., TOSCA policies, which govern use or access to the resources). We use 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. The semantic annotation of resource models including the attached policies enables machine reasoning which is then used for both the discovery and the composition of resources.

4.10.2 Architecture

Figure 28 shows the design of our support for discovering TOSCA compliant nodes (resources) using semantic matchmaking. *SODALITE KnowledgeBase* (WP3) includes the semantic descriptions (ontological representations) of all the resources in the infrastructure. The *knowledgebase* can be populated by the resource experts manually. If there exists TOSCA-based descriptions of the resources, then, such descriptions can be automatically translated to the corresponding ontological representations using the APIs provided by the *knowledgebase*. At runtime, *Platform Discovery Service* (WP4) can automatically discover dynamically added resources to the infrastructure, create their ontological representations, and update the *knowledgebase*. *Deployment Option Discoverer* performs matchmaking by executing the SPARQL queries over the ontologies in the knowledgebase. It provides a high-level system support to the *Deployment Refactorer* to allow searching for resources, for example, *find (a logical expression over node properties). Deployment Option Discoverer* has the SPARQL query templates for different types of resource matchmakings. The query templates are instantiated with the input data received through the high-level API operations. Figure 29 shows a snippet of the SPARQL Query generated for retrieving nodes matching the constrinant *flavor = "m1.small" && image = "centos7".*

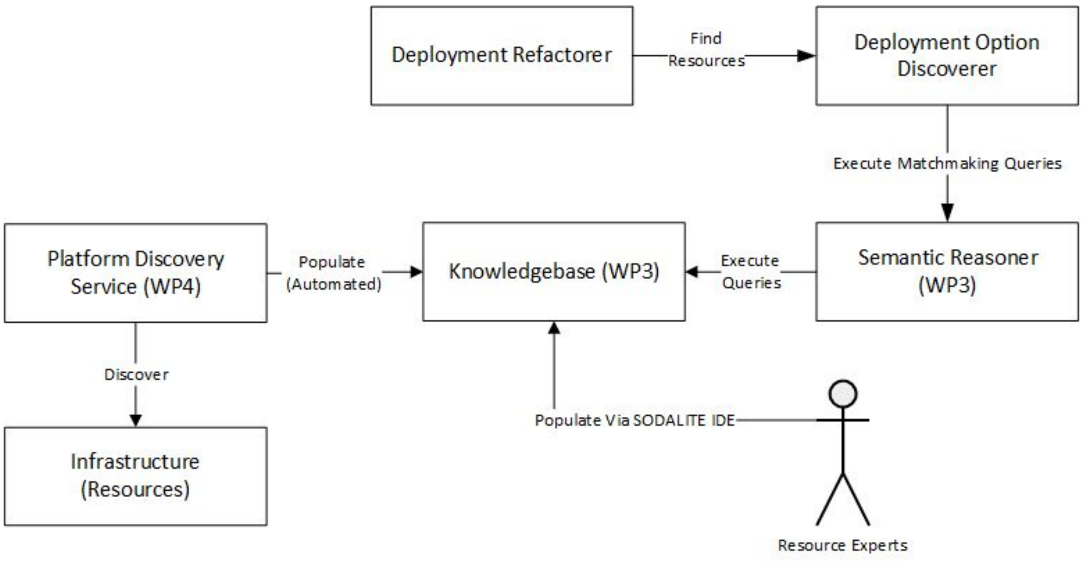
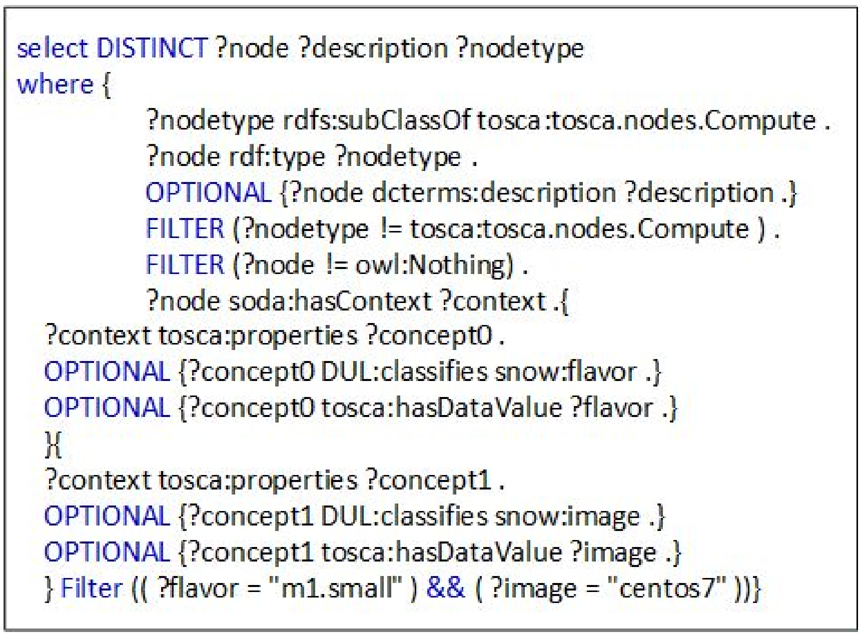

Deployment Refactorer
Find Resources
Deployment Option Discoverer
Execute Matchmaking Queries
Platform Discovery Service (WP4)
Populate (Automated)
Knowledgebase (WP3)
Execute Queries
Semantic Reasoner (WP3)
Discover
Infrastructure (Resources)
Populate Via SODALITE IDE
Resource Experts

Figure 28 - High-level Architecture of Dynamic Discovery and Use of Resources



```
select DISTINCT ?node ?description ?nodetype
where {
            ?nodetype rdfs:subClassOf tosca:tosca.nodes.Compute .
            ?node rdf:type ?nodetype .
            OPTIONAL {?node dcterms:description ?description .}
            FILTER (?nodetype != tosca:tosca.nodes.Compute ) .
            FILTER (?node != owl:Nothing) .
            ?node soda:hasContext ?context .{
  ?context tosca:properties ?concept0 .
  OPTIONAL {?concept0 DUL:classifies snow:flavor .}
  OPTIONAL {?concept0 tosca:hasDataValue ?flavor .}
  }{
  ?context tosca:properties ?concept1 .
  OPTIONAL {?concept1 DUL:classifies snow:image .}
  OPTIONAL {?concept1 tosca:hasDataValue ?image .}
  } Filter (( ?flavor = "m1.small" ) && ( ?image = "centos7" ))}
```

Figure 29 - A Snippet of the SPARQL Query Generated for Retrieving Nodes Matching the Constrinant flavor = "m1.small" && image = "centos7"

4.10.3 Features

The semantic matching capabilities provided by this component include:

- Matchmaking based on TOSCA node properties

The resources (nodes) have various properties, e.g., flavor and image of a VM node. Such properties can be used to select the desired resources.

- Matchmaking based on TOSCA node capabilities and requirements

  The resources (nodes) can also specify their capabilities and their requirements, e.g., the capability to host a database engine. Such capabilities/requirements can be used to select the desired resources.
- Matchmaking based on TOSCA policies

  The resources (nodes) can also specify the policies that determine deployment, access, and management of the resources, for example, auto scaling policy or placement policy. The resource policies can be used to select the desired resources.

4.10.4 Status

- **Resource matchmaking based on TOSCA node properties**

  The logical expressions on the constraints on node properties can be used to select the nodes. For example, a VM with 'CentOs' image and 'small' flavor can be discovered.
- **Resource matchmaking based on TOSCA node capabilities and requirements**

  The nodes are discovered by matching the requirements of the source node with the capabilities provided by the candidate target nodes. For example, a database node that is capable of hosting a *MySQL database* can be discovered.
- **Resource matchmaking based on TOSCA policies**

  We support the policies that are either evaluated to true or false based on the values provided by one or more properties of the policy. For example, a node that will be only deployed on the data centers in Germany or Italy can be discovered by checking if the node has a placement policy that enforces the desired deployment constraint.

4.10.5 Code quality

This module is written in Java, and it is integrated in *SonarCloud* for quality assessment and obtained the following quality score : 90.2% code coverage, 20 code smells and 0.0% code duplications. In the next releases we will focus on reducing the code smells, removing duplicate code by refactoring the code base, and improving code coverage by adding more unit tests.

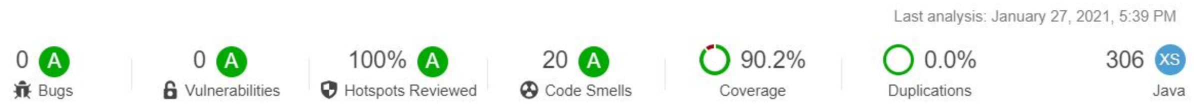


Figure 30 - Code quality report for Refactoring Option Discoverer

4.10.5 Next steps

- *Improved matchmaking capabilities*

  The current matchmaking capabilities support some dynamic resource discovery requirements of SODALITE case studies. Those capabilities need to be extended to support all the resource discovery requirements.
- *Improved the Integration with the Deployment Refactorer*

  All matchmaking capabilities need to be exposed as REST API operations and *Deployment Refactorer* needs to be able use each matchmaking capability.

5 Extension of the existing components (Atos, All)

This section presents the current status of the components already included within the M12 release of the *Runtime Layer* [D5.1]. For each component, it is described the improvements incorporated since the last release, the analysis of the code quality and its planned development steps for next releases (M30 and M36).

5.1 Orchestrator

SODALITE uses *xOpera*, the lightweight TOSCA orchestrator compliant with TOSCA Simple Profile in YAML Version 1.3[90], to provide provisioning, configuration and application deployment through TOSCA/Ansible blueprints. SODALITE further improved the usability of *xOpera orchestrator* providing a REST API wrapper, described in the subsection 5.1.2. In this section the most important improvements on both the *xOpera orchestrator* and the REST API are described in more detail.

5.1.1 xOpera

5.1.1.1 Improvements

During the second year of the project many changes and improvements were introduced in the *xOpera orchestrator*. The *xOpera orchestrator* continued on its path to support the latest TOSCA standard developments and implementations. Many of the added features present a novelty and improve the projects where xOpera is used for orchestration over different and heterogeneous environments. Many of the new features represent architectural improvements like enabling *xOpera* to be used as a Python API library while some of the added features represent a novel approach to TOSCA deployments. In the next sections we will only focus on the most prominent and innovative features implemented based on SODALITE requirements to support the execution of SODALITE proofs of concepts (PoC) and real world application deployment scenarios introduced by SODALITE use case demonstrators.

Resume Command: The *xOpera* resume command was introduced to support a broken execution topology deployment workflow. In real world scenarios the execution of complex application deployments over different heterogeneous platforms can often break for many different reasons - such as resource unresponsiveness, faulty configuration introduced based on human errors or configuration drift, etc. In many cases resuming the deployment from a faulted node in a provided blueprint helps to successfully deploy the entire blueprint, especially in a large topology when the execution of the previous successfully executed nodes could be lengthy and costly.

During the deployment of application components, failures might occur, interrupting the deployment progress. For example, an installation of a software package has failed after the provisioning of a virtual machine or an element of a workflow execution has failed after successful execution of previous elements. In this case, the *Orchestrator* should provide functionality to resume the deployment progress from the failed component, as it becomes costly and error-prone to recover the deployment from the beginning.

A resume option for *xOpera* was implemented allowing to restart the deployment from the point of failure. Additionally, if resume behavior is not preferred, a clean state option can be selected that will clear the deployment progress and start over. The following presents an example.

Consider this topology template. The property "success" of TOSCA node type "SuccessOrFailure" defines whether the component deployment will be failed. In this case, the failure will be triggered for component_2, as shown in Figure 31:

```
topology_template:
  node_templates:

      component_1:
        type: SuccessOrFailure
        properties:
          success: true

      component_2:
        type: SuccessOrFailure
        properties:
          success: false

      component_3:
        type: SuccessOrFailure
        properties:
          success: true
```

Figure 31 - Sample deployment TOSCA blueprint showing failure

If we try to deploy this template, an error will occur as shown in Figure 32:

```
$ opera deploy fail.yaml
[Worker_0]   Deploying component_1_0
[Worker_0]    Executing create on component_1_0
[Worker_0]   Deployment of component_1_0 complete
[Worker_0]   Deploying component_2_0
[Worker_0]    Executing create on component_2_0
[Worker_0] ------------
... Failure output
[Worker_0] ------------
[Worker_0] ============
```

Figure 32 - *xOpera* output showing the run of the failed execution of the TOSCA blueprint

In order to resume the deployment, the "success" property is set to "true" boolean value. As it can be seen in Figure 33 , the deployment started from the previously failed component.

```
$ opera deploy --resume fail.yaml
The resume deploy option might have unexpected consequences on the already
deployed blueprint.
Do you want to continue? (Y/n): Y
[Worker_0]   Deploying component_2_0
[Worker_0]    Executing create on component_2_0
[Worker_0]   Deployment of component_2_0 complete
[Worker_0]   Deploying component_3_0
[Worker_0]    Executing create on component_3_0
[Worker_0]   Deployment of component_3_0 complete
```

Figure 33 - xOpera output showing the run of the failed execution of the TOSCA blueprint

Parallel execution of deployment: SODALITE is a framework that enables the user to easily define and use heterogeneous resources and deploy complex applications on those resources with a special focus on performance optimization. SODALITE supports two different ways of application deployment optimizations. The first is static, initiated at deployment time based on selected target resources and application optimization parameters. The second one is a dynamic - runtime optimization, based on the possible redeployment choices the platforms offer to the user. In both cases the orchestrator deploys and executes a possibly complex application deployment blueprint over several infrastructures managing many different infrastructure configurations and application artifacts. Ansible as probably the first choice over the configuration tools used by DevOps teams, unfortunately, poorly supports parallelization of execution workflows which is used as the main executor in *xOpera*. SODALITE has upgraded xOpera to execute the deployment in a parallelized way resulting in substantial decrease of the time needed to deploy a complete complex application.

The *xOpera orchestrator* is able to execute a TOSCA/ansible blueprint in a parallelized way through the *-w* switch in the execution of the deploy and undeploy command. The implementation uses a pool of *w* threads that are assigned on a per need basis regarding the node dependencies.

Consider an application deployment topology depicted in Figure 34. The topology consists of many inter-dependent nodes typical for an application deployment, especially a job execution workflow on a HPC cluster where results of a job may depend on the calculation of another successfully executed job.
The service template used in the example can be found in the xOpera github repository under concurrency integration tests[91].

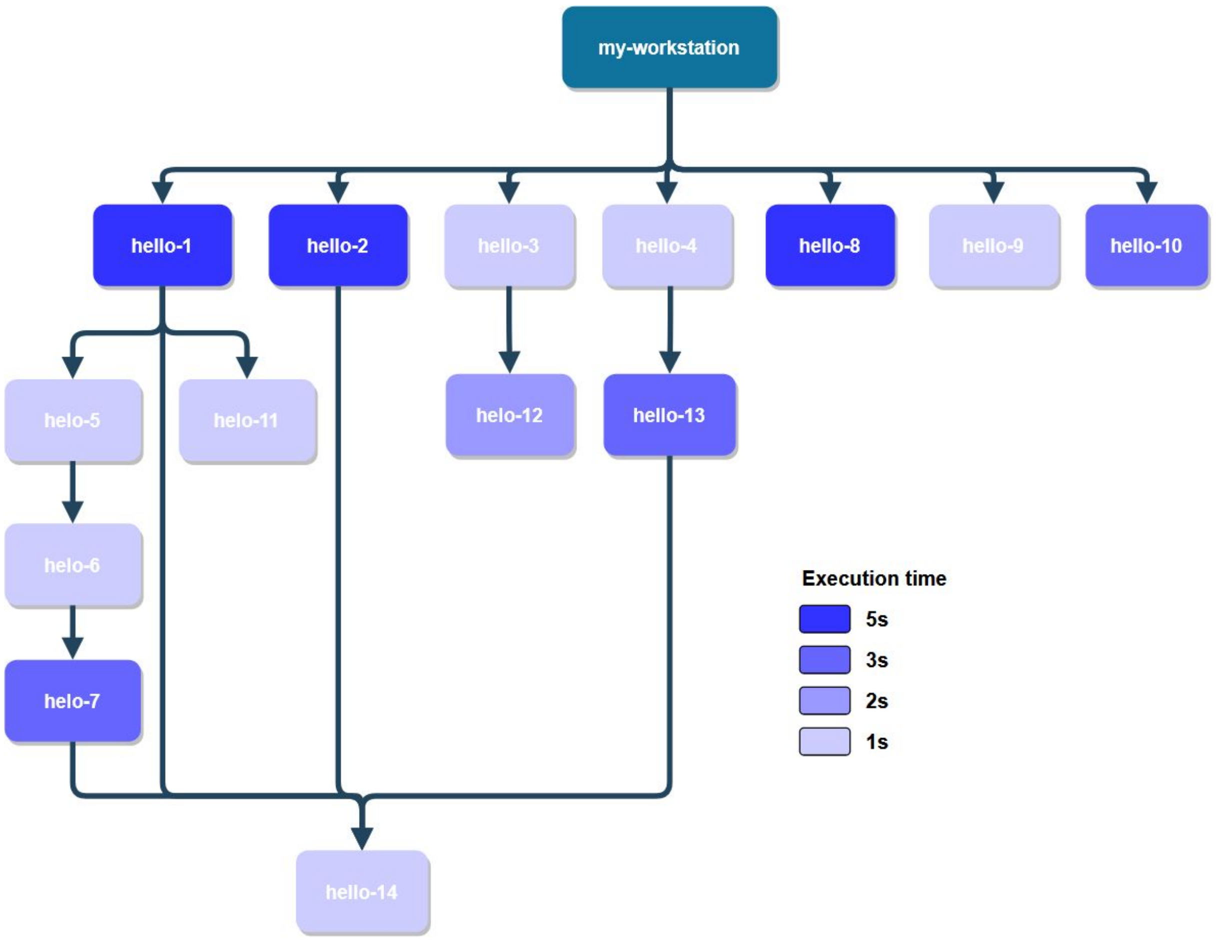


Figure 34 - Sample TOSCA template node topology used for testing concurrent deployment

The standard single threaded deployment is shown in Figure 35:

```
(.venv) ➜  concurrency git:(master) ✗ time opera deploy service.yaml
[Worker_0]   Deploying my-workstation_0
[Worker_0]   Deployment of my-workstation_0 complete
[Worker_0]   Deploying hello-1_0
[Worker_0]     Executing create on hello-1_0
[Worker_0]     Executing start on hello-1_0
[Worker_0]   Deployment of hello-1_0 complete
[Worker_0]   Deploying hello-2_0
[Worker_0]     Executing create on hello-2_0
[Worker_0]     Executing start on hello-2_0
[Worker_0]   Deployment of hello-2_0 complete
[Worker_0]   Deploying hello-3_0
[Worker_0]     Executing create on hello-3_0
[Worker_0]     Executing start on hello-3_0
[Worker_0]   Deployment of hello-3_0 complete
[Worker_0]   Deploying hello-4_0
[Worker_0]     Executing create on hello-4_0
[Worker_0]     Executing start on hello-4_0
[Worker_0]   Deployment of hello-4_0 complete
[Worker_0]   Deploying hello-5_0
[Worker_0]     Executing create on hello-5_0
[Worker_0]     Executing start on hello-5_0
[Worker_0]   Deployment of hello-5_0 complete
[Worker_0]   Deploying hello-6_0
[Worker_0]     Executing create on hello-6_0
[Worker_0]     Executing start on hello-6_0
[Worker_0]   Deployment of hello-6_0 complete
[Worker_0]   Deploying hello-7_0
[Worker_0]     Executing create on hello-7_0
[Worker_0]     Executing start on hello-7_0
[Worker_0]   Deployment of hello-7_0 complete
[Worker_0]   Deploying hello-8_0
[Worker_0]     Executing create on hello-8_0
[Worker_0]     Executing start on hello-8_0
[Worker_0]   Deployment of hello-8_0 complete
[Worker_0]   Deploying hello-9_0
[Worker_0]     Executing create on hello-9_0
[Worker_0]     Executing start on hello-9_0
[Worker_0]   Deployment of hello-9_0 complete
[Worker_0]   Deploying hello-10_0
[Worker_0]     Executing create on hello-10_0
[Worker_0]     Executing start on hello-10_0
[Worker_0]   Deployment of hello-10_0 complete
[Worker_0]   Deploying hello-11_0
[Worker_0]     Executing create on hello-11_0
[Worker_0]     Executing start on hello-11_0
[Worker_0]   Deployment of hello-11_0 complete
[Worker_0]   Deploying hello-12_0
[Worker_0]     Executing create on hello-12_0
[Worker_0]     Executing start on hello-12_0
[Worker_0]   Deployment of hello-12_0 complete
[Worker_0]   Deploying hello-13_0
```

```
[Worker_0]    Executing create on hello-13_0
[Worker_0]    Executing start on hello-13_0
[Worker_0]  Deployment of hello-13_0 complete
[Worker_0]  Deploying hello-14_0
[Worker_0]    Executing create on hello-14_0
[Worker_0]    Executing start on hello-14_0
[Worker_0]  Deployment of hello-14_0 complete
opera deploy service.yaml  63.72s user 3.26s system 53% cpu 2:04.55 total
```

Figure 35 - Sample single threaded deployment of the TOSCA template

Resulting in an average deployment time of **126.24 s** on 100 runs.
Sample execution with a pool of 10 workers:

```
(.venv) ➜  concurrency git:(master) ✗ time opera deploy -w 10 service.yaml
[Worker_0]  Deploying my-workstation_0
[Worker_0]  Deployment of my-workstation_0 complete
[Worker_1]  Deploying hello-1_0
[Worker_2]  Deploying hello-2_0
[Worker_3]  Deploying hello-3_0
[Worker_3]    Executing create on hello-3_0
[Worker_4]  Deploying hello-4_0
[Worker_1]    Executing create on hello-1_0
[Worker_2]    Executing create on hello-2_0
[Worker_0]  Deploying hello-9_0
[Worker_5]  Deploying hello-8_0
[Worker_7]  Deploying hello-10_0
[Worker_6]  Deploying hello-14_0
[Worker_4]    Executing create on hello-4_0
[Worker_0]    Executing create on hello-9_0
[Worker_6]    Executing create on hello-14_0
[Worker_5]    Executing create on hello-8_0
[Worker_7]    Executing create on hello-10_0
[Worker_4]    Executing start on hello-4_0
[Worker_6]    Executing start on hello-14_0
[Worker_0]    Executing start on hello-9_0
[Worker_3]    Executing start on hello-3_0
[Worker_7]    Executing start on hello-10_0
[Worker_5]    Executing start on hello-8_0
[Worker_1]    Executing start on hello-1_0
[Worker_2]    Executing start on hello-2_0
[Worker_6]  Deployment of hello-14_0 complete
[Worker_4]  Deployment of hello-4_0 complete
[Worker_8]  Deploying hello-13_0
[Worker_8]    Executing create on hello-13_0
[Worker_0]  Deployment of hello-9_0 complete
[Worker_3]  Deployment of hello-3_0 complete
[Worker_6]  Deploying hello-12_0
[Worker_6]    Executing create on hello-12_0
[Worker_7]  Deployment of hello-10_0 complete
[Worker_1]  Deployment of hello-1_0 complete
[Worker_4]  Deploying hello-5_0
[Worker_9]  Deploying hello-11_0
[Worker_9]    Executing create on hello-11_0
[Worker_4]    Executing create on hello-5_0
[Worker_5]  Deployment of hello-8_0 complete
[Worker_6]    Executing start on hello-12_0
[Worker_2]  Deployment of hello-2_0 complete
[Worker_8]    Executing start on hello-13_0
[Worker_9]    Executing start on hello-11_0
[Worker_4]    Executing start on hello-5_0
```

```
[Worker_6]   Deployment of hello-12_0 complete
[Worker_8]   Deployment of hello-13_0 complete
[Worker_9]   Deployment of hello-11_0 complete
[Worker_4]   Deployment of hello-5_0 complete
[Worker_0]   Deploying hello-6_0
[Worker_0]     Executing create on hello-6_0
[Worker_0]     Executing start on hello-6_0
[Worker_0]   Deployment of hello-6_0 complete
[Worker_3]   Deploying hello-7_0
[Worker_3]     Executing create on hello-7_0
[Worker_3]     Executing start on hello-7_0
[Worker_3]   Deployment of hello-7_0 complete
opera deploy -w 10 service.yaml  168.19s user 10.85s system 335% cpu 53.413 total
```

Figure 36 - Sample multithreaded deployment of the TOSCA template using a pool of 10 threads

The sample run shown in Figure 36 shows the usage of distribution of an application deployment execution among 10 workers resulting in an average deployment time of **54.712** s on 100 runs.

These results are very promising and bring a potentially real reduction in deployment time and consequently costs, knowing that deployment and execution time and costs usually have a linear correlation.
When applying this solution to a cloud scenario combined with the possibility of multiple reconfigurations through redeployment the potential reduction of cost compounds.
The usage of the parallel TOSCA deployment execution in SODALITE adds a third performance optimization dimension to the existing static and dynamic application deployment optimizations.

Blueprint deployment reconfiguration: TOSCA blueprints are defined as a topology of inter-dependent nodes with each of the nodes implementing its own life-cycle operations. TOSCA offers the means and definition for an application deployment, possibility of scaling through policies and reconfiguration using substitution nodes. However in a real world scenario a DevOps team creates a TOSCA deployment blueprint (TB1) definition with infrastructure resource provisioning and configuration typically associated with an application deployment and it's configuration (I1). Usually in a lifetime of a deployed instance (DI), applications need updates, restarts or some kind of maintenance not easily identified during the TOSCA model creation. In this case the only option for a DevOps team is to create an updated version of the TOSCA blueprint (TB2) usually with some configuration defined through an inputs file (I2) to substitute the existing deployed instance (DI), but only after issuing the *undeploy* command through the orchestrator which starts to execute the TOSCA undeploy workflow. In a standard TOSCA undeployment scenario the undeployment of the whole TOSCA blueprint is executed to tear down the existing deployed instance (D1).
What if the user wants to change only a small part of the blueprint, for instance to update the version of a running application on one of the nodes? Undeployment of the whole instance (D1=TB1 | I1) and after that deploying of the reconfigured instance (D2=TB2 | I2) can take considerable service downtime.

Few TOSCA orchestrators offer this kind of functionality only in a paid tier (e.g. *Yorc*[92]), usually with limited functionality targeting a single node or in some more complex cases a list of nodes (e.g. *Cloudify*[93]).
During the second year of the project the experimental feature enabling the reconfiguration of an existing deployed instance (DI) using a changed TOSCA blueprint (TB2) and a new set of inputs (I2) was implemented. The newly implemented *xOpera* commands *diff* and *Update* offer the *xOpera* user a possibility to first **calculate the difference** between the existing deployed instance (DI) and

the *deployable instance* (D2') enabling the user to **check** what changes will be executed to the running deployed instance (DI) upon the execution of *update,* if applied. After issuing the *update* command, the removal of the nodes not existing in the updated blueprint (T2) is executed, together with changes to the nodes that have a changed set of properties, attributes or implementation interface operations and the deployment/creation of the nodes, that are defined in the updated topology instance (D2).

The implementation of commands *diff* and *update* in *xOpera* introduce the possibility for the user to calculate the difference between two TOSCA blueprints, Instances such as a deployed one and a deployable one D2 derived from a combined use of TB2 and configuration inputs (I2).
The update command tries to execute the deployment of the computed calculation between the two TOSCA topologies as explained in the previous section.
Consider changes in deployment topology depicted in Figure 37. All nodes are hosted on the same workstation that remains unchanged. One node is deleted, one is added, one has 2 changed properties and one remains the same.
Service templates used, in this example, are simplified versions of the templates that can be found in the *xOpera* github repository under compare integration tests[94].

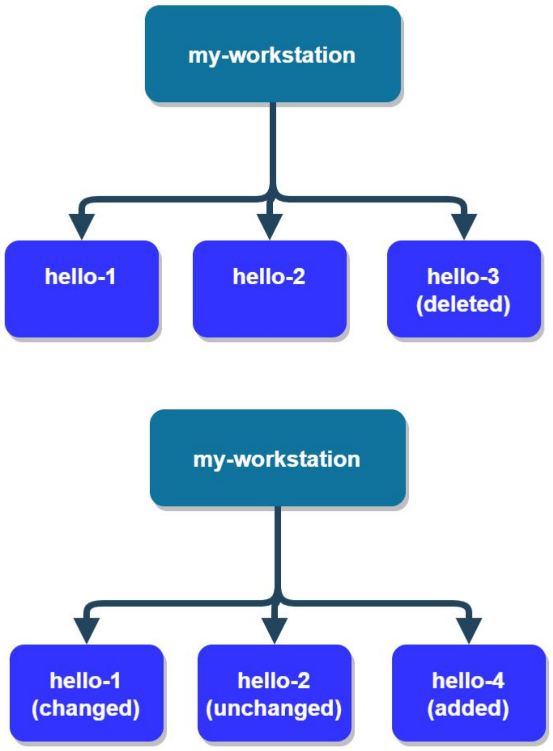


Figure 37 - Sample deployment topology changes

The output of the *diff* command would be:

```
nodes:
  added:
  - hello-4
  changed:
    hello-1:
      capabilities:
        test:
          properties:
            test1:
            - '2'
            - '3'
            test2:
            - '2'
            - '3'
  deleted:
  - hello-3
```

Figure 38 - Sample deployment showing the output of the diff command for topology changes

If the *update* command is executed, *hello-4 node* would be deployed, *hello-3 node* would be undeployed and *hello-1 node* would be undeployed and then deployed.

5.1.1.2 Code Quality

Current version of the *xOpera* orchestrator is 0.6.4 with a new minor version release on its way. The development of *xOpera* is steady and continuous supported by github toolset with added discussions for an easier planning of feature development. The development of *xOpera* is official and done through open discussion of features/contributions and implementation suited for the most open source repositories. Since *xOpera* is an upstream project, not owned by SODALITE, it currently does not use *SonarCloud* for code quality assessment, therefore no Sonarcloud QA can be added here at this point.

5.1.1.3 Next steps

The xOpera orchestrator is bound to be a lightweight TOSCA compliant orchestrator meaning that new TOSCA versions will be supported and features added.

Many new features are planned and described in the *xOpera* github issues[95]. Some of the most important features from SODALITE's point of view being Improving redeployment, adding TOSCA policy enforcement implementation, and improving secret handling.

5.1.2 xOpera REST API

Since the initial version of the *xOpera* REST API described in D5.1, there has been significant refactoring and improvements. The implementation of this component expands beyond just offering a REST API interface to *xOpera* commands as it implements deployment workflows, persists data and information about the deployed TOSCA blueprints by storing sessions for deployed TOSCA blueprints. During the second year of the project, several important upgrades have been made to improve the support for sharing and persisting the blueprint deployed, additional command interfaces were added for newly introduced *xOpera* commands and support for easier transition upgrades between *xOpera* versions added.

5.1.2.1 Improvements

In this section we focus on the most prominent features implemented in the *xOpera* REST in the second year of the project.

The usage of the API was significantly improved with the implementation of the **git plugin** which is able to use external publicly available git services to securely store and version the TOSCA blueprints registered in the API calls. This functionality was further improved by enabling the shared usage of the blueprint between the users, enabling a seamless collaboration through the usage of secure private repositories handled by public git SaaS like **github.com**, **gitlab.com** or private ones like *gitlab.xlab.si*.

xOpera REST API implements the improved CI/CD workflow enabling an automated deployment of the dockerized component on the testbed for staging purposes. In this workflow *xOpera* is used by jenkins to deploy the TOSCA blueprint describing xOpera REST API deployment and configuration on the SODALITE staging testbed.

Since [D5.1] the REST API was refactored to use *xOpera* as Python Library API for execution of deployment, undeployment instead of executing through a subprocess, heavily improving efficiency of the execution.

A new connexion library was introduced instead of *flask-restplus* library, which was not being supported any more, for enabling a simpler API design, OpenAPI 3.0 support, implementation and UI for the REST API calls.

For reverse proxy, *nginx*[96] was changed with *traefik*[97] to ease the usage of the dockerized REST API with simple labeling and to enable a simpler transition to Kubernetes (k8s) deployments. Additionally, in this period, support for both k8s and AWS libraries and dependencies were added to the REST API to support the execution of TOSCA/Ansible playbooks for these platforms.

Important security aspects were improved using IAM authorization for registering and deployment of the blueprints through keycloak token introspection. Additionally secret handling was improved by using Hashicorp Vault to support handling secrets at rest and in the deployment workflows.

xOpera REST API was extended to provide support for *diff* and *update* xOpera commands. Namely, there is a possibility to compare two previously created TOSCA blueprints and reconfigure the deployment of the blueprint according to the rules presented in Section *5.1.2 Improvements*, under *Blueprint deployment reconfiguration.*

xOpera API can be also used as a standalone offering the user a simple and easily understandable way to register, deploy, undeploy, persist data and handle the sharing of TOSCA blueprints.

The code and extensive information on how to build and use the xOpera is provided in the github repository: https://github.com/SODALITE-EU/xopera-rest-api.

5.1.2.2 Code Quality

Since the first xOpera REST API release automatic code quality checks were introduced using the online *SonarCloud* tool. Substantial improvements to the code were applied after enabling the *SonarCloud* code analysis of the xopera-rest-api component. One of the most important being the extension of the code coverage unit tests and reduction of code repetitions.

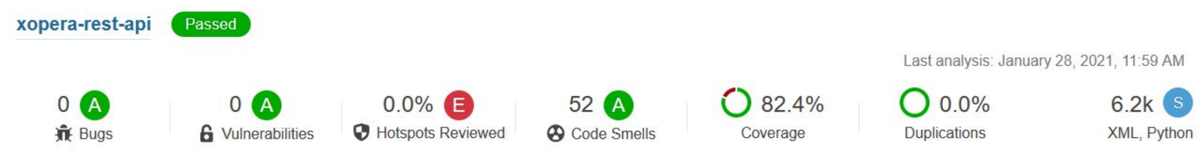


Figure 39 - Code quality report for xopera-rest-api

At the time of this writing the quality of the code in the xOpera REST API github repository was solida, giving some room for improvement on reducing security hotspots and code smells as shown in Figure 39.

5.1.2.3 Next steps

Improving security and integration with the calling components and the ones using xOpera REST API through various workflows is one of the most prominent next steps in the development of this component.
Additionally, adding support for OpenFaaS targeted executions which is planned in year 3 of the project, improved interface covering newly introduced commands in *xOpera* for feature support and the possibility to further improve high availability, possibly through Kubernetes cluster deployment.

5.2 Monitoring

This section describes the updates implemented in the following pre-existing monitoring components (reported in D5.1).

5.2.1 Prometheus/Grafana

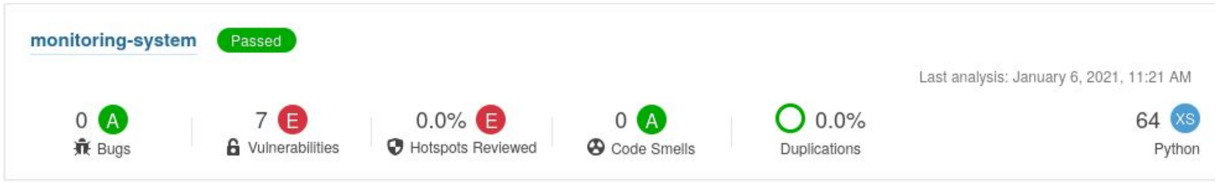
5.2.1.1 Improvements

In the initial version described in [D5.1], a *Prometheus* instance of the SODALITE architecture was deployed in a VM inside the OpenStack cloud testbed. It was prepared to automatically discover and monitor OpenStack resources thanks to a specific configuration hook for that supported by *Prometheus*. However, fixed endpoints for supported exporters had to be previously known and set in the configuration file. This was the case of the *Skydive exporter* included in that initial version. To overcome this rigid monitoring configuration, a dynamic monitoring approach was designed and implemented (see Section 4.3). Moreover, neither alerting nor recording monitoring rule files were supported (see section 4.7).
Additionally, an instance of the *Monitoring dashboard, Grafana*, was deployed within the same VM where *Prometheus* was installed.

In this new M24 release, the *Prometheus* component is declared in a TOSCA blueprint that starts a Docker container using the official *Prometheus* image[98]. A template for the *Prometheus* configuration file[99] is also provided with the blueprint. When the *Prometheus* component is deployed, this template becomes a valid configuration file that connects *Prometheus* to the other components of the monitoring system. Thus, in this intermediate version of the monitoring system, both the *OpenStack* discovery hook and the explicit endpoint references are replaced in the *Prometheus* configuration with the lines needed to make the *Prometheus* instance query *Consul* for any exporter (*node exporters*[100], *IPMI exporters*[101], *HPC exporters*[102], *Skydive exporters*[103]) alive in the computing infrastructure (see Section 4.3). Rule files are retrieved from a folder[104] on which a Docker volume that is shared with the *Rules Server* is mounted. An endpoint to the *Alert Manager*[105] component is also set (see Section 4.7).

5.2.1.2 Code quality

As commented in section 4.2, code quality can neither be reported for *Prometheus* and *Grafana*, as they are third-party components, that are integrated and deployed (currently for *Prometheus*) within the *Runtime Layer* by the *orchestrator*, using TOSCA blueprints and Ansible playbooks, which are not analysed by *SonarCloud*. They are part of the *monitoring-system repository*[106], whose QA report is presented in Figure 40.


monitoring-system Passed

Last analysis: January 6, 2021, 11:21 AM

0 A	7 E	0.0% E	0 A	0.0%	64 XS
Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Duplications	Python

Figure 40 - Code quality report for Monitoring

5.2.1.3 Next steps

Alike *Prometheus*, *Grafana* will be deployed into the *Runtime Layer* by the *orchestrator*, by using the SODALITE TOSCA blueprint and its Ansible playbooks its lifecycle. Moreover, *Grafana dashboard* for target infrastructures and execution environments will be configured based on metrics collected by associated *exporters*. On demand support, testing and bug fixing for automatic configuration and deployment of *Prometheus* within the SODALITE stack, using the *Orchestrator*, will be provided in following development phases.

5.5.2 Node exporter

5.5.2.1 Improvements

As described in section 4.5.2 of [D5.1], the *Node Exporter* is part of the *Prometheus* project and it exposes CPU, memory, network and OS metrics for any machine on which it is installed. The source code in Go is available in its official *Github repository*[107], although the compiled binaries can be downloaded from the download section of the *Prometheus website*[108].

Regarding its integration with the monitoring system, it is still automatically deployed with every new virtual machine created and then added to the monitoring targets. However, the Ansible playbook that creates the VM has been enriched for not only downloading and installing the *Node Exporter* but also for creating the JSON payload[109] that defines the service from a template[110] and registering it in the *Consul* server[111].

5.5.2.2 Code Quality

Node Exporter is shipped within *Prometheus*. See section 5.2.1.2.

5.5.2.3 Next Steps

On demand support, testing and bug fixing for automatic configuration and deployment of the *Node Exporter* in target infrastructures, using the *Orchestrator*, will be provided in following development phases.

5.5.3 IPMI exporter

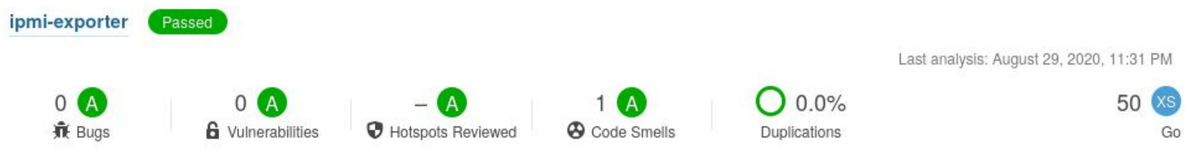
5.5.3.1 Improvements

IPMI Exporter was implemented and integrated for the initial version of the runtime described in D5.1. As section 4.4.1 of D5.1 states, this exporter is designed to expose for *Prometheus* the power consumption metrics of the machine it is running in, achieving this by exporting the measurement given by a physical sensor and obtained via "*ipmi_sensor*" command.

There are no changes in its implementation (*SODALITE-EU/ipmi-exporter*[112] repo in Github) for this intermediate version of the runtime, but now the integration with the monitoring system relies on its registration as a service in *Consul* instead of an explicit endpoint configuration in the *Prometheus* configuration file.

5.1.1.2 Code Quality

The code quality report for *IPMI Exporter* is shown in Figure 41, not manifesting significant quality issues.



ipmi-exporter Passed

Last analysis: August 29, 2020, 11:31 PM

0 A	0 A	– A	1 A	0.0%	50 XS
Bugs	Vulnerabilities	Hotspots Reviewed	Code Smells	Duplications	Go

Figure 41 - Code quality report for IPMI exporter

5.1.1.3 Next Steps

On demand support, testing and bug fixing for automatic configuration and deployment of the *IPMI Exporter* in target infrastructures, using the *Orchestrator*, will be provided in following development phases. Additional IPMI metrics required by use cases will be incorporated.

5.5.4 Skydive exporter

5.5.4.1 Improvements

In the initial version described in [D5.1], the *Skydive exporter* was registered statically in the *Prometheus* configuration file. Additionally, the skydive components were configured and brought up manually.

In the current version, a *Consul* server is used to register exporters that provide metrics to *Prometheus*. Both the *Skydive analyzer* and the *skydive-prometheus connector* were incorporated into the SODALITE platform blueprint to be brought up automatically with the rest of the SODALITE platform, and the *skydive-prometheus connector*[113] registers itself with the *Consul* server to be connected to *Prometheus*. In order to automate the entire operation, the *skydive analyzer* and *skydive-prometheus connector* have to be packaged as containers. The *skydive-prometheus connector* code was hardened, pushed upstream to the *skydive project*, and code was added in the SODALITE tree to create a container in the SODALITE repository.

5.5.4.2 Code Quality

Code Quality for *Skydive* and its extensions cannot be reported, as they are third-party components, and they do not provide the *SonarCloud* statistics.

5.5.4.3 Next Steps

We plan to extend the types of metrics that are collected and reported by the skydive-prometheus exporter. At present, the network metrics are simply reported, but no action is taken based on the network metrics. We aim to be able to recognize some network anomaly or bottleneck, and to be able to recommend a change in the configuration to improve performance.

6 Updated Runtime Layer Development Plan (Atos, All)

This section summarizes the development plan for the incoming M30 and M36 releases of the *Runtime Layer*, broken down into its different components. Additional details for each planned feature can be found in Sections 4 and 5.

6.1 M30 Release

Orchestration	
Component	**Planned features**
xOpera	• Enforcement of TOSCA policies, security, • Improved parser validation output, • REST API support for newly added commands in xOpera, • Improved support for reconfiguration scenarios - update method in interface that would allow not to remove instances
xopera-rest-api	• Support for OpenFaaS, • Improved secret handling, • REST API support for newly added commands in xOpera, • Improved integration with refactoring, • Improved support for Kubernetes, • Support for GoogleCloud
ALDE	• Improve integration with refactoring, Improve container deployment support, Enforce of TOSCA integration, Improve Quality & Tests

Table 3 - Runtime Orchestration Layer Release Plan for M30

Monitoring	
Component	**Planned features**
Prometheus, Consul, Grafana	• Automation of the registering and deregistering processes within the orchestrator: testing full integration with orchestrator, • Automatic configuration of Monitoring dashboards (Grafana) for monitoring exporters and execution targets,
Alert Manager	• Management of subscriptions, • Management of specific monitoring alerts , • Broadcasting of monitoring alerts, • Investigation of Grafana alerting feature
Node Exporter	• Automation of the registering and deregistering processes within the orchestrator: testing full integration with orchestrator
IPMI Exporter	• Automation of the registering and deregistering processes

	within the orchestrator, ● Automation of exporter configuration
HPC Exporter	● Automation of the registering and deregistering processes within the orchestrator: testing full integration with orchestrator, ● Automation of exporter configuration: addressing integration issues related to the lifetime, scope, number and containerization of HPC exporter
Skydive Exporter	● Additional metrics reported; use metrics for feedback to refactor
Edge Monitoring	● Dynamic discovery and registration of platform- and device-specific alerting rules, ● Extension of supported platforms / accelerators.

Table 4 - Runtime Monitoring Layer Release Plan for M30

Refactoring	
Component	**Planned features**
Deployment Refactorer	● Improved the policy language to support all deployment adaptation scenarios of SODALITE use cases, ● Optimization of deployment configuration selection, ● Extensive validation of deployment switching capabilities with all relevant SODALITE Use Cases, ● Service Network Anomaly Dataset and ML models for Service Network Anomaly Detection
Node Manager	● Improved integration with SODALITE infrastructure, ● Exploitation of SODALITE monitoring, ● Allowing deployment of Node Manager with the TOSCA SODALITE deployment blueprint (testing will be carried out on test-bed)
Refactoring Option Discoverer	● Improved Matchmaking based on TOSCA Policies, ● Improved Integration with Deployment Refactorer

Table 5 - Runtime Refactoring Layer Release Plan for M30

6.2 M36 Release

Orchestration	
Component	**Planned features**
xOpera	● Improved handling of secretes through local Ansible Vault , ● Support TOSCA 2.0 simple yaml standard
xopera-rest-api	● Improved support for secure storage handling, REST API support for newly added commands in xOpera

ALDE	• Improve integration with refactoring, • Improve container deployment support, • Enforce of TOSCA integration, • Improve Quality & Tests

Table 6 - Runtime Orchestration Layer Release Plan for M36

Monitoring	
Component	**Planned features**
Prometheus, Consul, Grafana,	• Specialized Monitoring dashboards (Grafana) for target infrastructures and execution environments required by use cases
Alert Manager	• Management of use-case specific monitoring alerts, • Integration of Grafana alerting feature
Node Exporter	• Configure new Node Exporter metrics demanded by use cases
IPMI Exporter	• Implementation of new IPMI metrics required by use cases
HPC Exporter	• Collection of HPC queue metrics for PBSPro and SLURM schedulers
Skydive Exporter	• Selective reporting of metrics
Edge Monitoring	• Hierarchical / Federated Prometheus and Alertmanager instances for node-local and cluster-wide monitoring & alerting.

Table 7 - Runtime Monitoring Layer Release Plan for M36

Refactoring	
Component	**Planned features**
Deployment Refactorer	• Improved Integration of Deployment Refactorer with Node Manager, • Support and Validate Deployment Refactoring Scenarios in all SODALITE Use Cases, • If possible, given the project resource constraints and other commitments, apply MLOps principles for Machine Learning Pipelines (Performance Prediction and Anomaly Detection)
Node Manager	• Integration with Deployment Refactorer for coordinated and improved resource management
Refactoring Option Discoverer	• Support and Validate with Dynamic Resource Discovery and Composition Scenarios in all SODALITE Use Cases

Table 8 - Runtime Refactoring Layer Release Plan for M36

7 Conclusion

This deliverable has reported on the intermediate release (M24) of the SODALITE *Runtime Layer*, emphasizing on the main new features that have been incorporated since the initial release reported in D5.1, but also reporting on the progress achieved on existing components.

The quality of the developed components, their interoperability and the *Runtime Layer* deployment as part of the SODALITE stack have been largely improved by the adoption of methods and toolsets for automatic building and delivery as well as for QA assessment.

The *Orchestration Layer* has incorporated support for the deployment of container-based applications in more Cloud infrastructures, including Kubernetes and AWS. It is also now supporting the deployment of job-based orchestrations in HPC clusters, mediated by schedulers such as SLURM and PBS Pro/TORQUE. Moreover, thanks to the collaboration with the RADON project, the orchestration has incorporated support for multiplatform, hybrid data management, which will be extended to HPC environments in next releases.

The *Monitoring Layer* has been refactored to support dynamic monitoring with the adjustable allocation and configuration of probes in multiplatforms, including now support to monitor jobs in HPC environments. Moreover, support for defining alerting rules and broadcast alerts to subscribers within the SODALITE *Runtime layer* has also been incorporated.

The *Refactoring Layer* adds support to adapt the deployment topology of a running application in response to its anomalous behavior, detected by monitoring. The prediction of the performance of computed deployment alternatives is supported by ML models, which can evaluate the different available deployment variants and switch, at runtime, the selection among them. The dynamic discovery of nodes was improved to support policies. *Node Manage*r was implemented to support runtime resource management, including load balancing, supervision of resource contention, and control of vertical scalability.

The main challenges foreseen in the further development the platform are: a) the complete integration of the *Runtime Layer* within the overall SODALITE stack, providing runtime orchestration, monitoring and refactoring support to the use cases, b) the incorporation of new required infrastructures, such as OpenFaaS and Google Cloud, to the set of supported target deployment environments in orchestration and monitoring, c) the specialization of alerting to detect runtime anomalies for adaptation scenarios, d) the specialization of monitoring dashboards, and e) the support of all redeployment adaptation scenarios. Implementations addressing these challenges will be progressively incorporated into the M30 and M36 releases of the *Runtime Layer*.

8 References

1. D3.1 - First version of ontologies and semantic repository. SODALITE Technical Deliverable 2020.
2. D4.2 - IaC Management - intermediate version. SODALITE Technical Deliverable 2021.
3. D5.1 - Application deployment and dynamic runtime - initial version. SODALITE Technical Deliverable 2020.
4. https://radon-h2020.eu/
5. https://www.oasis-open.org/committees/tosca
6. D1.4 - Management report, second version. SODALITE Technical Deliverable, 2020
7. https://www.jenkins.io/ The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project.
8. D6.3 - Intermediate Implementation and Evaluation of the SODALITE Platform and Use Cases. SODALITE Technical Deliverable 2021.
9. https://sonarcloud.io/ Code Quality assessment online tool Enhance - Your Workflow with Continuous Code Quality
10. https://docs.sonarqube.org/latest/user-guide/security-hotspots/
11. Not Available in M24. They will be available in the next release.
12. http://tango-project.eu/
13. https://github.com/TANGO-Project/alde
14. D2.2 - Requirements, KPIs, evaluation plan and architecture - Intermediate version. SODALITE Technical Deliverable, 2021
15. https://docs.globus.org/
16. https://github.com/onedata/onedata
17. https://lcgdm.web.cern.ch/dynafed-dynamic-federation-project
18. https://github.com/cern-fts/fts3
19. https://github.com/cern-fts/gfal2
20. https://github.com/rucio/rucio
21. https://github.com/rclone/rclone
22. https://github.com/scality/Zenko
23. https://github.com/sodafoundation/multi-cloud
24. https://mqtt.org/
25. https://github.com/eclipse/mosquitto
26. https://github.com/hivemq/hivemq-community-edition
27. https://github.com/apache/kafka
28. https://github.com/fledge-iot/fledge
29. https://github.com/minio/minio
30. https://github.com/apache/nifi
31. https://github.com/streamsets/datacollector
32. https://radon-h2020.eu/
33. https://github.com/radon-h2020/radon-particles
34. Deliverable D5.5 -Data pipeline orchestration I, Public Deliverable, RADON consortium, 2019
35. D7.4 - Impact Generation Report: Year 2. SODALITE Deliverable 2021.
36. https://datapipeline-plugin.readthedocs.io/en/latest/
37. https://prometheus.io/docs/introduction/overview/
38. https://www.consul.io/docs/intro#introduction-to-consul
39. https://www.consul.io/docs/intro#introduction-to-consul
40. https://www.consul.io/docs/k8s/service-sync
41. https://prometheus.io/docs/prometheus/latest/configuration/configuration/#openstack_sd_config
42. https://github.com/SODALITE-EU/monitoring-system/blob/master/consul-registration-poc/openstack/node-exporter/xopera/playbooks/node_exporter.json.tmpl
43. https://github.com/SODALITE-EU/monitoring-system
44. https://github.com/SODALITE-EU/monitoring-system/tree/master/consul-registration-poc/openstack
45. https://github.com/SODALITE-EU/iac-platform-stack/blob/7b65739d4ff5c334654ec9c1848ee6ff131b62ca/docker-local/service.yaml#L639-L650

46.
https://github.com/SODALITE-EU/monitoring-system/blob/master/consul-registration-poc/openstack/node-exporter/xopera/playbooks/vm/create.yml
47. Evans, T., Barth, W. L., Browne, J. C., DeLeon, R. L., Furlani, T. R., Gallo, S. M., ... & Patra, A. K. (2014, November). Comprehensive resource use monitoring for hpc systems with tacc stats. In 2014 First International Workshop on HPC User Support Tools (pp. 13-21). IEEE.
48. Moore, C. L., Khalsa, P. S., Yilk, T. A., & Mason, M. (2015, September). Monitoring high performance computing systems for the end user. In 2015 IEEE International Conference on Cluster Computing (pp. 714-716). IEEE.
49. Externally observing the HPC environment, therefore minimizing the consumption of HPC resources
50. Probes are dynamically configured to target the monitoring of submitted jobs, compatible with the HPC scheduler
51. Röhl, T., Eitzinger, J., Hager, G., & Wellein, G. (2017, September). LIKWID Monitoring Stack: A flexible framework enabling job specific performance monitoring for the masses. In 2017 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 781-784). IEEE.
52. https://github.com/SODALITE-EU/hpc-exporter
53. https://github.com/SODALITE-EU/hpc-exporter/blob/master/docker/Dockerfile
54. https://www.pcwdld.com/best-network-monitoring-tools-and-software
55. https://github.com/skydive-project/skydive-flow-exporter/tree/master/prom_sky_con
56. https://github.com/skydive-project/skydive-flow-exporter
57. https://github.com/skydive-project/skydive-flow-exporter/tree/master/prom_sky_con
58. http://skydive.network/blog/prometheus-connector.html
59. http://skydive.network/blog/exporters.html
60.
https://github.com/SODALITE-EU/iac-platform-stack/blob/7b65739d4ff5c334654ec9c1848ee6ff131b62ca/docker-local/service.yaml#L529-L577
61. https://flask.palletsprojects.com/en/1.1.x/
62. https://gunicorn.org/
63. https://registry.hub.docker.com/r/sodaliteh2020/monitoring-system-ruleserver/tags
64. https://github.com/SODALITE-EU/monitoring-system/tree/master/ruleserver
65. https://github.com/SODALITE-EU/monitoring-system
66. Fowler, Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.
67. Pereira, Juliana Alves, et al. "Learning software configuration spaces: A systematic literature review." arXiv preprint arXiv:1906.03018 (2019).
68. De Lemos, Rogério, et al. "Software engineering for self-adaptive systems: A second research roadmap." Software Engineering for Self-Adaptive Systems II. Springer, Berlin, Heidelberg, 2013. 1-32.
69. N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems,"IEEE TSE, vol. 39, no. 11, pp. 1467–1493, 2013.
70. N. J. Yadwadkaret al., "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach,"inthe 2017 Symposium on Cloud Computing, 2017, pp. 452–465
71. Chandola, Varun, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey." ACM computing surveys (CSUR) 41.3 (2009): 1-58.
72. Lomio, Francesco, et al. "RARE: a labeled dataset for cloud-native memory anomalies." Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation. 2020.
73. Blair, Gordon, Nelly Bencomo, and Robert B. France. "Models@ run. time." Computer 42.10 (2009): 22-27
74. Berger, Thorsten, et al. "A survey of variability modeling in industrial practice." Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems. 2013.
75. https://featureide.github.io/
76. C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel, "The interplay of sampling and machine learning for software performance prediction,"IEEE Software, vol. 37, no. 4, pp. 58–66, 2020.
77. Kumara, Indika, et al. "Quality Assurance of Heterogeneous Applications: The SODALITE Approach." European Conference on Service-Oriented and Cloud Computing (ESOCC 2020), Volume 2. Springer, Cham, 2020 (in print).
78. Kayes, A. S. M., et al. "A Survey of Context-Aware Access Control Mechanisms for Cloud and Fog Networks: Taxonomy and Open Research Issues." Sensors 20.9 (2020): 2464.
79. https://github.com/IndikaKuma/SODALITEDEMOS
80. https://github.com/uillianluiz/RUBiS

81. https://github.com/DescartesResearch/TeaStore
82. Eismann, Simon, et al. "TeaStore: A Micro-Service Reference Application for Cloud Researchers." 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, 2018.
83. L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, “A Discrete- Time Feedback Controller for Containerized Cloud Applications,” in Proceedings of the 2016 24th Int. Symposium on Foundations of Software Engineering. ACM, 2016, pp. 217–228.
84. J.Ding,R.Cao,I.Saravanan,N.Morris,andC.Stewart,“Characterizing Service Level Objectives for Cloud Services: Realities and Myths,” in 2019 IEEE Int. Conf. on Autonomic Computing (ICAC). IEEE, 2019, pp. 200–206.
85. R. Nozal, J. L. Bosque, and R. Beivide, “EngineCL: Usability and Performance in Heterogeneous Computing,” Future Gener. Comput. Syst., vol. 107, pp. 522–537, 2020.
86. Y. N. Khalid, M. Aleem, R. Prodan, M. A. Iqbal, and M. A. Islam, “E-OSched: a load balancing scheduler for heterogeneous multicores,” J. Supercomput., vol. 74, no. 10, pp. 5399–5431, 2018.
87. L. Chen, X. Huo, and G. Agrawal, “Accelerating MapReduce on a coupled CPU-GPU architecture,” in SC Conf. on High Performance Computing Networking, Storage and Analysis, J. K. Hollingsworth, Ed. IEEE/ACM, 2012, pp. 1–11.
88. S. Mittal and J. S. Vetter, “A Survey of CPU-GPU Heterogeneous Computing Techniques,” ACM Comput. Surv., vol. 47, no. 4, pp. 69:1– 69:35, 2015.
89. Brogi, Antonio, and Jacopo Soldani. "Finding available services in TOSCA-compliant clouds." Science of Computer Programming 115 (2016): 177-198.
90.
https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/csprd01/TOSCA-Simple-Profile-YAML-v1.3-csprd01.html
91. https://github.com/xlab-si/xopera-opera/blob/master/tests/integration/concurrency/service.yaml
92. https://github.com/ystia/yorc
93. https://cloudify.co/
94. https://github.com/xlab-si/xopera-opera/tree/master/tests/integration/compare
95. https://github.com/xlab-si/xopera-opera/issues
96. https://nginx.org/en/
97. https://traefik.io/ The Cloud Native Application Proxy - simplify networking complexity while designing, deploying, and operating applications.
98. https://hub.docker.com/r/prom/prometheus
99.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl
100.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl#L20
101.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl#L25
102.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl#L30
103.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl#L35
104.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl#L12
105.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/library/prometheus/playbooks/templates/prometheus.yaml.tmpl#L9
106. https://github.com/SODALITE-EU/monitoring-system
107. https://github.com/prometheus/node_exporter
108. https://prometheus.io/download/#node_exporter

109.
https://github.com/SODALITE-EU/monitoring-system/blob/master/consul-registration-poc/openstack/node-exporter/xopera/playbooks/vm/create.yml#L127-L134
110.
https://github.com/SODALITE-EU/monitoring-system/blob/master/consul-registration-poc/openstack/node-exporter/xopera/playbooks/node_exporter.json.tmpl
111.
https://github.com/SODALITE-EU/monitoring-system/blob/master/consul-registration-poc/openstack/node-exporter/xopera/playbooks/vm/create.yml#L136-L137
112. https://github.com/SODALITE-EU/ipmi-exporter
113.
https://github.com/SODALITE-EU/iac-platform-stack/blob/master/docker-local/service.yaml#L643-L653