Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

Initial Implementation and Evaluation of the SODALITE Platform and Use Cases

D6.2

USTUTT

31.1.2020

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

Deliverable data			
Deliverable	Initial implementation and evaluation of the SODALITE platform and use cases		
Authors	Kamil Tokmakov (USTUTT), Dimitris Liparas (USTUTT), Ralf Schneider (USTUTT), Dennis Hoppe (USTUTT), Kalman Meth (IBM), Elisabetta Di Nitto (POLIMI), Paul Mundt (ADPT), Román Sosa González (ATOS), Airán González Gómez (ATOS), Mario Martínez Requena (ATOS), Yosu Gorroñogoitia (ATOS), Dragan Radolović (XLAB), Piero Fraternali (POLIMI), Rocio Nahime Torres (POLIMI), Georgios Meditskos, Stefanos Vrochidis (CERTH), Karthee Sivalingam (CRAY), Indika Kumara (UVT/JADS)		
Reviewers	Luciano Baresi (POLIMI), Karthee Sivalingam (CRAY)		
Dissemination level	Public		
History of changes	Dimitris Liparas, Kamil Tokmakov USTUTT	Outline created	19 November 2019, outline
	All	First Draft created for review	10.01.2020
	All	First round of reviews completed	13.01.2020
	All	Second Draft created after corrections	22.01.2020
	All	Second round of reviews completed	27.01.2020
	All	Final version	30.01.2020

Acknowledgement

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

Table of Contents

List of Figures

List Of Images

List of Tables

List Of Tables

Executive Summary

In this deliverable, we report on our key contributions towards the implementation and evaluation of the SODALITE platform and use cases. Whereas Deliverable D6.1 paved the way in M6 for the SODALITE platform by detailing the overall implementation plan and roadmap, we now report on the current status of implementation up to M12.

Key contributions and achievements with respect to the SODALITE platform are:

- Both SODALITE testbeds, Cloud and HPC, are set up, accessible and configured accordingly, to allow efficient execution of use cases. While setting up testbeds, benchmarks were performed to identify bottlenecks (e.g. slower network bandwidth than expected) and to resolve these or find workarounds.
- Following the strategy of early adoption, SODALITE components were made available publicly on GitHub (https://github.com/SODALITE-EU) with the aim to attract more developers outside of the consortium and increase uptake of our solutions.
- A continuous integration and deployment platform (CI/CD) based on Jenkins was set up to further streamline integration of individual SODALITE components, and to improve the overall code quality through automatic tests. The setup is already validated by a selected set of components.
- Initial deployment updates are presented for the three key SODALITE layers:
    - Semantic Modelling Layer: components of this layer such as the SODALITE IDE are in their initial version developed, deployed, and integrated. The SODALITE IDE is already released and integrated with the Semantic Reasoner Engine.
    - Infrastructure as Code Management Layer: components of this layer are developed and in the deployment stage. For example, the Verification Model Builder is released and enables users to build semantic models to verify TOSCA topologies.
    - Runtime Layer: components of this layer are developed and partially deployed; integration is ongoing. For example, monitoring based on Prometheus is released and capable of discovering new instances and gathering relevant metrics.

Key contributions and achievements with respect to the evaluation of the use cases are:

- POLIMI Snow Use Case: All components planned to be developed in year 1 of the project were developed according to the implementation plan presented in D6.1.
- USTUTT Virtual Clinical Trial Use Case: Key components such as the solver were deployed and executed successfully in the HPC testbed. The analysis revealed that the original processing pipeline needed to be extended by additional components.
- ADPT Vehicle IoT Use Case: To advance this use case, an Edge-based instance of the backend and relevant services was prepared, and use case components were integrated with the SODALITE stack in order to enable SODALITE to directly manage the deployment and configuration of deployed components.

Overall, the SODALITE development environment composed of the testbeds, repositories and the continuous integration and deployment platform are set up and made accessible to developers and use case owners. Further, initial releases and developments were achieved across all three key layers of the SODALITE platform according to the implementation plan up to M12. Finally, the three use cases of SODALITE are already executed on the HPC/Cloud testbeds and have been used to experiment with the features offered by the layers of the SODALITE platform. This deliverable is part of a series, where the next iteration will focus on the implementation activities done in the period M12 to M24. Specifically, D6.3 will report on component integration, discussing additional features of the SODALITE components and use cases, as well as a more in-depth evaluation of the improvements provided by the SODALITE platform for the use cases.

Glossary

Acronym	Explanation
3D	Three Dimensional
AADM	Abstract Application Deployment Model
ALPR	Automatic License-Plate Recognition
API	Application Program Interface
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command-Line Interface
CRI	Container Runtime Interface
CSAR	Cloud Service Archive
CT	Computer Tomography
CV	Computer Vision
DEM	Digital Elevation Model
DICOM	Digital Imaging and Communications in Medicine
DMI	Daily Median Image
DSL	Domain-Specific Language
DXA	Dual Energy X-ray Absorptiometry
EAR	Eye Aspect Ratio
ECG	Electrocardiogram
EMF	Eclipse Modelling Framework
EXIF	Exchangeable Image File Format
FEM	Finite Element Method
FOV	Field of View
GA	Grant Agreement
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
HPC	High Performance Computing
HPVM	High Performance Virtual Machine
IaC	Infrastructure as Code
IaaS	Infrastructure-as-a-Service
IDE	Integrated Development Environment
IoT	Internet of Things
IPMI	Intelligent Platform Management Interface
ITK	Insight Segmentation and Registration Toolkit

KB	Knowledge Base
LRE	Lightweight Runtime Environment
M2T	Model-to-Text
MCA	Marching Cubes Algorithm
MIGR	Mountain Image Geo-registration
ML	Machine Learning
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
MTU	Maximum Transmission Unit
NIC	Network Interface Controller
OCI	Open Container Initiative
OCR	Optical Character Recognition
PERCLOS	Percentage of Eyelid Closure
QoS	Quality of Service
RDF	Resource Description Framework
REST	Representational State Transfer
SVC	Support Vector Classifier
SVM	Support Vector Machine
ToR	Top-of-Rack
TOSCA	Topology and Orchestration Specification for Cloud Applications
UDJ	Universal Data Junction
UGI	User Generated Images
VIN	Vehicle Identification Number
VM	Virtual Machine
VTK	Visualization Toolkit
WP	Work Package

1 Introduction

The objectives of work package WP6 are integration, evaluation and validation of the SODALITE framework as detailed and specified in WP2; components to be integrated are developed across work packages WP3, WP4, and WP5. Evaluation and validation is done with the support of the three SODALITE use cases. This deliverable reports therefore on the current status of the SODALITE platform and its use cases to assess the overall progress made in year 1 of the project.

The SODALITE platform is one of the key building blocks of SODALITE by providing the backbone infrastructure based on HPC and Cloud testbeds. These testbeds are composed of heterogeneous hardware components such as CPUs and GPUs, to reflect the overall vision of SODALITE to address heterogeneity, by providing sophisticated software tools to allow for faster development, deployment and execution of applications on different target platforms.

Faster development and deployment of individual software components and applications is supported through the SODALITE development environment, which is composed of the Cloud and HPC testbeds, various code repositories, and a sophisticated continuous integration and deployment set up. Further, execution of applications is eased via the SODALITE software layers for semantic modelling of applications, management for infrastructure as a code, and the final runtime layer. Use cases are then prepared to be executed via the SODALITE layers on the development infrastructure.

1.1 Structure of the Document

The structure of this deliverable is as follows:

- The remaining part of Section 1 discusses the advancements made with respect to deliverable D6.1, highlights the SODALITE architecture and its components, as well as presents the objectives and status of the First Prototype.
- Section 2 provides a description of the development environment, which includes the HPC and Cloud testbeds, the repository and the CI/CD pipeline.
- Section 3 presents development status of the First Prototype, which includes the status of the components in each layer of the SODALITE platform - Modelling, Infrastructure-as-Code and Runtime layers.
- Section 4 provides the development status of the SODALITE demonstrating use cases.
- Section 5 highlights the implementation status of the First Prototype by describing how demonstrating use cases have been using the features offered by the SODALITE platform.
- Section 6 concludes this report.

1.2 Continuation of Deliverable D6.1

The previous deliverable, D6.1 [1], which was due in project month M6, reported about the overall implementation plan and roadmap to realize the SODALITE platform and the individual use cases; follow-up deliverables then provide incremental updates and basically replace the previous ones. The objective of D6.2 is to report on the status of the initial implementation of the SODALITE components and their integration into the First SODALITE Prototype (due in M12 and reported in D6.5), the initial implementation of the demonstrating use cases, as well as their status at project month M12. It also provides detailed information about the advancements made with respect to the SODALITE development environment, which includes the HPC and Cloud testbeds, the SODALITE repository and the Continuous Integration/Continuous Delivery (CI/CD) pipeline for automated components integration and testing.

In contrast to deliverable D6.1, this document does not include the thorough overview of potential software artefacts to be used within the overall SODALITE technology stack (cf. Section 2 in D6.1). Instead, Section 3 discusses the development and integration status of selected components for

each technical work package, which were originally presented in the previous deliverable. Furthermore, detailed specifications of the HPC and Cloud testbeds are only included in D6.1 (cf. Section 3 on page 23ff), since they are not likely to change in the future. Instead, this deliverable reports on the current setup and initial benchmarking results to assess the performance of the overall testbeds in Section 2; this section also introduces the SODALITE code repositories and the realization of a continuous integration and deployment platform (CI/CD), which are not discussed in deliverable D6.1. Finally, this deliverable introduces a new section on the development status of the First Prototype in Section 3, which is complemented by Section 5 on the implementation status of the First Prototype.

1.3 SODALITE Architecture

For greater clarity, we reproduce a synopsis of the SODALITE architecture that is described in Deliverable D2.1 [2]. For the details of the functional description, inputs, outputs and dependencies of each component, please see the architecture Section (Section 3) in D2.1.

SODALITE aims to provide developers and infrastructure operators with tools that abstract their application and infrastructure requirements to enable simpler and faster development, deployment, operation and execution of heterogeneous applications on heterogeneous, software-defined, high-performance and cloud infrastructures. To this end, SODALITE aims to produce:

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

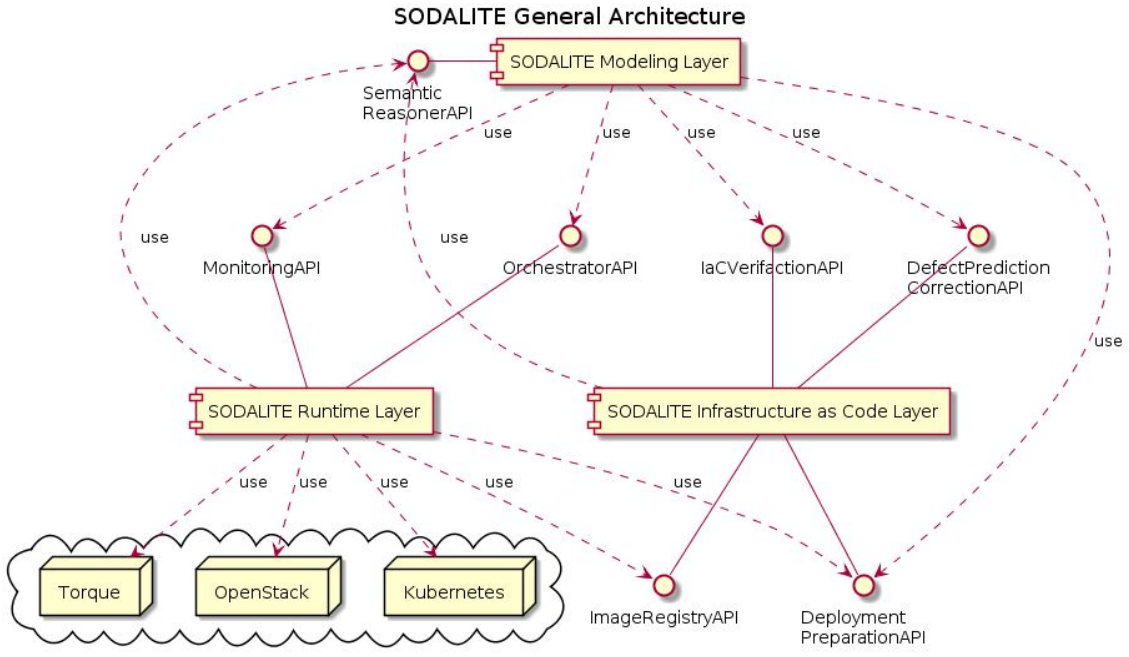

Figure 1 - SODALITE overall Architecture

The SODALITE platform is divided into three main layers, each covered by a separate work package. These layers are the Semantic Modelling layer (WP3), the Infrastructure as Code Management layer (WP4), and the Runtime layer (WP5). Figure 1 shows these layers together with their relationships.

1.3.1 SODALITE Semantic Modelling Layer

The components of the SODALITE Semantic Modelling Layer are presented in Figure 2.

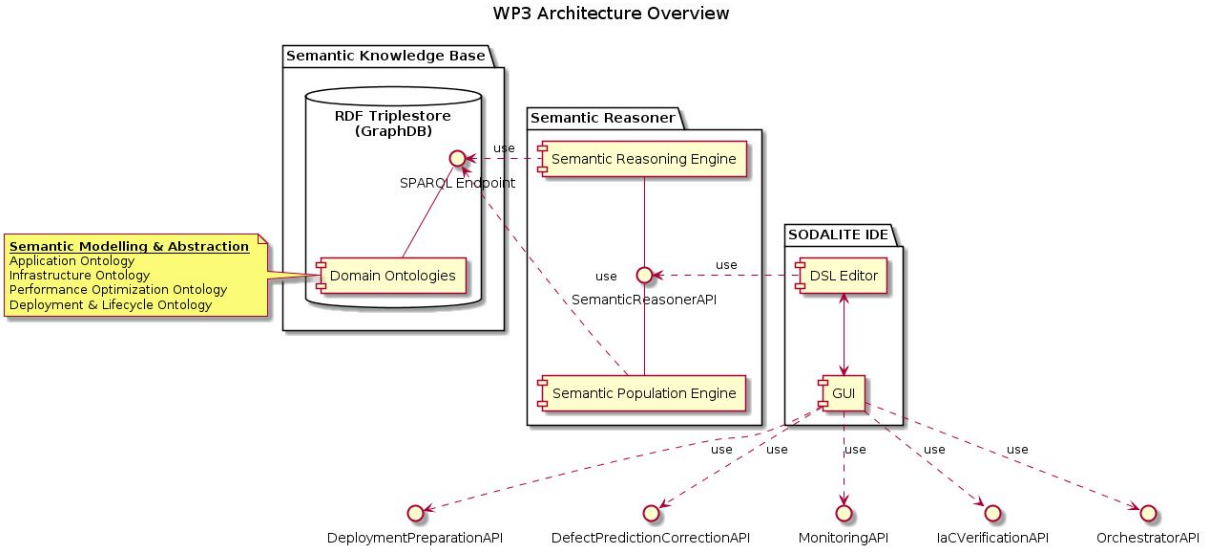

Figure 2 - SODALITE semantic modelling layer components (WP3).

The SODALITE IDE provides complete support for the authoring lifecycle of abstract application deployment models (see D2.1 for details) and assistance in composing application and resource models via GUI and DSL editor. The Semantic Knowledge Base (KB) is SODALITE's semantic repository that hosts the models (ontologies) created in WP3. The Semantic Reasoner is a middleware facilitating the interaction with the KB. In particular, it provides an API to support the population and retrieval of knowledge to/from the KB, and the application of rule-based semantic reasoning over the data stored in the KB. A broader description, implementation details and the development status of the components of the SODALITE Modelling Layer can be found in the technical deliverable D3.1 [3].

1.3.2 SODALITE Infrastructure as Code Management layer

The components of the SODALITE Infrastructure as Code (IaC) Management Layer are depicted in Figure 3.

The main task of the IaC Management layer is to take the modelling information provided by the SODALITE IDE (WP3) and produce an IaC blueprint. Deployment Preparation involves operations to build an IaC blueprint. These operations are handled by sub-components depicted in Figure 3 and are detailed in deliverable D2.1. A part of the architecture for the *Infrastructure as a Code* layer was redesigned. It now provides a single source of information from the SODALITE Knowledge Base instead of two independent repositories (namely IaC Repository and Knowledge Base), in order to eliminate the issues of synchronization between different sources. Additional components are envisioned to verify correctness of the provided model, to predict possible bugs in the provided model, and to optimise the application for a given target execution platform. A more detailed view on the components of the IaC Management layer is described in deliverable D4.1 [4].

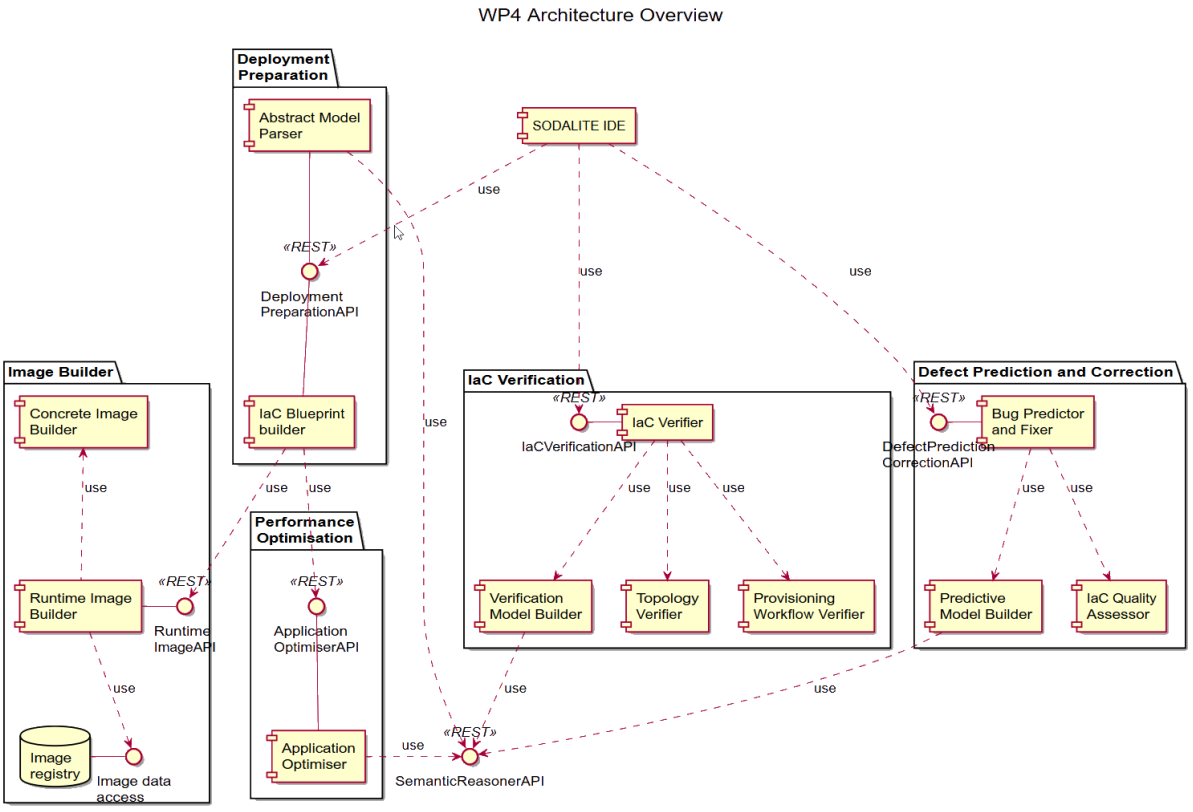

Figure 3 - SODALITE infrastructure as code management layer components (WP4)

1.3.3 SODALITE Runtime layer

The components of the SODALITE Runtime Layer are depicted in Figure 4.

The Runtime layer of SODALITE orchestrates the deployment of an application, monitors its execution and proposes changes to the application's runtime. It is composed of three main blocks: Orchestrator, Monitoring and Refactoring. The Orchestrator manages the lifecycle of an application deployed in heterogeneous infrastructures. The Monitoring component gathers metrics from the heterogeneous infrastructures. These metrics are used to determine if the application is running as expected. The Deployment Refactorer refactors the deployment model of an application in response to violations in the application goals. Deliverable D5.1 [5] describes the implementation details of the components of the Runtime layer.

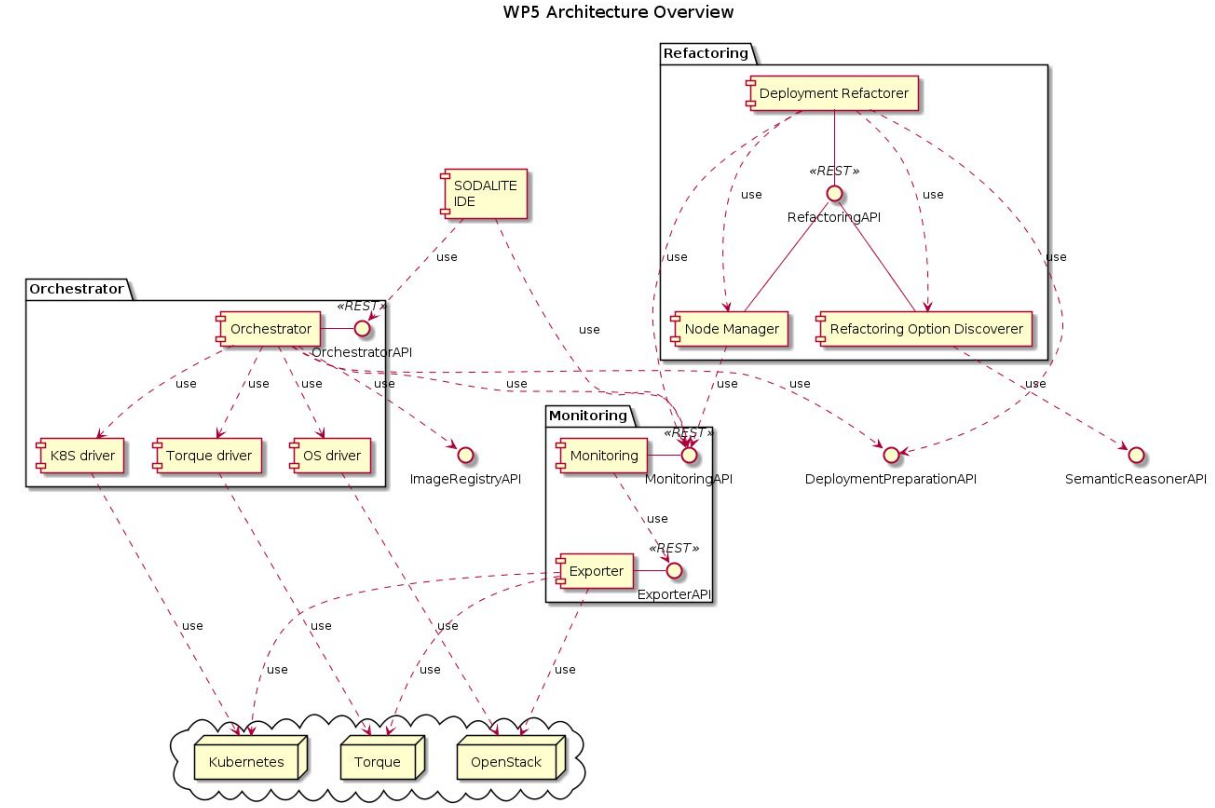

Figure 4 - SODALITE runtime layer components (WP5)

1.4 Objective of the First Prototype

In the First Prototype, most of the SODALITE components are expected to be released as initial and stable versions, and form the initial implementation of the SODALITE platform. The First Prototype is used to deploy and execute the initial implementation of the demonstrating use cases and aims to achieve goals that can be consolidated from the objectives of the SODALITE Architecture layers:

- Semantic Modelling Layer: the initial implementation of the semantic models, the repository and the IDE for supporting users in modelling the application and infrastructure.
- Infrastructure as Code (IaC) Management layer: the initial version of the deployment preparation for the selected infrastructure management systems and performance optimization; initial implementation of the analytics for the quality of the IaC - verification and bug prediction of deployment models.
- Runtime layer: the initial implementation of the cross-platform orchestrating tools, collection of monitoring metrics and initial version of predictive deployment refactoring.

1.5 Status of the First Prototype

The status of the SODALITE First Prototype at the end of project month M12 is presented in Table 1 and can be summarized as follows:

- The SODALITE development environment (composed of the HPC and Cloud testbeds, the SODALITE repository and the CI/CD pipeline) has been partially implemented and set up (more details are provided in Section 2 of this document). Several components of the First Prototype use the CI/CD pipeline to remotely build their artifacts. The components that do not require remote environments are built, tested and integrated manually on the local environments and deployments. Other components of the First Prototype do not utilize the CI/CD pipeline due to ongoing integration.

- In all three main layers of the SODALITE platform (Semantic Modelling, Infrastructure as Code Management, Runtime), the initial versions of the components have been released. Most of the components are partially deployed and integrated.
- The three demonstrating use cases of SODALITE have been defined and can be partially executed on the Prototype. More specifically, the developed components of the POLIMI Snow use case is executed on both Cloud and HPC testbeds using the SODALITE components of all the three layers of the Prototype; the developed components of the USTUTT Virtual Clinical Trial use case is executed on HPC testbed using only Infrastructure as Code Management and Runtime layers of the First Prototype; the developed components of the ADPT Vehicle IoT use case is executed on the Cloud testbed using only Infrastructure as Code Management and Runtime layers.

Table 1 - Overall status of the development environment, First Prototype and demonstrating use cases at M12

Components	**Status**
Development Environment	**HPC and Cloud testbeds** were set up.
	SODALITE repositories were structured and host the source code for SODALITE components.
	CI/CD server and pipeline were set up to remotely build software artifacts.
First Prototype Components	**Semantic Modelling Layer:** the initial versions were developed and released, locally deployed and integrated.
	Infrastructure as Code Management Layer: the initial versions were developed and released. The deployment and integration are partial for most of the components.
	Runtime Layer: the initial versions were developed and released. Most of the components are partially deployed and integrated.
Demonstrating Use Cases	**POLIMI Snow:** the components were released as scheduled.
	USTUTT Virtual Clinical Trial: the components were released as scheduled. The original processing pipeline needed to be extended by additional components.
	ADPT Vehicle IoT: the components were released as scheduled. The focus on refactoring for the Y1 demo further necessitated extension of the region routing component ahead of schedule.

It should be noted that this deliverable is the second iteration of four deliverables in total within Work Package 6 that report on the status of the SODALITE platform and the integration and evaluation of its use cases at regular intervals between project month M6 and M36. As detailed in

the first deliverable of WP6 [1], the following reporting period, specifically M12 to M24, will focus on component integration, delivery of more advanced features, as well as the initial evaluation of the improvement provided by the SODALITE platform for the demonstrating use cases. Finally, during the third year of the project, iterative measurements of the results produced by the SODALITE platform will be taken and based on these measurements, additional improvements will be applied to the SODALITE system.

2 Development Environment

For the development of SODALITE platform and its components, we introduce the project's development environment, which includes HPC and Cloud testbeds for provisioning virtual and bare-metal compute resources, SODALITE repository for storing the source code, infrastructure scripts and documentation of the SODALITE components and CI/CD pipeline for the automated testing and integration of the components.

In the following sections, the status of the development environment is described. Section 2.1 presents the current state of the HPC and Cloud testbeds. The structure of the SODALITE repository and CI/CD are presented in Section 2.2 and Section 2.3, respectively.

2.1 Cloud and HPC Testbeds

Figure 5 presents an overview on HPC and Cloud testbeds, their components, resources and the supported platforms for the experimentation with cross-system orchestration and monitoring.

The intention of the HPC testbed is to provide to developers and use case providers bare-metal compute resources (e.g. CPUs, GPUs) managed by the workload managers, such as Torque[1] (or its extension - vTorque[2]), whereas the Cloud testbed provisions virtualized resources (e.g. virtual machines, containers) managed by OpenStack[3] and Kubernetes[4]. Furthermore, the Cloud testbed hosts the development environment (DevCloud), which contains CI/CD server and deployed SODALITE components. The support for edge computing backends and serverless platforms (e.g. OpenWhisk[5]) is foreseen in the next iteration of the project.

HPC testbed. The HPC testbed is hosted in USTUTT and its functional description is presented in Figure 6. Three nodes were deployed: a front-end node and two compute nodes. The specification of the front-end node is 20 cores of Intel Xeon E5-2630v4 CPU, 192GB of DDR4 RAM and 37TB of RAID-60 HDD storage; for the compute nodes, it is 20 cores of Intel Xeon E5-2630v4 CPU, 128GB of DDR4 RAM and 1.8TB of SW RAID-1 SSD storage per each node. A full version of the specifications can be found in deliverable D6.1, Section 3.1.2.

[1] https://adaptivecomputing.com/cherry-services/torque-resource-manager/
[2] https://www.mikelangelo-project.eu/technology/vtorque-virtualization-support-for-torque/
[3] https://www.openstack.org/
[4] https://kubernetes.io/
[5] https://openwhisk.apache.org/

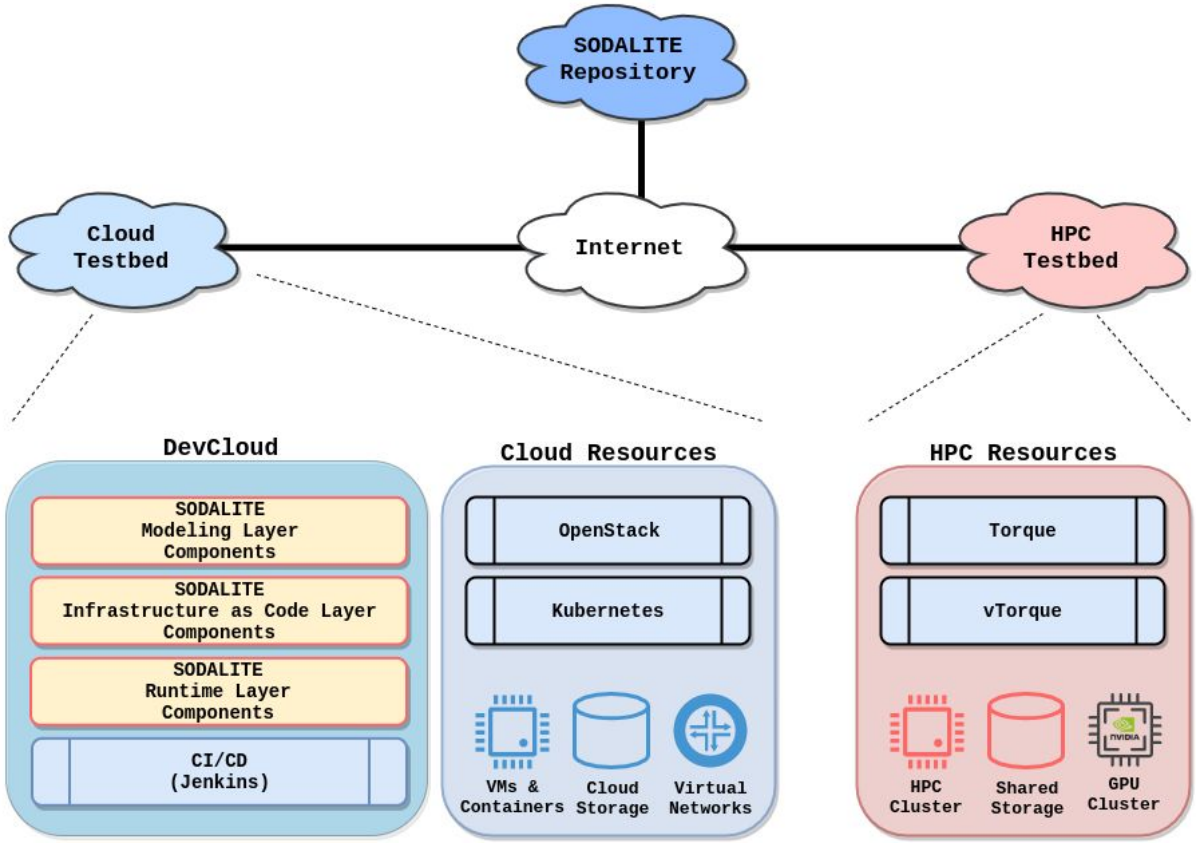

Figure 5 - HPC and Cloud testbeds overview

The front-end node hosts the Ironic[6] service - an OpenStack service to provision bare-metal nodes - in order to provide flexible and on-the-fly reconfiguration of the compute nodes, e.g. changing base operating system of the nodes. Moreover, it runs a Torque server (pbs_server) to manage the compute resources of the testbed and serves as a Torque client in order to submit PBS jobs to be run on the compute nodes. Additionally, the front-end node runs an LDAP server for centralized user authentication, shares an NFS storage with other nodes and hosts a local Singularity[7] registry, which allows building and pulling container images. The front-end node can be accessed at sodalite-fe.hlrs.de remotely via SSH, providing an endpoint for the SODALITE Runtime Layer.

The two compute nodes were deployed via Ironic and host Torque compute clients (Machine Oriented Mini-server, pbs_mom) in order to provide management endpoints to be used by the Torque server. The compute nodes support MPI for process parallelization and Singularity runtime as a low-overhead, lightweight and secure runtime specifically developed for HPC workloads (see the deliverable D5.1, Section 5 about various lightweight runtime environments). Additionally, the compute nodes contain GPUs attached via PCIe slot, providing hardware acceleration in graphics and image processing.

[6] https://wiki.openstack.org/wiki/Ironic
[7] https://singularity.lbl.gov/

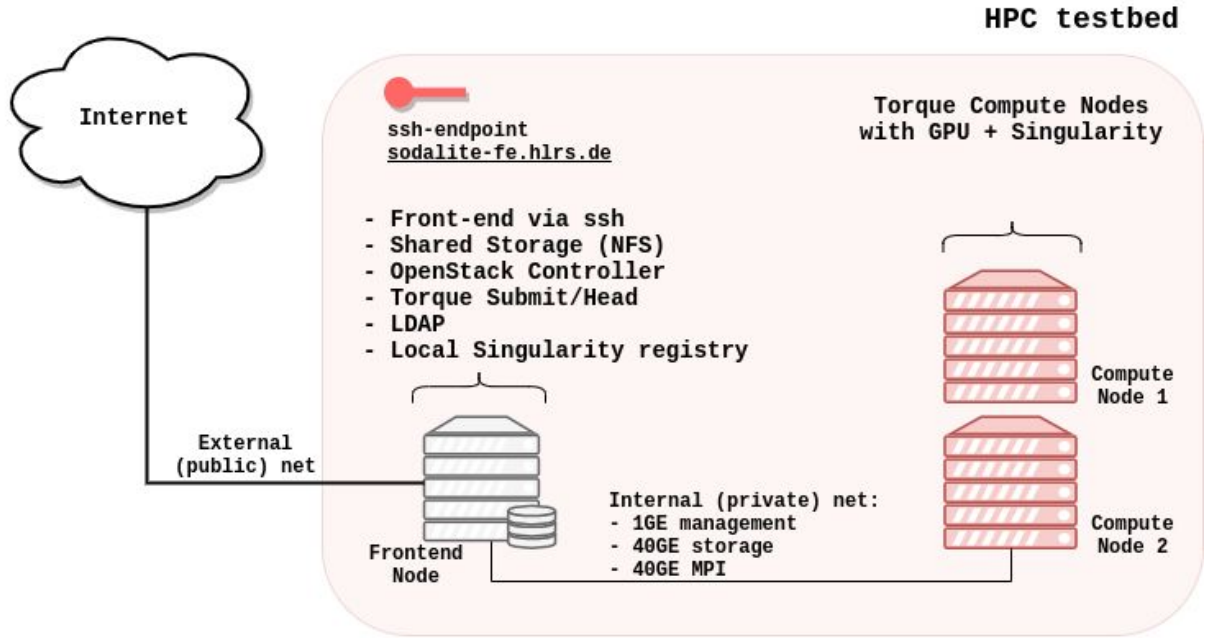

Figure 6 - A functional description of HPC testbed

The reason for such a small scale (3 nodes out of available 9 nodes) of the testbed is to experiment and obtain requirements, perform tuning, optimization for the SODALITE infrastructure before a large scale testbed is deployed for performance oriented tasks. As such, benchmarks to model the performance of the infrastructure described in the deliverable D3.3 [6] were preliminary evaluated on the current state of the testbed, helping to diagnose the issues with the testbed infrastructure. In particular, the results of the Effective Bandwidth[8] (b_eff) benchmark, shown in Table 2, have identified a communication bottleneck (poor Randomly Ordered-Ring Bandwidth). Therefore, a dedicated faster interconnect (40GE) between the nodes of the HPC testbed was installed and the bandwidth has increased by a factor of 16.

Table 2 - Bandwidth improvements by changing the interconnect in the HPC testbed

Interconnect	1GE Ethernet	40GE Ethernet
Randomly Ordered-Ring Bandwidth, GBps	0.115635	1.92319

Nevertheless, while the compute resources were enough to run the initial implementation of the SODALITE use cases, it wasn't sufficient to run the benchmarks at the fullest extent to derive the performance model, hence for the next stages of the project, the number of nodes will be increased.

HPC testbed has been used to run the entire Virtual Clinical Trials use case running Density Mapping, Probabilistic Mapping and Solver components, and to train Skyline extraction CNN model of the Snow Use Case using GPUs. The baseline measurements of the performance of the use cases can be found in Section 4 and in deliverable D3.3 (Section 6).

Cloud testbed. The Cloud testbed hosted in ATOS consists of three nodes: two compute nodes and one storage node with similar specs - 16 cores of Intel Xeon E5-2670 with 64GB DRR3 RAM, however,

[8] https://fs.hlrs.de/projects/par/mpi//b_eff/

for compute nodes the internal storage size is 12TB each, whereas for storage node, it is 18TB. A more detailed specification can be found in the deliverable D6.1 (Section 3.1.1).

Functional components of the Cloud testbed are presented in Figure 7. It is backed with OpenStack that provides virtual IaaS resources via OpenStack services: virtual machines (VMs) via Nova Compute, block storage via Cinder and networking via Neutron. Furthermore, these resources are used to deploy a Kubernetes cluster consisting of two VMs - one controller and one worker node. The testbed internally provides interfaces, such as IPMI, and tools, such as Skydive, for basic monitoring of the infrastructure and dynamic runtime environments. Therefore, the Cloud testbed provides OpenStack and Kubernetes endpoints for the orchestration, as well as the endpoints for monitoring through the SODALITE Runtime Layer. Please refer to the deliverable D5.1 about orchestrating and monitoring capabilities of the SODALITE platform.

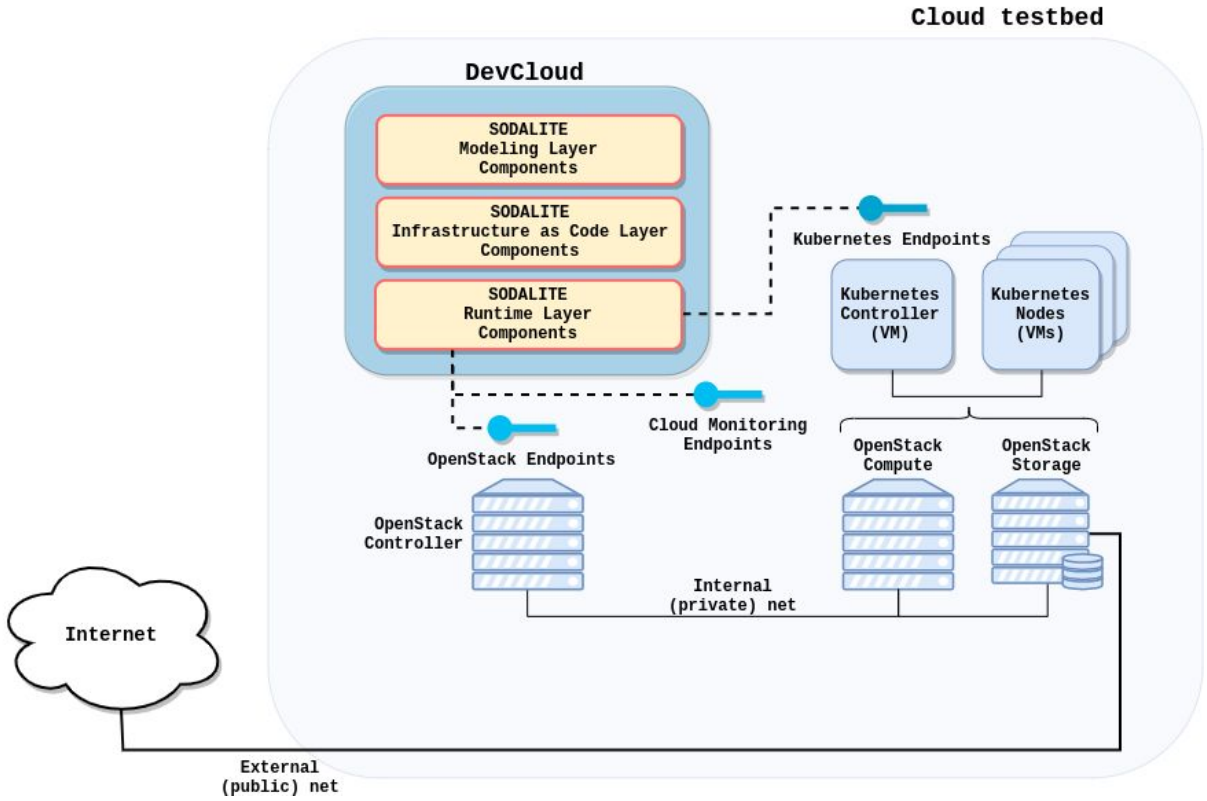

Figure 7 - A functional description of Cloud testbed.

The development environment (DevCloud) for the deployment and integration of SODALITE components resides in the Cloud testbed. For the first iteration of the project, the components were deployed on virtual machines sharing the resources with the demonstrating use cases. For the next iterations, an isolated and containerized environment will be provided to avoid the resource contention and provide versioning (e.g. development, production) and staging (e.g. test, integration) of the developed components. Table 3 outlines the SODALITE components that were deployed on the Cloud testbed.

Table 3 - SODALITE components in Cloud testbed

Name	WP	Description	Resources (vCPUs | RAM | Storage)
Image Registry	4	A private Docker[9] image registry to store	2 | 4GB | 40GB

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

		the images (artifacts) of the components of the demonstrating use cases. See the deliverable D4.1 (Section 3.2.3) for more details.	
xOpera REST API	5	A REST API for xOpera[10] orchestrator that manages the lifecycle of the applications	2 | 4GB | 40GB
xOpera Persistence Engine	5	A PostgreSQL[11] database that stores the internal deployment states of xOpera orchestrator	2 | 4GB | 40GB
Monitoring Server	5	A monitoring server that contains Prometheus[12] server as a metrics collector, Skydive analyzer and Prometheus exporters	1 | 2GB | 20GB
Jenkins server	6	A Jenkins[13] server that hosts and runs CI/CD pipelines	4 | 16GB | 200 GB
Kubernetes Master Node	6	A Kubernetes control node that exposes the Kubernetes API	2 | 8GB | 80GB
Kubernetes Worker Node	6	A Kubernetes worker node that runs containerized workloads	8 | 16GB | 50GB

The Cloud testbed provisions resources on-demand for most of the components of Snow and Vehicle IoT use cases. The baseline performance of the use cases deployed in the Cloud testbed can be found in Section 4 and in the deliverable D3.3 (Section 6).

2.2 SODALITE Repositories

The adopted organization mimics the structure behind the conceived architecture and provides a dedicated repository for each component in the project. The consortium created the SODALITE-EU organization (https://github.com/SODALITE-EU) on GitHub and identified 19 repositories for the components planned and released so far. This decision was made to better manage the different contributions and ease the management of different and heterogeneous technologies. All these components are currently released under the Apache 2 license scheme. Different license models might be used and integrated in the project in the next phases. The artifacts produced by SODALITE are stored in a set of dedicated GitHub repositories. The interested reader is referred to deliverable D2.4 [7] for additional information and detailed guidelines on how to contribute to the project.

2.3 CI/CD Pipeline

Jenkins was chosen as the CI/CD integration tool for SODALITE. A Jenkins server was installed in the cloud testbed to handle the CI/CD process. At this stage of the project, several of the SODALITE

[10] https://github.com/xlab-si/xopera-opera
[11] https://www.postgresql.org/
[12] https://prometheus.io/
[13] https://jenkins.io/

components (e.g. Semantic Reasoner[14] and Defect Prediction[15]) have been incorporated into the Jenkins CI/CD pipeline to verify its proper functionality. As the project progresses and additional automated tests are built, we expect to add other repositories to the pipeline and automate more of the process using Jenkins. It is envisioned that as the changes of the source code of the components are submitted to the SODALITE repository by the developers via GitHub Pull Requests, it then triggers Jenkins to run the CI/CD pipeline, where automated unit, integration and functional tests of SODALITE components are scheduled. If all tests pass, the Pull Request is marked as successful on GitHub and can then be merged into the master branch of the repository. The SODALITE components are then ready for deployment (as a new production version of the SODALITE platform) and are subsequently available to users.

3 Development Status of the First Prototype

This section describes the development status of the First Prototype and its constituent components and modules. Table 4 highlights the overall view on the initial development, deployment and integration of the SODALITE components done during Y1 of the project. The initial versions of all the components are developed and they are ready to be used. For most of the components, there is ongoing work on the deployment in the testbeds and integration to the overall platform. The deployment and integration work for the components that are expected to be released in M18 (Application Optimiser, IaC Model Repository and Node Manager) has not yet started.

Table 4 - Development status of the First Prototype

Semantic Modelling Layer			
Component	**Development**	**Deployment**	**Integration**
SODALITE IDE			
Semantic Reasoner			
Semantic KB			
IaC Management Layer			
Component	**Development**	**Deployment**	**Integration**
Abstract Model Parser			
IaC Blueprint Builder			
Runtime Image Builder			
Concrete Image Builder			
Application Optimiser			
IaC Verifier			
Verification Model Builder			
Topology Verifier			
Provisioning Workflow Verifier			
Bug Predictor and Fixer			
Predictive Model Builder			
IaC Quality Assessor			
IaC Model Repository			
Image Registry			
Runtime Layer			
Component	**Development**	**Deployment**	**Integration**

[14] https://github.com/SODALITE-EU/semantic-reasoner
[15] https://github.com/SODALITE-EU/defect-prediction

Orchestrator + Drivers			
xOpera REST API			
Monitoring			
Deployment Refactorer			
Node Manager			
Refactoring Option Discoverer			
Table legend	**Completed**	**Partial**	**Not started**

The following subsections provide a detailed description of the development status of each layer of the First Prototype, additionally presenting the source code location in the SODALITE repository, issues and steps towards the next prototype.

3.1 SODALITE Semantic Modelling Layer

The Semantic Modelling layer provides the framework for semantically representing abstractions of a) applications, capturing higher-level information that will enable the conceptual description of artifacts, code, functional and non-functional requirements, software dependencies etc.; b) infrastructures, available services and service capabilities in terms of functionalities, resources and business characteristics offered, and QoS; and c) application and infrastructure performance optimisation in terms of application models reflecting their classification to a set of known performance qualities and infrastructure models representing performance features of the hardware. These semantic abstractions are realised in the form of Resource Description Framework (RDF) Knowledge Graphs[16], aiming at the formal representation and linking of application and infrastructure requirements that enables semantic reasoning framework to be developed on top of the RDF graphs to support search, discovery, validation and reuse. The Semantic Modelling Layer also provides the textual editor (IDE) of SODALITE that allows end users to define Abstract Application Deployment Models (AADMs) by reusing components and resources from the KB.

The following subsections provide details on the status of the components of the Semantic Modelling Layer.

3.1.1 SODALITE IDE

The SODALITE IDE that provides the GUI and the DSL Editor to assist end users in composing resource and application models.

Table 5 - Development status of SODALITE IDE

Module name	**SODALITE IDE**
Github location	https://github.com/SODALITE-EU/ide
Development status	Released version - v0.6
Deployment status	IDE is a standalone client, intended to be installed for each Sodalite AOEs/REs on their computer. IDE is not deployed in any (virtualized) infrastructure

[16] http://www.w3.org/RDF/

Integration status	IDE is integrated with the Semantic Reasoner Engine through exposed REST API
Integration issues/dependencies	IDE integrates with the Semantic Reasoner API (v0.6). IDE will require integration with the IaC layer
Next steps	IDE extension for AADM and RM authoring. Seamless integration with new releases of the Reasoner API. Integration of IoC Layer.

3.1.2 Semantic Reasoner

The Semantic Reasoner, which is a logical middleware that facilitates the interaction with the KB through the REST API (Semantic Reasoning Engine module - SRE), as well as the population of the KB with information coming from the SODALITE IDE users (Semantic Population Engine module - SPE).

Table 6 - Development status of Semantic Reasoning Engine

Module name	**Semantic Reasoning Engine**
Github location	https://github.com/SODALITE-EU/semantic-reasoner
Development status	Released version - v1.0 (provides the basic reasoning infrastructure to WP4 for developing searching and validation services; it provides the REST API that can be used to save and get data from the semantic triple store)
Deployment status	Deployed locally (the REST API is fully functional and deployed locally in a web container)
Integration status	Integrated locally with SODALITE IDE and Semantic Knowledge Base (all components are able to communicate with the REST API through HTTP calls)
Integration issues/dependencies	The Semantic Reasoning Engine depends on the Semantic Knowledge Base
Next steps	Enriching REST API

Table 7 - Development status of Semantic Population Engine

Module name	**Semantic Population Engine**
Github location	https://github.com/SODALITE-EU/semantic-reasoner
Development status	Released version - v1.0 (the component is able to map the SODALITE DSL to the conceptual model (ontologies) of SODALITE)

Deployment status	Deployed locally (the component can be invoked through the reasoner REST API)
Integration status	Integrated locally with SODALITE IDE and Semantic Knowledge Base (IDE is able to push DSL definitions using the REST API through HTTP calls)
Integration issues/dependencies	The Semantic Reasoning Engine depends on the Semantic Knowledge Base
Next steps	Support more sophisticated KB population capabilities

3.1.3 Semantic Knowledge Base

The Semantic Knowledge Base (KB), which is SODALITE's semantic repository (RDF triple store) that hosts the models (domain ontologies), created in WP3.

Table 8 - Development status of RDF Triple Store

Module name	**RDF Triple Store**
Github location	N/A (third-party semantic repository)
Development status	N/A
Deployment status	Deployed locally (a third-party RDF triple store has been deployed to a local machine, providing a native SPARQL endpoint used by the Semantic Reasoner)
Integration status	Integrated locally with the Semantic Reasoner (the RDF triple store is accessible through HTTP calls)
Integration issues/dependencies	N/A
Next steps	N/A

Table 9 - Development status of Domain Ontologies

Module name	**Domain Ontologies**
Github location	https://github.com/SODALITE-EU/semantic-models
Development status	Released version - v1.0
Deployment status	Deployed locally
Integration status	Ontologies have been imported into the RDF triple store, implementing the conceptual model of SODALITE

Integration issues/dependencies	N/A. There are no dependencies. Ontologies need to be inline with the conceptual model of SODALITE and the Semantic Population Engine (that is responsible for populating the triple store with instances)
Next steps	Updates on the vocabulary, according to the modelling requirements

3.2 SODALITE Infrastructure as Code Management layer

The main task of the IaC Management layer is to take the modelling information provided by the SODALITE IDE (WP3) and produce an IaC blueprint that will be consumed by the Runtime layer.

In the following subsections, the status of the modules of the IaC Management layer is presented.

3.2.1 Abstract Model Parser

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

Table 10 - Development status of Abstract Model Parser

Module name	**Abstract Model Parser**
Github location	https://github.com/SODALITE-EU/iac-blueprint-builder
Development status	Released version - v0.1
Deployment status	Deployed locally
Integration status	Partially integrated
Integration issues/dependencies	The Abstract Model Parser is dependent on the SODALITE IDE.
Next steps	Define a parsing grammar to parse the abstract model provided by SODALITE IDE and saved in the KB

3.2.2 IaC Blueprint Builder

This component internally produces the IaC blueprint based on the abstract application deployment model and the data parsed by the Abstract Model Parser.

Table 11 - Development status of IaC Blueprint Builder

Module name	**IaC Blueprint Builder**
Github location	https://github.com/SODALITE-EU/iac-blueprint-builder
Development status	Released version - v0.1
Deployment status	Deployed locally
Integration status	Partially integrated
Integration issues/dependencies	The IaC Blueprint Builder is dependent on SODALITE IDE and the Abstract Model Parser.
Next steps	Provide a consolidated prototype. Study the extensibility of the approach to other languages rather than TOSCA (e.g., Ansible).

3.2.3 Runtime Image Builder

Runtime image builder builds the runtime images based on Target architecture and artifact definition. A runtime image is equipped with configuration, artifact executable binary, configuration metadata and a monitoring artifact.

Table 12 - Development status of Runtime Image Builder

Module name	**Runtime Image Builder**
Github location	https://github.com/SODALITE-EU/image-builder
Development status	Released version - v0.1. Initial TOSCA/Ansible blueprints for building of the images, REST API endpoints
Deployment status	Deployed on the testbed
Integration status	Partially integrated
Integration issues/dependencies	The Runtime Image Builder is dependent on Image Registry, Semantic Knowledge Base (API), IDE, and Concrete Image Builder.
Next steps	Provide an extension for docker and singularity image building with integration to application optimizer

3.2.4 Concrete Image Builder

It is used for the implementation of a concrete image builder for the execution platform to handle the specific implementation regarding configuration, deployment or monitoring. Since there could be significant differences between the images for various execution platforms (HPC/Cloud/Kubernetes), Concrete Image Builder builds platform specific images.

Table 13 - Development status of Concrete Image Builder

Module name	**Concrete Image Builder**
Github location	https://github.com/SODALITE-EU/image-builder
Development status	Released version - v0.1.
Deployment status	Deployed on the testbed
Integration status	Partially integrated
Integration issues/dependencies	The Concrete Image Builder is dependent on Runtime Image Builder and the Image Registry
Next steps	Identify an adapter pattern to satisfy and bridge the different approaches for targeting HPC/Cloud/Kubernetes execution platforms; Analyze the post deployment configuration done by the Orchestrator. Setup a separate connection to singularity image registry

3.2.5 Application Optimiser

Static Application Optimiser optimises application for a given target platform based on the optimisation options selected.

Table 14 - Development status of Application Optimiser

Module name	**Application Optimiser**
Github location	https://github.com/SODALITE-EU/application-optimisation
Development status	Released version - v0.1. Baseline version of Skyline Extraction training built and profiled. Density Mapping and Solver (Code Aster) components of Clinical trials built and profiled.
Deployment status	NOT STARTED

Integration status	NOT STARTED
Integration issues/dependencies	The Application optimiser is dependent on the IaC Model repository and the Image registry
Next steps	Identify Application performance features or parameters that influence performance; Map application features to infrastructure and actual application optimisation; Build optimised containers for applications: AI training, Big data Analytics and Traditional HPC (Solver) to enable optimisation

3.2.6 IaC Verifier

This component acts as a facade to the Topology Verifier and Provisioning Workflow Verifier, and coordinates the processes of verification of the application deployment topology and provisioning workflow.

Table 15 - Development status of IaC Verifier

Module name	**IaC Verifier**
Github location	https://github.com/SODALITE-EU/verification
Development status	Released version - v0.1 Basic REST API
Deployment status	Deployed locally and on Cloud Testbed
Integration status	Partially integrated Integrated with Topology Verifier
Integration issues/dependencies	This module uses other modules Verification Model Builder, Topology Verifier, and Provisioning Workflow Verifier.
Next steps	Integrate provisioning workflow verification; Update REST API as new verification capabilities are added.

3.2.7 Verification Model Builder

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

Table 16 - Development status of Verification Model Builder

Module name	**Verification Model Builder**
Github location	https://github.com/SODALITE-EU/verification
Development status	Released version - v0.1 Build semantic models for TOSCA topology verification Initial mappings of Ansible workflows to Petri Net were done. Ongoing work to build the model transformer for translating Ansible workflows into Petri Net.
Deployment status	Deployed locally and on Cloud Testbed
Integration status	Partially integrated Integrated with Semantic Reasoning Engine
Integration issues/dependencies	This module uses Semantic Knowledge Base, and Semantic Reasoner.
Next steps	Support building a formal model for verifying the provisioning workflow described in IaC artifacts; Update the semantic model used by the Topology Verifier

3.2.8 Topology Verifier

This component verifies the constraints over the structures of the TOSCA blueprints and Ansible scripts. This will implement the verification of the requirements of the nodes, the node-relationships, the capabilities of the nodes, and node substitutability.

Table 17 - Development status of Topology Verifier

Module name	**Topology Verifier**
Github location	https://github.com/SODALITE-EU/verification
Development status	Released version - v0.1 Initial support for TOSCA topology verification
Deployment status	Deployed locally and Cloud Testbed
Integration status	Partially integrated Integrated with Verification Model Builder
Integration issues/dependencies	This module depends on the formal model built by the Verification Model Builder;
Next steps	Complete the verification of the deployment topology described in the TOSCA blueprint;

	Add the support for checking substitutability of the nodes in the deployment topology (TOSCA substitution mappings)

3.2.9 Provisioning Workflow Verifier

This component verifies the constraints over the deployment (provisioning) workflow of the application using one of the widely used techniques for verifying workflows such as Petri Nets [8]. The workflow is described in the Ansible scripts in terms of tasks, roles, plays, and variables.

Table 18 - Development status of Provisioning Workflow Verifier

Module name	**Provisioning Workflow Verifier**
Github location	https://github.com/SODALITE-EU/verification
Development status	Ongoing work to create and verify Petri-net models programmatically and via the ProM tool[17]
Deployment status	Ongoing work to deploy and test the Petri-net based verifier locally.
Integration status	Ongoing work to integrate with the Predictive Model Builder
Integration issues/dependencies	This module depends on the formal model built by Verification Model Builder.
Next steps	Complete the verification of the soundness and well-structuredness of a provisioning workflow

3.2.10 Bug Predictor and Fixer

Bug Predictor and Fixer detects the smells in TOSCA and Ansible artifacts and suggests corrections or fixes for each smell.

Table 19 - Development status of Bug Predictor and Fixer.

Module name	**Bug Predictor and Fixer**
Github location	https://github.com/SODALITE-EU/defect-prediction
Development status	Released version - v0.1 Detect code, design, and security smells for Ansible Detect security smells for TOSCA Web-based User Interface for checking bugs in TOSCA and Ansible files.
Deployment status	Deployed locally and on Cloud Testbed

[17] http://www.promtools.org/doku.php

Integration status	Partially integrated Integrated with Predictive Model Builder
Integration issues/dependencies	This module depends on the modules Predictive Model Builder, IaC Quality Assessor, and Semantic Knowledge Base.
Next steps	Complete data-driven approaches to predicting named-based and module usage related bugs in Ansible; Add correction support for a subset of bugs

3.2.11 Predictive Model Builder

This component builds the models that can find the smells in TOSCA and Ansible artifacts.

Table 20 - Development status of Predictive Model Builder

Module name	**Predictive Model Builder**
Github location	https://github.com/SODALITE-EU/defect-prediction
Development status	Released version - v0.1 Build semantic models for detecting security smells for TOSCA; Build rule-based and heuristics based model for detecting smells in Ansible
Deployment status	Deployed locally and on Cloud Testbed
Integration status	Partially integrated Integrated with Semantic Reasoning Engine
Integration issues/dependencies	This module uses Semantic Knowledge Base, and Semantic Reasoner.
Next steps	Complete a number of ongoing tasks on machine learning based models for defect prediction; Improve semantic prediction models for new types of bugs

3.2.12 IaC Quality Assessor

This component can calculate different software quality metrics for TOSCA and Ansible artifacts.

Table 21 - Development status of IaC Quality Assessor

Module name	**IaC Quality Assessor**
Github location	https://github.com/SODALITE-EU/iac-quality-framework

Development status	Released version - v0.1 Can calculate the IaC metrics required by the design smell detection for Ansible
Deployment status	Deployed locally and on Cloud Testbed
Integration status	Partially integrated Used by Ansible smell detection
Integration issues/dependencies	The metrics calculated by this module are used by the heuristic based models for predicting different types of smells.
Next steps	Support new types of IaC metrics for TOSCA and Ansible (as required by the bug prediction tools)

3.2.13 IaC Model Repository

IaC Model repository is a part of the Knowledge Base and will contain:

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

Table 22 - Development status of IaC Model Repository

Module name	**IaC Model Repository**
Github location	https://github.com/SODALITE-EU/iac-management https://github.com/SODALITE-EU/application-optimisation
Development status	Released version - v0.1 Prototype of application and infrastructure model developed
Deployment status	NOT STARTED
Integration status	NOT STARTED
Integration issues/dependencies	IaC Model repository interacts with the SODALITE IDE and contains the Performance Model of infrastructure and application (offline analysis).
Next steps	Develop the schema for the Model Repository and define API for accessing the model repository; Develop the full Application and performance Model.

3.2.14 Image Registry

Image Registry will store the executable runtime image of the artifact defined in the application design process and built in the SODALITE deployment preparation process.

Table 23 - Development status of Image Registry

Module name	**Image Registry**
Github location	https://github.com/SODALITE-EU/iac-management
Development status	Released version - v0.1 Prototype of a private internal and secured docker registry deployed through IaC
Deployment status	Partially deployed: Only Docker image registry is deployed in the Cloud testbed
Integration status	Partially integrated: Only Docker images can be retrieved by the Orchestrator
Integration issues/dependencies	TLS access key distribution to other components (eg. orchestrator and image builder); REST API; xOpera as the IaC blueprint execution engine
Next steps	Singularity image registry to be deployed and integrated.

3.3 SODALITE Runtime layer

The Runtime layer of SODALITE is in charge of the deployment of SODALITE applications into heterogeneous infrastructures, its monitoring and the refactoring of the deployment in response to violations in the application goals.

The subsections below describe the status of the initial implementation of the components of Runtime layer.

3.3.1 Orchestrator + Drivers

The Orchestrator of the Runtime Layer is in charge of managing the lifecycle of applications deployed in heterogeneous infrastructures. The drivers/plugins facilitate the management of a specific infrastructure, e.g. ALDE[18] provides an abstraction for the orchestrator for platform-independent application lifecycle management in HPC environment.

Table 24 - Development status of Orchestrator

Module name	**Orchestrator**
Github location	https://github.com/SODALITE-EU/orchestrator Orchestrator xOpera referenced as a git submodule

[18] https://github.com/TANGO-Project/alde

Development status	Released version - v0.1 Initial version of xOpera and REST API bundled and dockerized
Deployment status	Deployed in the Cloud testbed
Integration status	Partially integrated
Integration issues/dependencies	xOpera orchestrator / xOpera REST API / Image Registry / Postgres for blueprint persistence
Next steps	Extend the support of other execution platforms (e.g. Slurm, Docker Swarm). Support TOSCA Workflows.

Table 25 - Development status of ALDE

Module name	**ALDE**
Github location	https://github.com/SODALITE-EU/orchestrator
Development status	Released version - v1.0 Outcome of TANGO[19]
Deployment status	NOT STARTED
Integration status	NOT STARTED
Integration issues/dependencies	Adapt to the SODALITE Infrastructure
Next steps	Torque support, integrate with HPC testbed. Support for HPC workflows, provisioning of security layer.

3.3.2 Monitoring

The Monitoring is responsible for collecting metrics at the level of application and infrastructure. This is done with a conjunction of two elements: exporters situated in each service or machine that wants to be monitored that expose the required metrics and a central monitoring service (Prometheus) that collects and gathers the metrics provided by the exporters. The following repositories include Prometheus configuration files from the working monitoring system and a custom IPMI exporter.

[19] http://www.tango-project.eu/

Table 26 - Development status of Monitoring system

Module name	**Monitoring system**
Github location	https://github.com/SODALITE-EU/monitoring-system
Development status	Released version - v1.0 System functioning, capable of discovering new instances and providing metrics.
Deployment status	Deployed in the Cloud testbed
Integration status	Partially integrated
Dependencies	Prometheus and various exporters
Next steps	Include new metrics. Develop a monitoring platform/GUI.

Table 27 - Development status on IPMI exporter

Module name	**IPMI exporter**
Github location	https://github.com/SODALITE-EU/ipmi-exporter
Development status	Released version - v1.0 Functioning. Capable of exposing power measurements from physical nodes.
Deployment status	Deployed in the Cloud testbed
Integration status	Partially integrated
Dependencies	Prometheus
Next steps	Include additional metrics.

Table 28 - Development status on HPC exporter

Module name	**HPC exporter**
Github location	https://github.com/SODALITE-EU/hpc-exporter
Development status	Development not yet started

Deployment status	NOT STARTED
Integration status	NOT STARTED
Dependencies	Prometheus
Next steps	Start development of Prometheus exporter that monitor different metrics from HPC infrastructure and applications running on it.

Table 29 - Development status on HPC exporter

Module name	**LRE exporter**
Github location	https://github.com/SODALITE-EU/monitoring-lre-agent
Development status	Development not yet started
Deployment status	NOT STARTED
Integration status	NOT STARTED
Dependencies	Prometheus
Next steps	Start development of Prometheus exporter that monitor different metrics at the level of LRE.

3.3.3 Deployment Refactorer

The deployment refactorer can modify the deployment model of an application at runtime to prevent the violations of the application performance goals.

Table 30 - Development status of Deployment Refactorer

Module name	**Deployment Refactorer**
Github location	https://github.com/SODALITE-EU/refactoring-ml
Development status	Released version - v0.1 Rule-based refactoring; Extension to RuBiS Cloud benchmark application; Preliminary Machine learning based performance prediction model
Deployment status	Deployed locally and in the Cloud Testbed

Integration status	Partially Integrated Integrated with Refactoring Option Discoverer
Integration issues/dependencies	This module depends on Refactoring Option Discoverer, Node Manager, Deployment Preparation API, Semantic Knowledge Base, Semantic Reasoner, Monitoring Agent
Next steps	Integrate with Node Manager and Monitoring Agent; Complete the first milestone of the machine learning based approach for refactoring decision making; Complete the refactoring logic required for Vehicle IoT Use case.

3.3.4 Node Manager

This component is deployed on each machine deployed in the Cloud and oversees the performance of running containers. It performs vertical resource scalability in order to fulfill goals assigned by the Deployment Refactorer. Given proper collected monitoring metrics it exploits control-theory to plan next allocations.

Table 31 - Development status of Node Manager

Module name	**Node Manager**
Github location	https://github.com/SODALITE-EU/refactoring-ct
Development status	Released version - v0.1 Implemented a working version integrated with Kubernetes that supports the control of TensorFlow applications by means of GPUs and CPUs vertical scalability.
Deployment status	NOT STARTED
Integration status	NOT STARTED
Integration issues/dependencies	This module depends on Deployment Refactorer, Orchestrator, and Monitoring Agent.
Next steps	Complete the integration with Monitoring Agent and Deployment Refactorer

3.3.5 Refactoring Option Discoverer

The refactoring option discoverer can discover the substitutable nodes (refactoring options) for the nodes in the deployment model of an application .

Table 32 - Development status of Refactoring Option Discoverer

Module name	**Refactoring Option Discoverer**
Github location	https://github.com/SODALITE-EU/refactoring-option-discoverer
Development status	Released version - v0.1 Discover compute and software nodes matching a given set of constraints
Deployment status	Deployed locally and in the Cloud Testbed
Integration status	Partially Integrated Integrated with the Semantic Reasoning Engine. Used by the Deployment Refactorer.
Integration issues/dependencies	This module depends on the Semantic Knowledge Base and Semantic Reasoner.
Next steps	Improve the constraint-based discovery of new deployment options (semantic-matchmaking); Support pattern-based discovery of new deployment options.

3.3.6 xOpera REST API

The xOpera orchestrator is a light-weight TOSCA compliant orchestrator with a CLI interface. For a easier integration with other SODALITE components a REST API was developed to enable this kind of integration. The xOpera REST API was also dockerized and deployed with xOpera on the SODALITE cloud testbed.

Table 33 - Development status of xOpera REST API

Module name	**xOpera REST API**
Github location	https://github.com/SODALITE-EU/orchestrator https://github.com/SODALITE-EU/xopera-rest-api
Development status	Released version - v0.1 Prototype of the REST API for xOpera orchestrator featuring blueprint registration and persistence, session management, status of deployment, history of deployment, documented with swagger
Deployment status	Deployed in the Cloud testbed
Integration status	Partially integrated

Integration issues/dependencies	xOpera orchestrator
Next steps	Extend support for TOSCA CSAR files, follow the xOpera development by introducing features with special focus on security.

4 Development Status of the Demonstrating Use Cases

This section provides the development status of the three demonstrating use cases of SODALITE: POLIMI Snow, USTUTT Virtual Clinical Trial and ADPT Vehicle IoT.

4.1 POLIMI Snow UC

The goal of this use case is to exploit the operational value of information derived from public web media content, specifically from mountain images contributed by users and existing webcams, to support environmental decision making in a snow-dominated context. An automatic system crawls geo-located images from heterogeneous sources at scale, checks the presence of mountains in each photo and extracts a snow mask from the portion of the image denoting a mountain.
Two main image sources are used: touristic webcams in the Alpine area and geo-tagged user-generated mountain photos in Flickr in a 300 x 160 km Alpine region. Figure 8 shows the different components of the pipeline.

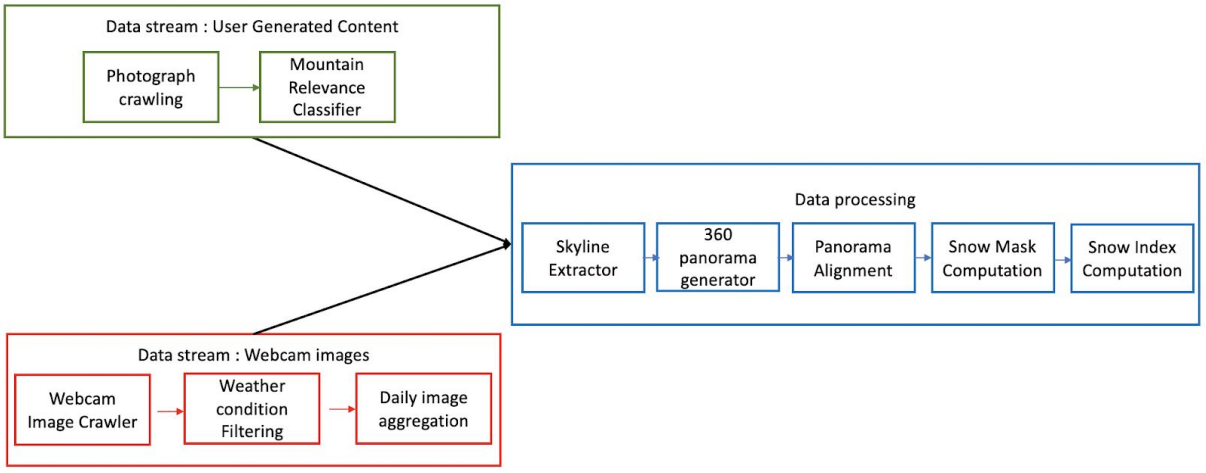

Figure 8 - Components of the Snow Use Case pipeline.

For the first year, a reduced version of the pipeline is delivered, which contains a relevant subset of the components as presented in Figure 9.

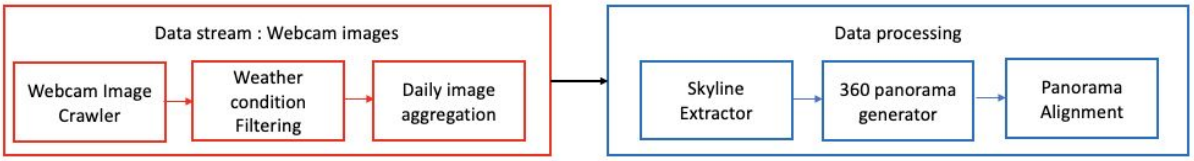

Figure 9 - Initial version of the pipeline as a sub-group of the components of the original one.

Our work so far is based on the implementation plan presented in deliverable D6.1 SODALITE Platform and Use Case Implementation Plan. In Figure 10 we present the planned schedule with

the released components highlighted. All components planned for the Y1 were developed according to the schedule.

ACTIVITY	Start	End		Y1												Y2											
WP6 (Integration and validation)			Deliverable	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19	M20	M21	M22	M23	M24
T6.3 Use Case Implementation (Snow UC)	**4**	**36**																									
User Generated image Crawler (UGIC)	13	16	D6.3																								
Mountain relevance classifier (MRC)	15	17	D6.3																								
WebCam image crawler (WIC)	6	9	D6.3																								
Weather condition filter (WCF)	8	11	D6.3																								
Daily median image aggregation (DMIA)	11	12	D6.2																								
Skyline Extraction (MIGR-SE)	2	7	D6.2																								
360 panorama generation (MIGR-360 PG)	3	6	D6.2																								
Peak Alignement (MIGR-PA)	5	7	D6.3																								
Snow Mask Computation (SMC)	12	15	D6.3																								
Snow Index Computation	14	16	D6.3																								
Pipeline coodination	18	22	D6.3																								
Use Case and Architecture Evaluation	**7**	**36**																									
Baseline Measurements	7	15	D6.2, D6.3																								
Continuous Benchmarking	16	36	D6.3, D6.4																								
Validation and Evaluation of the SODALITE Architecture	9	36	D6.2, D6.3, D6.4																								

Figure 10 - Implementation Plan of Snow use case.

A review of the developed components, which are being deployed in the testbeds, is presented in the subsequent sections.

4.1.1. WebCam crawler

Public webcams expose a URL which returns the most recent available image. The webcam crawler:

- Loads the list of all the webcams in the dataset and starts asynchronous loops, one for each webcam.
- At each loop iteration, it checks the corresponding webcam image and adds the image to the dataset if it is changed w.r.t. the previous iteration, then idles and starts over again. Since downloading the entire image for each iteration would consume unnecessary bandwidth, the check is performed only on a portion of the image. For example, only the first 5KB of the image are downloaded, hashed and compared to the previous webcam hash: if the hash is different, it is saved as the new hash and the rest of the image is downloaded. After the crawler boots, the first image acquired from every webcam is discarded, as there are no guarantees on its timestamp (some webcams, due to failures, propose the same images for days or months).

In Table 34, a summary of the webcam image crawler (WIC) component is provided.

Table 34 - Webcam image crawler component summary

Input	A list of webcams' endpoints
Processing	For each webcam: • Connect to the each webcam service to download the first 5KB of an image • Generate the hash of the downloaded portion and compare with last downloaded image • Compare hash of the two images, if the two hashes are equal, skip • Download and save the entire image

	• Wait 1’
Output	Images for each webcam saved on disk
Implementation technologies and languages	• JavaScript • NodeJS

Examples of WebCam images are shown in Figure 11

Figure 11 - Webcam images examples.

4.1.2. Weather condition filter

Due to bad weather conditions that can significantly affect short and long-range visibility (e.g., clouds, heavy rains and snowfalls), only a fraction of the images can be exploited as a reliable source of information for estimating snow cover. Examples of images with good and bad weather are shown in Figure 12. The *weather condition filter* is based on the assumption that if visibility is sufficiently good, the skyline mountain profile is not occluded (Table 35).

Figure 12 - Webcam image with good weather (first two on the left) and bad weather (two on the right)

Table 35 - Weather filter component summary

Input	A webcam image The binary mask corresponds to a particular webcam.
Processing	• The edge map of the input image is computed. • The skyline visibility value is computed by comparing with the binary mask
Output	The database is updated by setting the visibility (boolean) of the image to 0 (bad weather) or 1 (good weather).
Implementation technologies and languages	• Python

4.1.3. Daily median aggregation

Good weather images might suffer from challenging illumination conditions (such as solar glare and shadows) and moving obstacles (such as clouds and persons in front of the webcam). At the same time, snow cover changes slowly over time, so that one measurement per day is sufficient. Therefore, the Daily Median Image Aggregation (DMIA) aggregates the images collected by a webcam over one day, to obtain a single representative image to be used for further analysis. A median of the images can deal with images taken in different conditions, removing transient occlusions and glares. Given N good daily weather images the Daily Median Image (DMI) is obtained by applying the median operator along the temporal dimension. The component is summarized in Table 36.

Table 36 - Daily median image aggregation component summary

Input	A list of images obtained within one day for each single webcam
Processing	• Calculate the global offset of each image with respect to the first image of the day • Adjust each image based on the calculated offset • Calculate and save DMI
Output	DMI for each webcam
Implementation technologies and languages	• Python

Examples of images with the calculated DMI are shown in Figure 13.

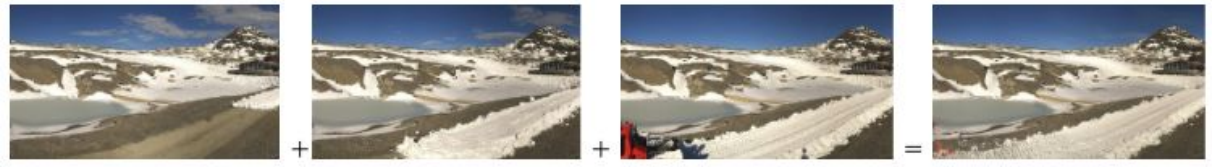

Figure 13 - Example of DMIA applied on three webcam images.

4.1.4. Skyline Extraction

To compute the alignment of the photo and the virtual panorama, these two images should have the same scale, i.e. the same pixel size. Since the photograph and the virtual panorama are taken/generated from the same location, the angular size of the mountains on the photograph and that of the mountains on the panorama are equal by definition. The horizontal FOV (Field Of View) of the photograph is calculated from the focal length and the size of the camera sensor. Then, the photograph is rescaled as the width of the panorama corresponds to a 360° FOV. The next step is to obtain the landscape skyline of a photograph, i.e., the set of all the points that represent the boundary between the terrain slopes and the sky. For this purpose, the image is fed to a binary classifier in the form of patches and it returns its prediction. The training and validation of the classifier is performed using a mountain images dataset, with annotated skyline presence feeding the model with positive and negative patches of a fixed size [9].
In Table 37, a summary of the skyline extraction (MIGR-SE) component is presented.

Table 37 - Skyline extraction component summary

Input	Image
Processing	• Provide the image to the classifier that will output a mask indicating for each pixel whether it corresponds to the skyline or not.
Output	Skyline mask
Implementation technologies and languages	• Java • OPENCV

Example of skyline extraction is presented in Figure 14.

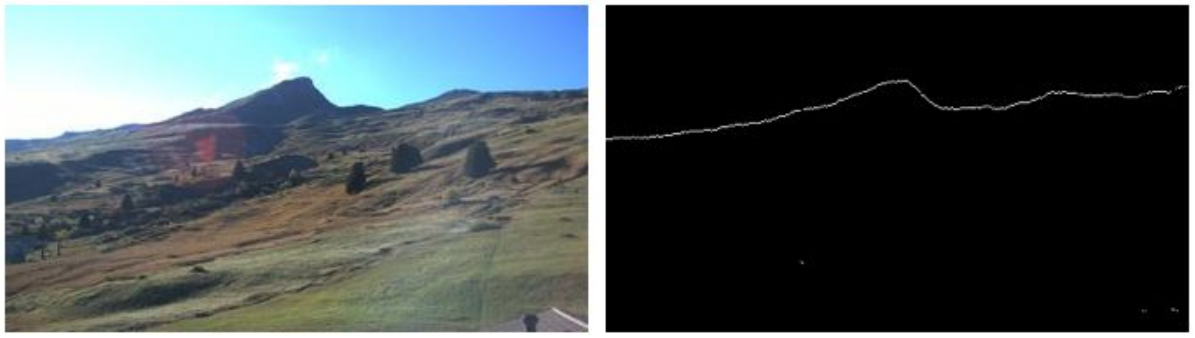

Figure 14 - Example of skyline extracted from webcam image.

To measure its processing and response time, a small set of 20 images was created. Such images are on average 1MB in size. We measure the time that it takes from sending the request to writing the result to the file system. In this case, the resulting image in jpg format contains the skyline mask.

On an average, the component takes 1.5 seconds to process an image of size 1MB. If we subtract the time the client needs to prepare the request (0.2s) from the total time it took to send, process the request and write the results, the component takes 1.3 seconds on average.

4.1.5. 360 Panorama generation

From the coordinates of the picture, we process the 360° panoramic view of the terrain using the Digital Elevation Model (DEM) of the publicly available terrain. The functionality is exposed as a service.
The rendering model is composed of a C++ program that exploits hardware-accelerated graphics capabilities by invoking shader programs to perform the rendering operations.
This component uses the DEM files provided by NASA.
In Table 38, the 360° panorama generation (MIGR-360PG) component is summarized.

Table 38 - ° panorama generation component summary.

Input	Latitude and Longitude Terrain Model Precision (3'' or 1'') Relative altitude of the viewer (meters from the ground) Maximum visible distance
Processing	• Loading of DEM, • Initialisation or OpenGL rendering, • Execution of OpenGL rendering, • Extraction and conversion of results
Output	360° panorama
Implementation technologies and languages	• C++ 14 compliant • OPENGL, EGL • Java, JavaScript and NodeJS (to make the panorama web-accessible)

Examples of render generated images based on latitude and longitude of a webcam image are shown in Figure 15. The alignment between the panorama and the skyline of the mountain image is the next step.

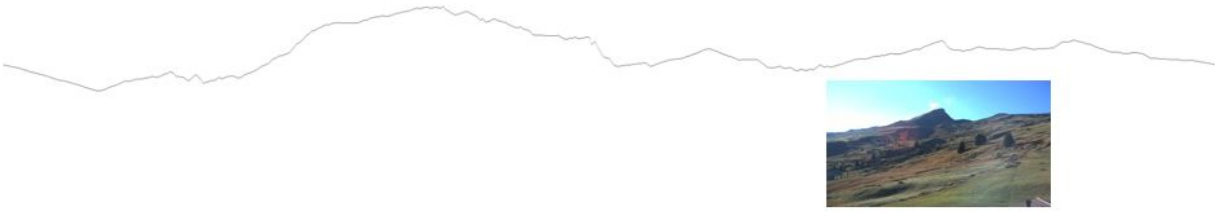

Figure 15 - Example of render-generated from webcam coordinates with the webcam image of reference.

4.1.6. Panorama Alignment

The alignment can be seen as the search for the correct overlap between two cylinders (assuming the zero tilt of the photograph): one containing the 360° panorama and the other one containing the photo, suitably scaled. It is useful to obtain the mountain mask that will be the input to the successive components to calculate the Snow Index.
In Table 39 the peak alignment (MIGR-PA) component is described.

Table 39 - Peak alignment component summary

Input	An image with its corresponding skyline annotation and the 360° panorama corresponding to its location
Processing	• Perform global alignment between skyline and panorama

Output	M = A mask indicating pixels that correspond to the mountain surface.
Implementation technologies and languages	• Java • OPENCV

Examples of mountain mask images computed from a webcam image are presented in Figure 16.

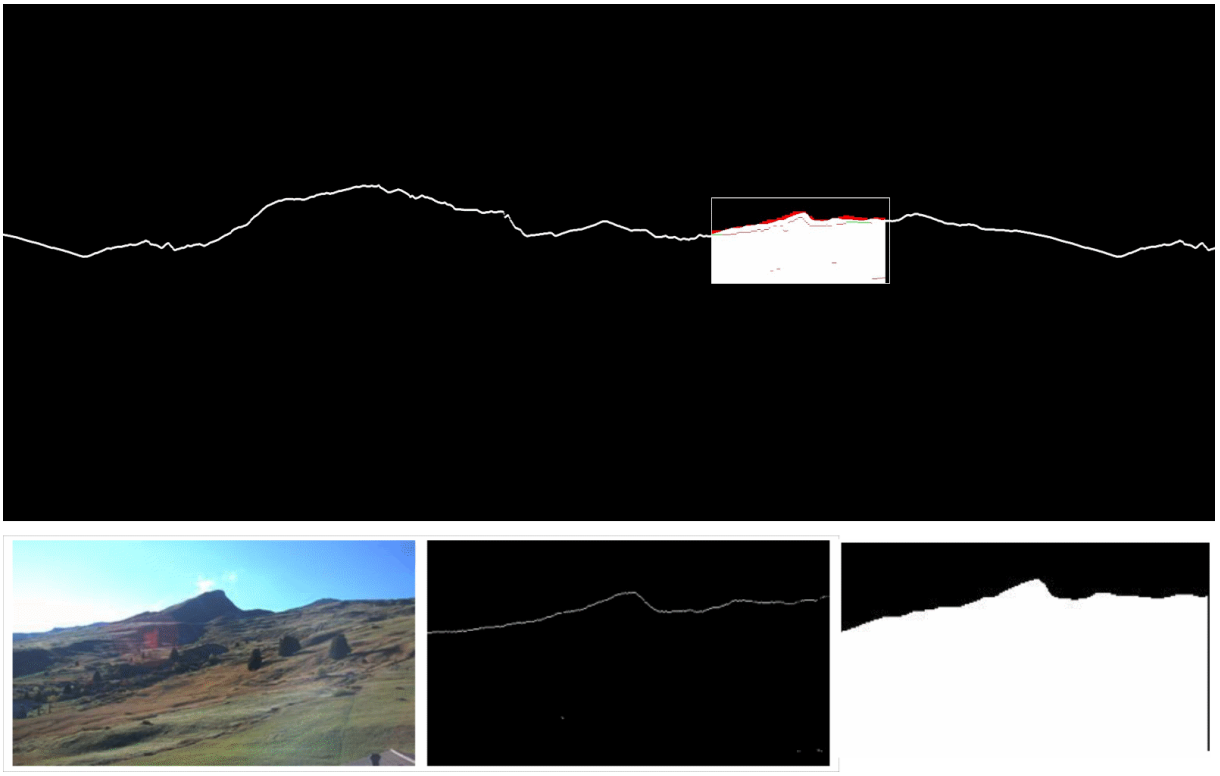

Figure 16 - Example of mountain mask extracted from webcam image based on the skyline.

To measure its processing and response time, the same set of 20 images used in the Skyline Extractor component was employed.

We measure the time that it takes to send the request and write the result to the file system. In this case, the result is a json file containing the mountain mask (along with other attributes) that will be used to compute the snow index at a later point.

On average, the component takes 13.2 seconds to process a 1MB image. This time can be split into different parts of the process. If the time the client needed to prepare the request (0.2s) is subtracted from the total time it took to send, process the request and write the results, the component takes on average 13 seconds.

4.2 USTUTT Virtual Clinical Trial UC

The “In-silico clinical trials for spinal operations” use case targets the development of a simulation process chain supporting in-silico clinical trials of bone-implant-systems in Neurosurgery, Orthopedics and Osteosynthesis. It deals with the analysis and assessment of screw-rod fixation systems for instrumented mono- and bi-segmental fusion of the lumbar spine by means of continuum mechanical simulation methods. The components of the complete simulation process chain as depicted in Figure 17 are described in the deliverable D6.1 [1] (Section 4.2).

In this document we describe the implementation and development done so far as well as additional steps, which had to be introduced into the processing pipeline. These steps had to be introduced before the “Extraction” step due to in depth analysis of the “Imaging Data”. In addition, it became necessary to reorganize the data flow around the components "Density Mapping", "Applying Boundary Conditions" and "Probabilistic Mapping", since the original program versions had to be modified to allow the use of Code_Aster. Some of the components, namely Density Mapping, Probabilistic Mapping and Solver, were deployed and run in the HPC testbed, and their baseline performance was measured and reported in the deliverable D3.3 "Prototype of application and infrastructure performance models - First version".

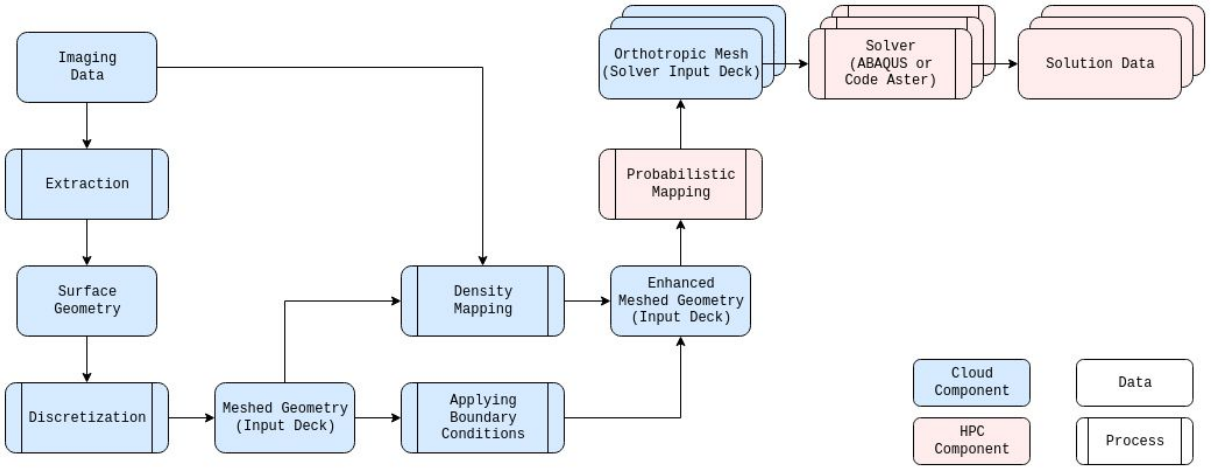

Figure 17 - Schema of the Virtual Clinical Trial use case pipeline.

4.2.1 Extraction

Since the initial step “Extraction” is directly based on the clinical imaging datasets, they were first analysed with respect to their content as well as their quality using 3D Slicer[20].

Both datasets are provided by the clinic for neurosurgery at the university medical center Knappschaftskrankenhaus Bochum, Germany[21] as DICOM[22] (Digital Imaging and Communications in Medicine) datasets. While there is no standard on how DICOM datasets have to be stored physically, they are organized logically by header information stored within each image. In Figure 18 it can be seen that the datasets are logically organized firstly by patient and secondly by so called studies, which in turn can contain several series.

Even though the datasets contain several imaging modalities like full body X-ray images, dual energy X-ray absorptiometry (DXA) scans, magnetic resonance imaging (MRI) data as well as

[20] https://www.slicer.org/
[21] https://www.kk-bochum.de/en/Clinics-Centers-Departments/Clinics/Neurosurgery_Clinic/index.php
[22] https://www.dicomstandard.org/ - *“DICOM is the international standard to transmit, store, retrieve, print, process, and display medical imaging information”*

computer tomography (CT) images, we currently concentrate on the CT-images as they are the basis for the three dimensional (3D) reconstruction of the spine's bone structures.

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

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

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

Figure 18 - Logical organization of patient datasets.

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

The left subfigures of Figure 19 show an isosurface along with two cutting planes and resulting contour lines. As a basis for this feature extraction, the CT-series from the pre-operative study, reconstructed in the x-y-plane was taken. In the bottom-right subfigure the contour line of a vertebra in the x-y-plane is shown in detail. It can be seen that the vertebra's contour was extracted smoothly. In the top-right subfigure the contour line of the same vertebra, taken on the y-z cutting plane is shown. Here it can be seen that the vertebra's contour could not be extracted smoothly due to the low resolution of this CT-series in the y-z- direction.

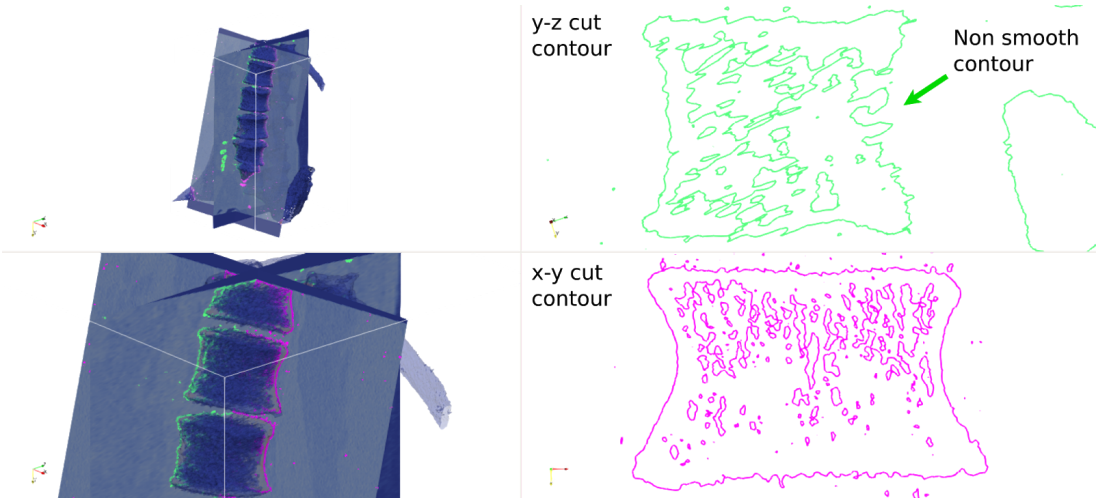

Figure 19 - y-z cut contour line and x-y cut contour line - x-y reconstructed dataset.

For the results shown in Figure 20 the same feature extraction as shown in Figure 19 was applied to the CT-series from the pre-operative study reconstructed in the y-z plane. In contrast to Figure 19, in the top right subfigure of Figure 20 it can be seen that the contour line on the y-z cutting plane in that case reproduces the vertebra's contour smoothly. In the lower right subfigure of Figure 20, however it can be seen that the contour in the x-y plane is no longer reproduced smoothly once the feature extraction is based on the CT-series reconstructed in the y-z plane.

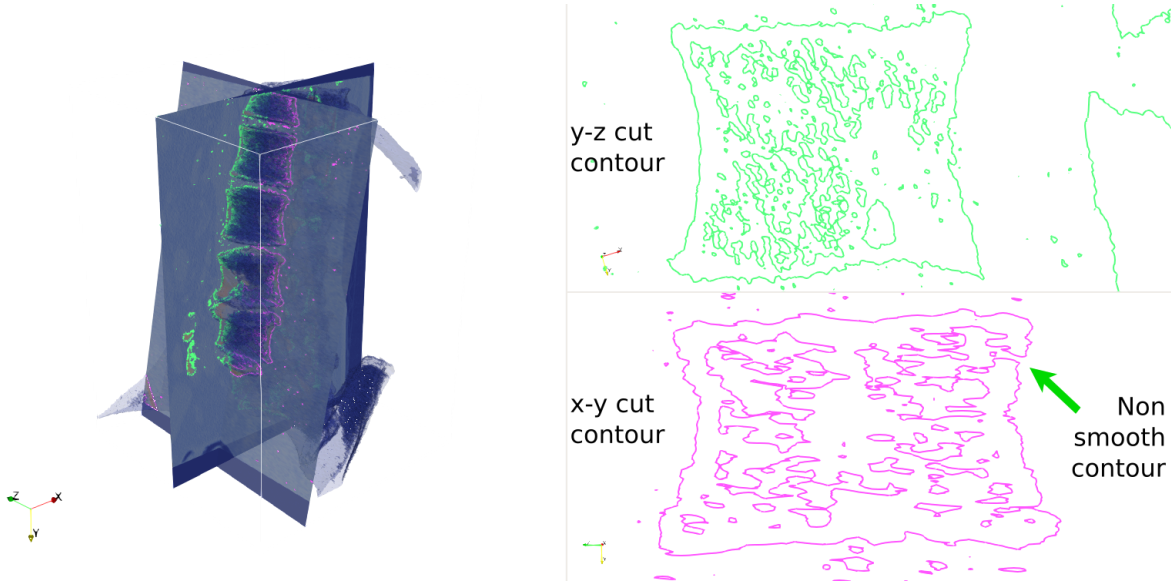

Figure 20 - y-z cut contour line and x-y cut contour line - y-z reconstructed dataset.

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

We consider to exploit different reconstruction planes in combination with polynomial interpolation and anisotropic diffusion image filtering. This is based on image analysis and also past experiences in the implementation of a merge filter for the three CT-series in each study. These two filtering steps will be introduced right before the Extraction as part of the Image Processing and Filtering and will produce an "Enhanced Image-Data", as shown in Figure 21.

Due to the higher complexity of the reconstruction and discretization steps, it was decided to postpone the implementation of the automatic extraction and discretization steps for the first prototype and to initially perform manual segmentation, geometry extraction and discretization. This means that the simulation process chain currently starts at the "Density Mapping" step, the development status of which is described in the next subsection.

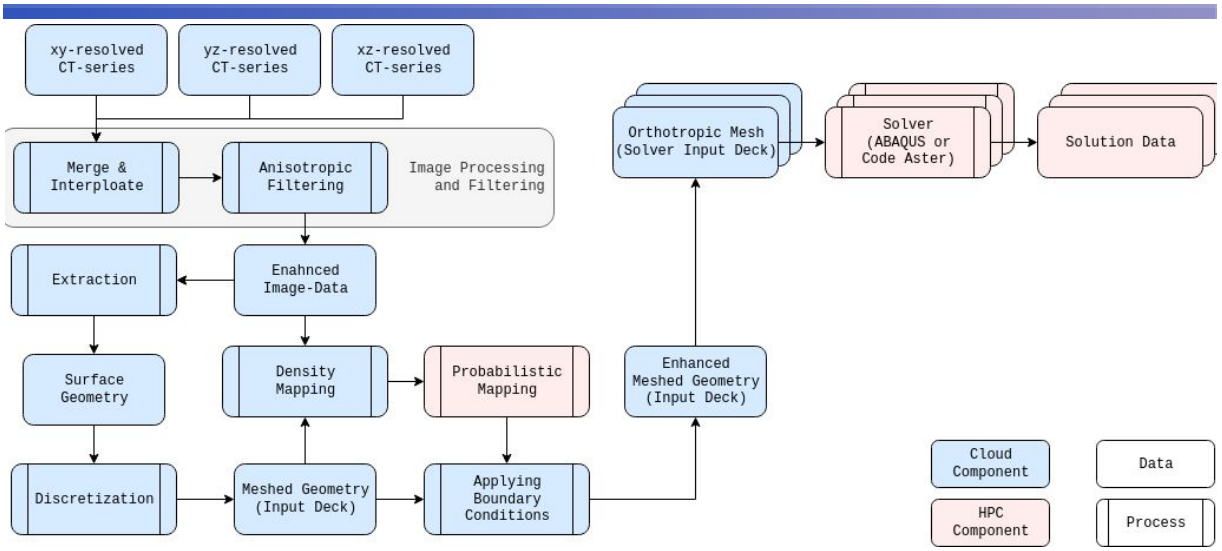

Figure 21 - Schema of the Virtual Clinical Trial use case pipeline with additional steps.

4.2.2 Density Mapping

In this step, the three input decks as well as the three CT data sets are taken as input. By means of direct geometrical mapping, the grayscale distribution of the respective CT data set is mapped onto the volume mesh provided by the discretization. After the mapping, each element in the volume mesh holds a density value. The “Density Mapping” step is written in Fortran 2003. Its initial implementation was started in 2007 and is documented by Schneider [10]. The algorithmic principles are published by Schneider et al. [11]

Since the original implementation relied on an internal data format and was not able to load DICOM datasets with the given complex structure, the data input part had to be modified. 3D Slicer, which was used for the analysis of the datasets, has a very advanced DICOM reader implemented and is also able to export the loaded data as VTK STRUCTURED_POINTS[23]. Because of these features and since it is planned to implement the “Anisotropic filtering” step for image improvement based on VTK , it was decided to add the capability to load VTK STRUCTURED_POINTS data in VTK XML format to the “Density Mapping” component.

By this extension, the “Density Mapping” step is now able to map the density distribution from the CT-data on the “Meshed Geometry”. This results basically in one density value per element.

To give an impression of the resulting density distribution, the final mapping result is visualized side by side with the original data in Figure 22. Since the probabilistic mapping step acts locally on each element, no topological information has to be passed on between the density mapping component and the probabilistic mapping component. Everything that has to be passed is a list that contains one integer value per element representing its averaged density value. These values have to be passed as a file containing binary 64-bit integer values.

The initial version of Density Mapping was encapsulated into a Singularity container and was run on the HPC testbed, taking on average 122.4 seconds to complete.

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

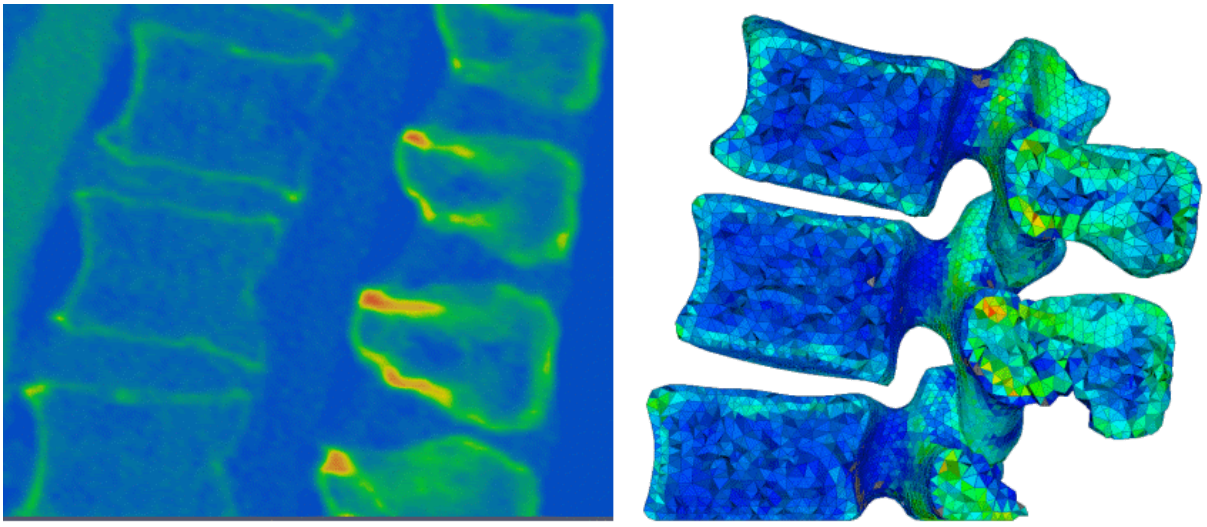

Figure 22 - Density Mapping component - Left: Input data - Right: Mapping result.

4.2.3 Probabilistic Mapping

Based on the input from the “Density Mapping” component, the “Probabilistic Mapping” component produces three probability distributions per element, one for each of the three elastic moduli of an orthotropic stiffness matrix. This is done by means of the transfer functions between density and orthotropic elastic moduli given in Schneider et al. [11]. The low and high bounds and the mode of 95% confidence interval are computed from the aforementioned probability distributions. This finally results in the output of three files where each file contains three elastic moduli for each element.

Since these outputs have to be extended by shear moduli and poisson ratios to form complete, orthotropic material distributions and then become integrated with the “Meshed Geometry” to form a complete Input Deck, i.e. the “Enhanced Meshed Geometry”, the data path in the simulation process chain was changed, as can be seen in Figure 21. Instead of treating the “Enhanced Meshed Geometry” directly with the “Probabilistic Mapping” component and passing its output directly to the “*Solver*” component, its output is now sent to the “Applying Boundary Conditions” component. This change in the data flow was decided during the parallel development of the "Probabilistic Mapping" component and the prototype of the Code_Aster[24] model, which now forms the basis for the current ongoing development of the "Applying Boundary Conditions" component. During the development, it was recognized that it would amount to less effort to add the integration of the results from the “Probabilistic Mapping” component to the "Applying Boundary Conditions" component than to implement the treatment of the “Enhanced Meshed Geometry” by the “Probabilistic Mapping” component. This is due to the fact that in the "Density Mapping" component, from which we derive the "Applying Boundary Conditions" component, the algorithmic part is already implemented and only the output part needs to be changed.

The initial implementation of Probabilistic Mapping allows parallelisation via MPI. MPI was utilized over different cores within a single node of the HPC testbed. Better performance was achieved when running it with a 16 MPI ranks configuration, taking on average 21.42 seconds to complete.

4.2.4 Applying Boundary Conditions

The “Applying Boundary Conditions” component is derived from the original implementation of the “Density Mapping” component. The component will integrate each of the three output files of the “Probabilistic Mapping” component with the “Meshed Geometry” and additional information

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

needed by the “Solver” component. This will result in three input decks, which can then be loaded by the “Solver” component directly.

4.2.5 Solver

In the “Solver” component, a reference model prototype had been developed for ABAQUS[25] and its fully featured transfer to Code_Aster is currently under development. Initial results of this development are displayed in Figure 23.

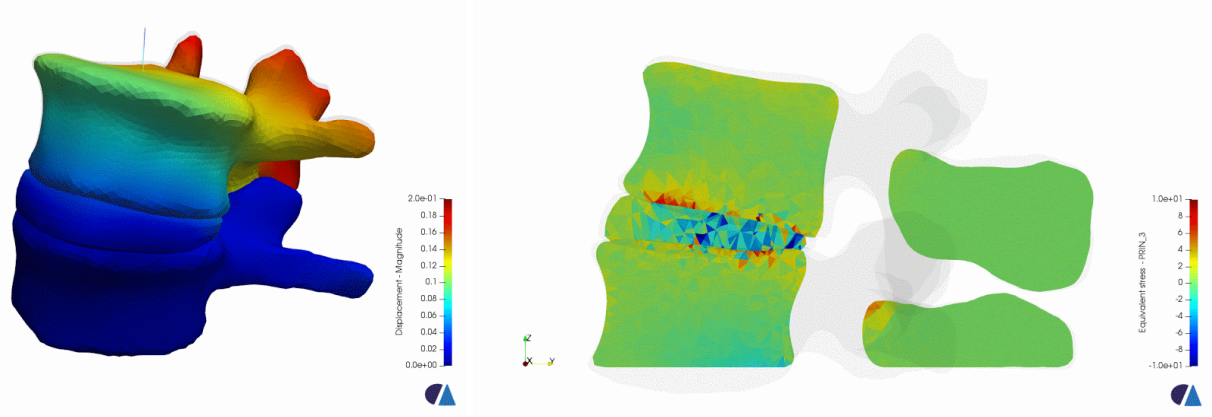

Figure 23 - Displacement (left) and 3rd principal stress (right) results of the Code_Aster prototype model.

Additionally, the coupling of the "Applying Boundary Conditions" component to the “Solver” component is currently under development. The implementation of the coupling requires basically the implementation of the mesh output in Code_Aster’s HDF5-based MED-format[26] and the implementation of the output of Code_Aster’s input parameter file to the "Applying Boundary Conditions" component.

The Code_Aster based solver was also containerized with Singularity and run on the HPC testbed using only a single core. The future configuration will include MPI parallelisation and will be able to run on multiple cores. To execute the Solver, it took 789.17 seconds on average.

4.3 ADPT Vehicle IoT UC

The Vehicle IoT use case focuses on situationally-aware processing of data subject to various latency, security, and regulatory constraints within a connected vehicle. The precise requirements of the workload are subject to change based on factors such as the regulatory environment, the privacy preferences of the driver, resource availability, requisite processing power, connectivity state, etc. This use case targets mixed Cloud/Edge deployment models and focuses on dynamic adaptation and run-time redeployment/reconfiguration in order to satisfy both its performance and compliance requirements.

The implementation plan for the UC is outlined in Figure 24 below. At present, all of the planned Y1 work was carried out on schedule, with Y2 work beginning as planned. It is not expected that there will be any significant deviations or changes required for Y2.

[25] https://www.3ds.com/de/produkte-und-services/simulia/produkte/abaqus/
[26] https://docs.salome-platform.org/latest/dev/MEDCoupling/developer/index.html

While the existing application components are capable of dynamic adaptation and reconfiguration of the application-level (e.g. turning on/off certain features, dynamically switching between backend instances, etc.), little work has been done on extending this to run-time reconfiguration of the infrastructure and deployment itself.

ACTIVITY				Y1												Y2											
WP6 (Integration and validation)	**Start**	**End**	**Deliverable**	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**	**M8**	**M9**	**M10**	**M11**	**M12**	**M13**	**M14**	**M15**	**M16**	**M17**	**M18**	**M19**	**M20**	**M21**	**M22**	**M23**	**M24**
T6.3 Use Case Implementation (Vehicle IoT UC)	**4**	**36**																									
License Plate Detection Service - Initial	4	6	D6.2																								
Drowsiness Detection Service - Initial	6	8	D6.2																								
Intrusion and Theft Detection Service - Initial	7	10	D6.2																								
Edge Gateway Implementation - Initial	8	12	D6.2																								
Cloud Function Implementation and Cloud Deployment	5	11	D6.2																								
License Plate Detection Service - Intermediate	13	15	D6.3																								
Drowsiness Detection Service - Intermediate	15	17	D6.3																								
Intrusion and Theft Detection Service - Intermediate	16	19	D6.3																								
Edge Gateway Implementation - Intermediate	13	24	D6.3																								
Cloud Function Edge Deployment and Federation	16	24	D6.3																								
Component Integration and Optimization	6	36	D6.2, D6.3, D6.4																								
T6.4 Use Case and Architecture Evaluation	**7**	**36**																									
Baseline Measurements	7	12	D6.2																								
Continuous Benchmarking	13	24	D6.3																								
Validation and Evaluation of the SODALITE Architecture	13	36	D6.3, D6.4																								

Figure 24 - Vehicle IoT UC Implementation Plan

4.3.1. Vehicle Services

A number of vehicle services, dedicated microservices taken from ADPT's KnowGo Car[27] platform, are leveraged within this use case. These are briefly expanded upon in the subsections below. While the functionality for each of these services is self-contained within a microservice, they are, at present, only used on a Cloud backend (specifically, GCP) in production.

In the initial implementation at the end of Y1, all vehicle services are up and running on the SODALITE Cloud testbed, through Ansible Playbooks converted from a Docker Compose file. An Edge instance of a corresponding service can be triggered on-demand via accessing the OpenFaaS function path on the Edge gateway directly.

The deployment pattern at the start of SODALITE, with a single Cloud backend implementing the API Gateway pattern is shown in Figure 25 below:

[27] https://www.knowgo.io

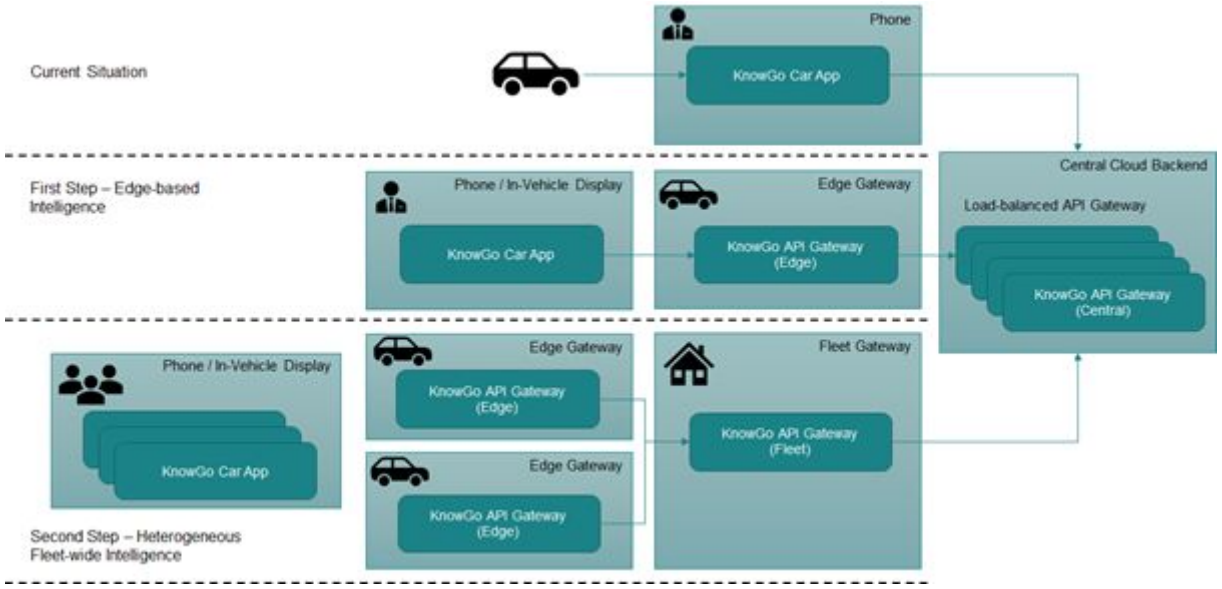

Figure 25 - Schema of the Vehicle IoT use case deployment phases

During Y1 of the project, the focus has been on two key areas: (1) preparing an Edge-based instance of the backend and relevant services, and (2) tying the existing use case components into the SODALITE stack in order to enable SODALITE to directly manage the deployment and configuration of deployed components.

4.3.1.1. License Plate Detection Service

The license plate detection service exposes a simple REST API in which plate image data is submitted, and the text-based plate number returned, together with a measure of confidence in prediction. Images fall into 3 different categories – detectable (confidence >= 90%), partially detectable (able to identify the plate, but unable to extract the plate number accurately, confidence < 90%), and undetectable (unable to detect a plate in the provided image).

For the underlying ML models, two different cases exist: (1) The detection of a plate within the image; and (2) Textual extraction of the plate number from the detected plate. In the case of (1) the OpenCV model itself must be retrained – this is currently not handled, but remains an exercise for Y2, and in the case of partial detection, plates are saved off for manual tagging and re-generation of a Tesseract OCR training sheet. Both re-training scenarios are manual, and while they can support a degree of parallelization at computation time, are unsuited for online learning.

Partially detectable and undetectable images are saved off to storage volumes for periodic scraping for model training.

4.3.1.2. Drowsiness Detection Service

The initial version of the drowsiness detection service implements two components: (1) the video stream producer, which is responsible for connecting to the hardware-specific camera and obtaining a video stream to communicate via a dedicated Kafka queue, and (2) a dedicated microservice that consumes the video stream and carries out real-time analytics on the video frames, tracking the percentage of eye closure over a period of time (in this case, a number of consecutive video frames). At the end of Y1, both components are implemented, and a basic analysis on both Edge Gateway hardware targets is possible. In Y2, it is expected to: (1) expand the stream producer for a wider range of sources; (2) to implement topic authorization on the Kafka topic through the API Gateway; and (3) to support detection on each Edge Gateway configuration, with service deployment adapting to the unique Gateway-specific platform capabilities.

4.3.1.3. Intrusion and Theft Detection Service

The intrusion and theft detection service builds on the face detection model from the drowsiness detection service in order to provide a facial recognition capability specific to the authorized user of the vehicle. In the initial implementation of this component, the user is able to generate their own customized SVC classifier, trained with a number of sample images submitted by the user, and is capable of recognizing the individual's face. The service itself is implemented as a dedicated microservice, which takes an image as input and provides a JSON-encoded response with the results. It is up to the caller to take action based off of the returned detection results, which is left to be elaborated in Y2. The initial implementation has been validated on the Edge Gateway, using cameras directly connected to the Gateway device (elaborated in the following section).

During Y2, we will look to: (1) deploy this service onto the differing Edge Gateway configurations and have the deployment adapt to the unique hardware capabilities of the platform; (2) enable online learning/training of the classifier to mitigate false negatives; and (3) facilitate classifier portability, allowing the vehicle owner to grant authorization to other individuals.

4.3.2. Edge Gateway

A number of different hardware platforms with unique capabilities and configurations are being experimented with for the Edge Gateway implementation, enabling SODALITE to consider the availability of heterogeneous hardware resources not only at the Cloud infrastructure level, but also directly at the Edge.

During Y1, an initial Edge Gateway has been instantiated on a Raspberry Pi 3B+ and an NVIDIA Jetson Nano, as per the hardware evaluation Table 40 below. The implementation of the Edge Gateway has necessitated changes for multi-arch Docker images in the existing application components, as well as a number of changes to the OpenFaaS runtime to support policy-based access control for Cloud Functions through Open Policy Agent (https://openpolicyagent.org) and various language run-times used by the use case (Golang, Dart) – these aspects are further detailed in D7.3.

In Y2, focus will instead shift towards performance and capability-based assessment of the Edge Gateway hardware, allowing deployment patterns to be adapted to reflect the specific capabilities of the specific Edge Gateway implementation.

Table 40 - Differing hardware configurations for the Edge Gateway (Vehicle IoT UC)

	RPi 3B+	**RPi 3B+ + Intel Neural Compute Stick 2 (NCS2)**	**NVIDIA Jetson Nano**	**Google Coral Dev Board**
CPU	1.4 GHz 64-bit Quad-Core ARM Cortex-A53		1.4 GHz 64-bit Quad-Core ARM Cortex-A57 MPCore	1.5 GHz 64-bit Quad-Core ARM Cortex-A53, plus Cortex-M4F
GPU	Broadcom VideoCore IV		128-Core NVIDIA Maxwell	Vivante GC7000 Lite
ML Accelerator	None	Intel NCS2	None	Google Edge TPU
RAM	1GB LPDDR2		4GB LPDDR4	1GB LPDDR4
Performance	21.4 GFLOPS	21.4 GFLOPS (GPU) 4 TOPS (NCS2)	472 GFLOPS	32 GFLOPS (GPU) / 4 TOPS (TPU)
Camera	Raspberry Pi Camera Module V2 (8MP) Raspberry Pi Pi NoIR Camera V2 - infrared version			Coral Camera (5MP)
Networking	WiFi: 802.11ac, Bluetooth: 4.2			

4.3.3. Microservices -> Cloud Functions

During Y1 of the project, the initial set of vehicle microservices have each been wrapped into deployable Cloud Functions under OpenFaaS, allowing them to be deployed on-demand directly on the Edge Gateway in response to client-initiated events.

4.3.4. Region-aware Gateway Routing

The Vehicle IoT use case makes use of multiple region-specific backend Gateways for both compliance and QoS purposes, and uses region-aware routing through a single entry point into the API Gateway in order to: (a) determine the appropriate backend for a specific client request; (b) deploy new region instances when no suitable backend exists; and (c) force the client to reconfigure or otherwise disable specific application logic which would bring it out of compliance in the current environment.

This functionality is implemented through an open source (Apache 2.0 licensed) region router, developed by ADPT (https://github.com/adaptant-labs/go-region-router), which makes use of HashiCorp's Consul (https://consul.io) service registry to monitor the deployment state of backend instances and matches the in-bound client request with a suitable backend. The client may set the region identifier in its request header directly based on reverse geocoding of geolocation at the

client side or may be queried by the router based on the inbound requesting IP. An overview of location-aware routing in a production capacity is outlined in Figure 26 below:

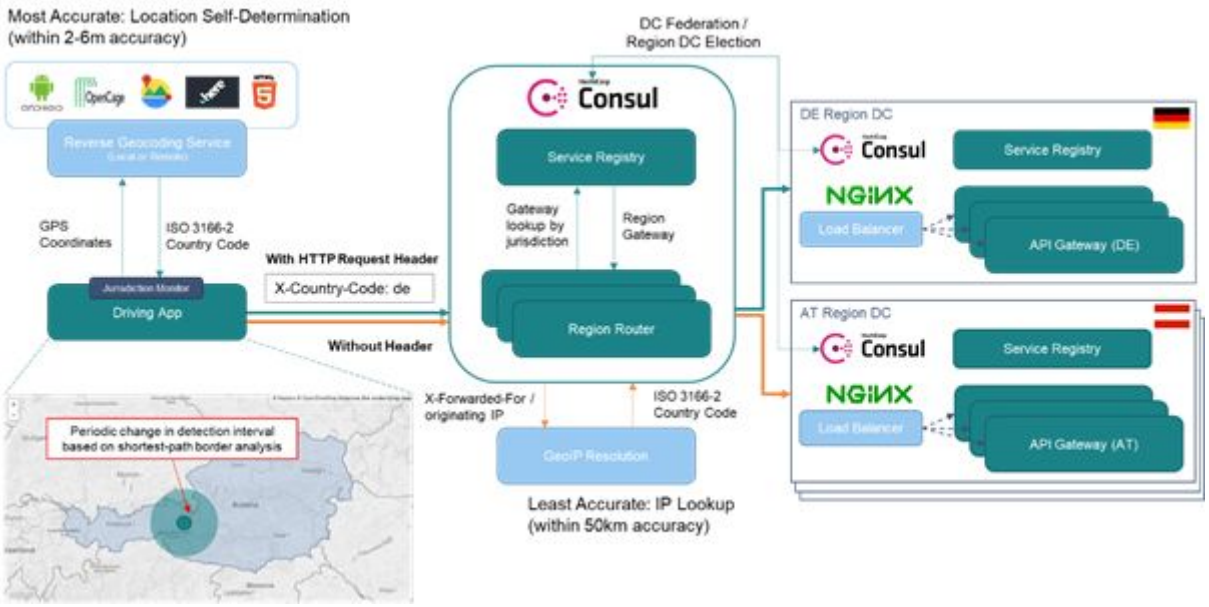

Figure 26 - Location-aware Multi-DC Region Routing

Under SODALITE in Y1, the router has been extended to include notifications to a designated REST API endpoint in the case where no suitable deployment is found, allowing for the SODALITE refactorer to dynamically initiate a new region deployment based on the defined TOSCA blueprint. The following payload is included in the POST body to the notification endpoint (locations are ISO 3166-1 alpha-2 encoded, in this case, demonstrating a need to deploy a new instance in Italy):

```
{
      "event_type": "DeploymentNeeded",
      "new_location": "it"
}
```

In Y2, it is expected that the router will be further enhanced with a suitable pub/sub interface which will integrate directly with the SODALITE run-time monitor. It may also be extended to acquire knowledge of deployment facts from other sources, depending on the technical direction ultimately taken by SODALITE.

5 Implementation status of the First Prototype

Table 41 depicts the actual coverage of the SODALITE UML use cases, which were introduced within the context of deliverable D2.1 “Requirements, KPIs, evaluation plan and architecture - First version” (see Section 2 “Requirement Elicitation and Analysis”) [2] and will be run as part of the evaluation of the SODALITE platform, by the three SODALITE demonstrating use cases at the end of project month M12. Please note that not all of the defined UML cases were tested by the demonstrating use case owners during the first project year, as the initial versions for some of them (UC12, UC14 and UC15) will be released in M18 of the project. With respect to the ones (i.e. initial versions of UML cases) that were released in M12:

1. The Virtual Clinical Trial use case focused on UC5 (Predict and Correct Bugs) and has demonstrated that the current implementation of this UC has been able to correct the initial TOSCA blueprints and Ansible playbooks prepared for the Virtual Clinical Trial. Moreover, this use case has also experimented with UC6 (Execute Provisioning, Deployment and Configuration) and UC7 (Start Application) showing that the corrected set of blueprints and playbooks can actually support the deployment and the execution of the Virtual Clinical Trial application on the HPC testbed.
2. The Snow use case focused on the modeling activities showing that it is possible to define a deployment model that fulfills the needs of the application (UC1), to select the resources to be used at runtime (UC2), to obtain generated IaC artefacts (UC3, UC4) and runtime images in the private image registry (UC16), and finally run and monitor the deployment (UC6-UC8) on the Cloud testbed. Additionally, partial redeployment (UC10) was executed by means of the WP5 toolset.
3. The Vehicle IoT use case focused on the runtime monitoring and refactoring aspects and has identified and applied refactoring options at runtime (UC9), providing notice to the runtime monitoring (UC8), and triggering redeployment and re-configuration (UC6) of the deployed components on the Cloud testbed.
4. Testbed Providers, acting as Resource Experts and Quality Experts, modeled the Cloud and HPC resources available in the testbed and added them into the Knowledge Base (UC13), as well as contribute to the Knowledge Base with the classification of typical bugs and their resolutions (UC11).

Table 41 - Coverage of the SODALITE UML use cases by the project’s demonstrating use cases by M12

UML Use Case	Virtual Clinical Trial	Snow	Vehicle IoT	Testbed Providers
UC1 Define Application Deployment Model (WP3)				
UC2 Select Resources (WP3)				
UC3 Generate IaC code (WP4)				
UC4 Verify IaC (WP4)				
UC5 Predict and Correct Bugs (WP4)				
UC6 Execute Provisioning, Deployment and Configuration (WP5)				
UC7 Start Application (WP5)				
UC8 Monitor Runtime (WP5)				
UC9 Identify Refactoring Options (WP5)				

UC10 Execute Partial Redeployment (WP5)				
UC11 Define IaC Bugs Taxonomy (WP4)				
UC12 Map Resources and Optimisations (WP3)	Will be released in M18			
UC13 Model Resources (WP3)				
UC14 Estimate Quality Characteristics of Applications and Workload (WP3)	Will be released in M18			
UC15 Statically Optimize Application and Deployment (WP4)	Will be released in M18			
UC16 Build Runtime images (WP4)				

At the end of Y1, all of the use cases are able to partially run on the testbeds (in-line with the expected state at MS3-1st SODALITE Prototype), and baselines for the individual use cases have been gathered and elaborated in D3.3. The future plan is to ensure that all the use cases provide increasing coverage of the individual UML use cases and are able to continuously evaluate them against the Y1 baselines. Below is the description of how the use case providers have experienced the development with the SODALITE platform.

Virtual Clinical Trial UC

For the Virtual Clinical Trial use case, we have prepared Singularity images for each component and developed TOSCA blueprints and Ansible playbooks. Since this use case is HPC oriented, we experienced a steep learning curve during the development due to the current focus of SODALITE platform to the cloud infrastructure with respect to IaC generation and image building. However, it helped to identify additional requirements for the Runtime layer, such as support for TOSCA workflows, which is now part of ongoing work of the SODALITE Orchestrator.

The developed blueprints and playbooks were submitted to the Bug Predictor component, which checked them and identified several bugs, especially in the Ansible playbooks, which were iteratively solved. For example, for getting the job status, we used pipe commands (qstat command piped with grep command), but did not consider possible failures in the pipeline causing the wrong exit status. Therefore, we included the pipefail option (set -o pipefail) before the commands run to ensure correct failure handling.

Snow UC

For the Snow use case, we partially modeled the pipeline using the knowledge base-empowered SODALITE IDE. The result is a JSON file, containing all the data and meta-data needed for the deployment and links to proper Ansible playbooks, that enable the setup and management of Snow components. This JSON is then sent to the iac-blueprint-builder component that converts it to a proper TOSCA blueprint (in the near future more formats will be supported) that can be used by an orchestrator (e.g., xOpera) to enact the deployment.

Currently, the main open issue is to connect the various components of the pipeline. The original and current deployment of Snow was limited to two VMs, and components were connected using local hard-drives and ad-hoc methods. In the next months, we are going to address this issue by using more scalable and de-coupled ways to connect components (e.g., using queues or distributed DBs) so that the portability of the use case and its runtime management could be more effective.

At runtime, we verified that the generated blueprints could actually automate the deployment and configuration of the whole pipeline and we started and monitored such pipeline. Furthermore, we focused on the Skyline Extraction component by testing how it could be dynamically scaled in order to fulfill requirements on the response time. The current approach implemented in the Node

Manager exploits heuristics and control-theory to dynamically change the CPU and GPU allocations according to the workload and context changes. The results retrieved with Skyline Extraction are promising and we plan to extend it to all the other components of the pipeline that can be executed on heterogeneous resources (CPU/GPU).

Vehicle IoT UC

For the Vehicle IoT use case, we have ensured that use case components are deployable from the SODALITE container registry, and developed TOSCA blueprints and Ansible playbooks (converted from an existing Docker Compose stack) to enable deployment on the Cloud testbed. One of the key use case components (the region-aware router that sits in front of the API Gateway) has been extended to provide deployment hints to the run-time monitor and refactorer, which in turn communicates with xOpera to obtain the TOSCA blueprint and trigger a new deployment when the vehicle enters a country where no deployment exists. This is an important milestone for the use case, as it allows the use case deployment to be managed directly by SODALITE, and provides linkage between Infrastructure-level adaptation (reconfiguration/re-deployment) and the pre-existing service-level adaptation capabilities built in to the existing use case components (building on the state of the art attained at the end of the RestAssured H2020 project[28]).

6 Conclusions

This deliverable presents the next iteration of the SODALITE platform, and specifically describes the initial implementation of the components making up the platform and of the project's three demonstrating use cases. D6.2, along with the other SODALITE technical deliverables which will be submitted by the end of the first project year (i.e., in M12; D2.4 "Guidelines for contributors to the SODALITE framework", D3.1 "First version of ontologies and semantic repository", D3.3 "Prototype of application and infrastructure performance models", D4.1 "IaC Management - initial version", D5.1 "Application deployment and dynamic runtime - Initial version" and D6.5 "SODALITE framework - First version"), explains the current status of the SODALITE system in terms of repositories, functionalities, procedures and workflows of the First Prototype. Specifically, deliverable D6.2 introduced new insights with respect to the SODALITE development environment such as the testbed setup and configuration, code repositories and the newly introduced continuous integration and deployment platform (Section 2). Furthermore, this document discussed the first prototype implementation of the overall SODALITE platform in Section 3, which was accompanied by the implementation status of the prototype evaluated by the demonstrators in Section 5. In between, Section 4 highlighted the initial development status of key components from each technical work package.

The updates of the current document, namely deliverables D6.3 "Intermediate implementation and evaluation of the SODALITE platform and use cases" and D6.4 "Final implementation and evaluation of the SODALITE platform and use cases" that will be provided at the end of the second and third project year, respectively, will present the second (intermediate) and third (final) iterations of the SODALITE platform. These incremental deliverables will report on more advanced features and functionalities, as well as the evaluation of the improvements provided by the SODALITE platform for the project's Demonstrating Use Cases.

[28] https://restassuredh2020.eu/

References

[1] D6.1 - SODALITE platform and use cases implementation plan. SODALITE Technical Deliverable 2019.

[2] D2.1 Requirements, KPIs, evaluation plan and architecture. SODALITE Technical Deliverable 2019.

[3] D3.1 - First version of ontologies and semantic repository. SODALITE Technical Deliverable 2020.

[4] D4.1 - IaC Management - initial version. SODALITE Technical Deliverable 2020.

[5] D5.1 - Application deployment and dynamic runtime - initial version. SODALITE Technical Deliverable 2020.

[6] D3.3 - Prototype of application and infrastructure performance models - First version. SODALITE Technical Deliverable 2019.

[7] D2.4 - Guidelines for contributors to the SODALITE framework. SODALITE Deliverable 2020.

[8] Petri, Carl Adam. "Kommunikation mit automaten." (1962).

[9] Frajberg, D., Fraternali, P., & Torres, R. N. (2017, September). Convolutional neural network for pixel-wise skyline detection. In International Conference on Artificial Neural Networks (pp. 12-20). Springer, Cham.

[10] Schneider, R., Modellierung des inhomogenen orthotropen Materialverhaltens der kortikalen Femurstruktur auf der Basis klinischer CT- bzw. Dichte-Daten, Thesis, Institute of Mechanics, Structural Analysis, and Dynamics of Aerospace Structures (ISD), University of Stuttgart, 2007.

[11] Schneider, R., Faust, G., Hindenlang, U., Helwig, P.: Inhomogeneous, orthotropic material model for the cortical structure of long bones modelled on the basis of clinical CT or density data. Computer Methods in Applied Mechanics and Engineering. 198, 2167 - 2174 (2009).