Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

D2.1

Requirements, KPIs, evaluation plan and architecture - First version

Editor: POLIMI

July 2019

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

Deliverable data			
Deliverable	Requirements, KPIs, evaluation plan and architecture - First version		
Authors	Luciano Baresi (POLIMI), Elisabetta Di Nitto (POLIMI), Panos Mitzias (CERTH), Dragan Radolović (XLAB), Kalman Meth (IBM), Yosu Gorroñogoitia (ATOS), Román Sosa González (Atos), Javier Carnero (Atos), Kamil Tokmakov (USTUTT), Indika Kumara (JADS), Karthee Sivalingam (Cray), Adrian Tate (Cray), Paul Mundt (ADPT), Piero Fraternali (POLIMI), Daniel Frajberg (POLIMI), Rocio Torres (POLIMI), Dimitris Liparas (HLRS)		
Reviewers	Aleš Černivec (XLAB) Kalman Meth (IBM)		
Dissemination level	Public		
History of changes	Name	Change	Date
	v1	first release ready for internal review	12/07/2019
	v2	final release	31/07/2019

Executive Summary

This is the first technical deliverable produced by the SODALITE project. It aims at setting the stage for all future development activities by providing the following contributions:

- A definition of the main UML use cases and associated requirements for the SODALITE framework.
- A definition of the SODALITE architecture described in terms of its components and of the behavior of the components pipeline when executing the relevant UML use cases.
- A preliminary definition of the KPIs (Key Performance Indicators) that will be used to assess the success of the SODALITE framework. Such definition will be revised and possibly updated in the forthcoming versions of this deliverable.
- A preliminary definition of the evaluation plan that will be actuated through the project. This preliminary definition will be made more concrete and precise in the next steps of the project.

The UML use cases and the architecture presented in this deliverable have been defined with the contribution of all partners in the project and constitute the main guidelines that will steer the work in the technical work packages during the first iteration in the project. Both UML use cases and architecture will be treated as living artifacts and will be evolved and updated on the basis of the new findings that will arise through the project. Any evolution will be reported in the new releases of this document (D2.2 and D2.3).

This document has been written in parallel with D6.1 and is complementary to that one. In particular, D6.1, among the other data, provides a list of the used technologies with the corresponding references, the time plan for the development of the SODALITE platform, as well as the implementation plan of the SODALITE case studies, and describes how the SODALITE case studies plan to cover the UML use cases defined in this document.

Glossary

This section is meant to be used as a reference for the main terms used in this document. Most of the terms are defined elsewhere in the document, but their definition is also reported here to allow the reader to find it quickly. To improve readability, all terms are classified under seven main categories.

Acronyms

AADM	Abstract Application Deployment Model
ADM	Application Deployment Model
AOE	Application Ops Expert
CPU	Central Processing Unit
DSL	Domain Specific Language
GPU	Graphic Processing Unit
HPC	High Performance Computing
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
KB	Semantic Knowledge Base
KPI	Key Performance Indicator
LRE	Lightweight Runtime Environment
MOM	Message oriented middleware
OASIS	Organization for the Advancement of Structured Information Standards
OWL	Web Ontology Language
PBS	Portable Batch System
QE	Quality Expert
RDF	Resource Description Framework
RE	Resource Expert
SD	SODALITE Design-time
SR	SODALITE Runtime
Torque	Terascale Open-source Resource and QUEue Manager
TOSCA	Topology and Orchestration Specification for Cloud Applications

General terms

Adaptation plan	An ordered set of actions that modify the current deployment of a system.
Anti-pattern	A common design solution/decision that generates known negative consequences onto the design.
Blueprint	A plan or set of proposals to carry out some work. An IT blueprint is an artifact created to guide priorities, projects, budgets, staffing and other IT strategy-related initiatives. As for IaC, a blueprint is the scripting code that enables resource provisioning, configuration, and application deployment.
Code smell	Any characteristic in the code that possibly indicates a potential defect/bug.
Design pattern	Recurring solution that carries positive consequences onto the design.
Design smell	Any element in the design that indicates violation of fundamental design principles and negatively affects design quality.
Domain Specific Language	A design language that is specific to a particular domain.
Infrastructure as Code	Code that does not define the application logic but, instead, defines the way a computational infrastructure is to be provisioned and configured and the way an application is to be deployed on top of it.
IaC artifacts	These are the documentation and models associated to Infrastructure as Code, as well as the code itself.
Infrastructure as a Service	A specific service model that corresponds to offering virtualized hardware, that is, virtual machines and similar abstractions.
Lightweight application base image	A container image (e.g., Docker or Singularity image).
Models@runtime	Indicates maintaining the models of a system at runtime to reason on the system.
Over-provisioning	The allocation of more computing resources (e.g., virtual machines and CPUs) than strictly necessary.
Playbook	Ansible recipe (or script) for executing a series of steps.
Use case	A possible case of usage of a certain piece of software. SODALITE distinguishes between UML use cases, those reported in this document, and Demonstrating use cases, that is, the specific application we exploit to demonstrate the

	SODALITE framework. These last ones are also called SODALITE case studies.

SODALITE human actors

Application Ops Expert (AOE)	The actor in charge of operating the application and, as such, of all the aspects that refer to the deployment, execution, optimization and monitoring of the application.
Quality Expert (QE)	The actor in charge of the quality of service both provided by the execution infrastructure and required by the executing application.
Resource Expert (RE)	The actor in charge of dealing with the different resources required to deploy and execute the application.

Resources managed by SODALITE

Application component	An executable the application of interest is partitioned in.
Container Engine	An engine for running lightweight containers. It enables operating-system-level virtualization and the existence of multiple isolated container instances.
Edge/Fog computing	A distributed computing paradigm that brings computation and data storage closer to the location where they are needed, to improve response times and save bandwidth.
Execution platform	Provides the means to execute the different application components; e.g HPC, GPU, Openstack Cloud, etc.
Lightweight Runtime Environment	A “simple” execution environment provided by operating systems or by virtualization technologies.
Message oriented middleware	Software infrastructure that supports sending and receiving messages among distributed elements.
Middleware framework	The underlying glue that helps both storing the different data and artifacts and make the different elements communicate.
Monitoring agent	Software entity that collects usage and performance statistics about system resources.
Resource	Any computing artifact needed to deploy and run an application.
Serverless computing	A cloud-computing execution model in which the user submits the tasks to execute and cloud provider manages the computing infrastructure transparently.

Specific targeted technologies

Docker	An open platform for developing, shipping, and running applications. Docker provides the ability to package and run an application in a loosely isolated environment called a container.
Istio	A Service Mesh on top of a cluster manager such as Kubernetes.
Kompose	Kompose is a conversion tool for Docker Compose to container orchestrators such as Kubernetes.
Kubernetes	An open-source system for automating deployment, scaling, and management of containerized applications.
OpenStack	An open source cloud operating system.
OpenWhisk	A popular and highly scalable serverless computing / cloud functions platform that allows for functional logic to be written and triggered in response to events or directly via a REST API.
Portable Batch System	A job scheduler that is designed to manage the distribution of batch jobs and interactive sessions across the available nodes in the HPC cluster.
Singularity	A container solution like Docker that is created specifically for scientific applications and workflows in a HPC environment.
Skydive	A software tool that produces network monitoring metrics.
Slurm	An open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters.
Terascale Open-source Resource and QUEue Manager (Torque)	A distributed resource manager that provides the functionality of PBS but also extends it to provide scalability, fault tolerance, usability and functionality.

SODALITE elements

Abstract Application Tuple	An Abstract Application tuple comprises an abstract description of the application, its infrastructure, and its non-functional requirements..
Application Deployment Model/Abstract Application Deployment Model	An abstract model defined through the use of SODALITE DSL with concrete definitions for constraints, parameters, functional and nonfunctional requirements and goals, thus defining an instance of the DSL model.
Infrastructure Abstract Pattern	A defined set of infrastructure resource types, interlinked with known relationship types (dependencies, compatibility, etc),

	aimed at supporting the recommendation generating mechanism of the Semantic Reasoner.
Semantic Knowledge Base	All modeling artefacts made available to the SODALITE users.
SODALITE Design-time	All SODALITE components made available to the user to support the design and development of Infrastructure as Code (IaC).
SODALITE DSL	The modeling language offered to the SODALITE users to support design and development of IaC.
SODALITE Runtime	All SODALITE components supporting the execution of applications on top of heterogeneous resources.
Taxonomy of Infrastructure Bugs/Defects and Resolutions	A classification of the common bugs and their resolutions for infrastructure designs and IaC code specifications.

Interchange languages

OWL2	An ontology language for the Semantic Web with formally defined meaning. OWL2 ontologies provide classes, properties, individuals, and data values and are stored as Semantic Web documents. OWL2 ontologies can be used along with information written in RDF, and OWL 2 ontologies themselves are primarily exchanged as RDF documents.
TOSCA	An OASIS standard that defines the interoperable description of services and applications hosted on the cloud and elsewhere, thereby enabling portability and automated management across cloud providers regardless of underlying platform or infrastructure; thus expanding customer choice, improving reliability, and reducing cost and time-to-value.

Table of contents

1. Introduction

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

SODALITE wants to address the gap emerging around the aforementioned software-defined, application-specific infrastructures and 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 reflecting diverse circumstances on heterogeneous, software-defined, high-performance, cloud infrastructures, with a particular focus on performance, quality, manageability, and reliability. According to the Grant Agreement, SODALITE will produce several tangible results:

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

The purpose of this document is to analyse in detail the scope and goals of SODALITE and, based on the feedback by our case studies, to define the use cases and requirements for the project. Additionally, this document provides also a first definition of the SODALITE architecture and details its behavior when fulfilling the defined use cases. Finally, the document includes a first definition of the technical KPIs for the project and of the plan that will be adopted for evaluation.

1.1. Relationships with other WPs

Including the definition of requirements and of the SODALITE architecture, this deliverable is the first cornerstone produced by the project and will steer the work of the technical work packages (WP3-WP5). In fact, the deliverable defines the focus of technical WPs in terms of use cases and the architecture and interfaces offered by the components that will be developed under their responsibility.

This deliverable will also influence the work in WP6 as it defines the KPIs to be assessed during the evaluation as well as a preliminary plan for evaluation. As such, this deliverable and deliverable D6.1 [1] from WP6 have been developed in a coordinated and coherent way.

1.2. Preliminary notes

Note on component specification and technologies: according to the grant agreement, D6.1 should have included "specifications of the components' functionality". Instead, we have decided to include them in this deliverable to improve readability and understandability of the architectural diagrams we present here. Instead, a description of the technologies that will be adopted in the project, is presented in D6.1 together with the associated references.

Note on terminology: we distinguish between the **UML use cases**, which are those reported in this document and the **demonstrating use cases**, also called **SODALITE case studies**, which are the

specific applications we will exploit to experiment with the usage of the SODALITE framework. These last ones are developed within the context of WP6 and are described in D6.1.

Note on UML: In this document we adopt UML as a modeling language for use cases (see Figure 1) and for defining the architecture of the SODALITE framework in terms of component diagrams and sequence diagrams (see Figures from 2 to 20). However, the UML notation adopted in this document is not always fully compatible with the UML standard [2]. This is due to the fact that the tool we have adopted, PlantUML[1], does not support some features of the standard. The most significant difference is the absence in the tool of the dock icon representing the concept of required interface. We have replaced this concept with a simple usage relationship.

1.3. Structure of the document

Consistently with its objectives, this document is structured as follows:

- Section 2 focuses on requirement analysis. In particular, it defines the scope and goals of the project, taking the Grant Agreement as a reference. It also identifies the main actors that are relevant in SODALITE and the main use cases that we aim at fulfilling.
- Section 3 focuses on the architecture of the SODALITE framework. It defines the components that will be developed as part of each work package and describes in detail how these components interact with each other in the fulfillment of the use cases.
- Section 4 briefly defines the technical KPIs we will consider in the project.
- Finally, Section 5 provides a preliminary draft of the evaluation plan which essentially reports the approach that we will adopt in the evaluation.

We plan to further detail the content of Sections 4 and 5 in the next versions (D2.2 and D2.3) of this deliverable.

[1] http://plantuml.com/

2. Requirement Elicitation and Analysis

The first phase of any development process is the elicitation of requirements and the subsequent analysis. This activity is even more important in SODALITE given the nature and complexity of the solution the consortium wants to conceive. To elicit requirements we have adopted a scenario-based approach [3] that has lead us to the definition of the following elements:

- *UML Use cases*: these identify a) the boundaries of the SODALITE system in terms of the UML actors the system interacts with, and b) the functionalities the SODALITE system offers to its users (the human actors in UML terms). Note: in the following of this document we will use the generic term use case to mean a UML use case.
- *Requirements*: these are prescriptive assertions that describe what the SODALITE system should offer to its users [4].
- *Domain assumptions*: these are descriptive assertions that should be true in the domain of adoption of the SODALITE system to ensure that this last one can work properly [4].

Both technology partners and domain experts have been involved. In a first iteration, each partner -no matter the specific expertise - has contributed his/her needs and wishes with respect to the SODALITE system. All the requirements identified during this first iteration have been filtered and classified and have allowed us to identify, in the second iteration, the SODALITE actors and the UML use cases and to describe these in detail. In the third iteration, requirements have been assigned to the UML use cases they were most related to. Additionally, a few domain assumptions have been identified.

The following of this section is organized as follows: To help the reader in understanding the vision behind SODALITE, Section 2.1 briefly presents its context and the main goals. Subsequently, Section 2.2 introduces the UML use cases, details each of them and lists the associated requirements and domain assumptions (if any). Finally, Section 2.3 provides a summary and a conclusion for this part.

2.1. SODALITE Context and Goals

The SODALITE vision is to *support Digital Transformation of European Industry through (1) increasing design and runtime effectiveness of software-defined infrastructures, to ensure high-performance execution over dynamic heterogeneous execution environments; (2) increasing simplicity of modelling applications and infrastructures, to improve manageability, collaboration, and time to market* (quote from the grant agreement [5]).

Within this vision, SODALITE will *provide application developers and infrastructure operators with tools that (a) abstract their application and infrastructure requirements to (b) enable simpler and faster development, deployment, operation, and execution of heterogeneous applications reflecting diverse circumstances over (c) heterogeneous, software-defined, high-performance, cloud infrastructures, with a particular focus on performance, quality, manageability, and reliability* (quote from the grant agreement).

In particular, SODALITE is focusing on supporting the entire life cycle of the so-called Infrastructure as Code (IaC). IaC means limiting the need to manually provision resources, configuring them and deploying an application by offering to DevOps teams the possibility to code such tasks into proper scripts that are then executed by proper orchestrators, thus introducing a significant automation in the application life cycle.

If we can define IaC, it means that we can deal with such code as we do with traditional code. Therefore, activities concerning the design of IaC, its verification, its optimization, its dynamic evolution (i.e., the possibility of modifying, on the fly, the IaC associated to a software system so that its deployment and configuration can change) can be envisaged and have the potential to significantly improve the state of practice which is still relatively immature in this area. The purpose of SODALITE is exactly to contribute to this goal.

2.2. Actors and use cases

This section proposes a first identification of the main UML use cases that SODALITE will implement, along with the actors (roles) that can trigger these use cases or contribute to them by providing application elements, execution capabilities, and storage and communication infrastructures.

The actors (see Figure 1) can easily be grouped into two families: the **users**, experts in charge of interacting with and exploiting SODALITE, and the **resources needed by the system** to carry out the different activities. As for users, we have:

- **Application Ops Experts (AOE).** They are in charge of operating the application and, as such, are in charge of all the aspects that refer to the deployment, execution, optimization and monitoring of the application. They are supposed to know the applications to execute and the requirements on both the deployment/execution environment and the quality of services they are interested in.
- **Resource Experts (RE).** They are in charge of dealing with the different resources required to deploy and execute the application. These persons are in charge of application component technologies, of cloud, HPC, and GPU-based computing infrastructures, or of middleware solutions for both storing data and allowing components to communicate.
- **Quality Experts (QE).** They are responsible for the quality of service both provided by the execution infrastructure and required by the executing application. Being part of the SODALITE ecosystem, they are in charge of offering libraries of patterns for addressing specific performance and quality problems in the SODALITE applications.

As for resources, they merely cover all the possible ones needed to operate applications on SODALITE. These resources can be specialized in:

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

The use cases reflect the main activities human actors can trigger or participate in as part of the life cycle management of IaC. More specifically:

- To make the SODALITE framework usable by AOEs, it must be populated with information concerning the resources that can be exploited at runtime. This requires modeling resources (UC13) and making them available, as part of the SODALITE Domain Specific Language, to AOEs. This activity is performed by Resource Experts which are also in charge of mapping the modeled resources into specific optimization patterns (UC12).
- The Quality Expert defines a bug taxonomy for IaC (UC11) that helps AOEs in predicting bugs (UC5). Moreover, he/she experiments with application components and prototypes to estimate their quality characteristics (UC14).
- AOEs start their activity by defining an application deployment model (UC1). This model includes the main components of an application and any constraint or requirement on their deployment, configuration or execution. At this point they can either rely on the resources the SODALITE system would assign by default, or they could select specific resources (UC2). After this step, they are ready to trigger the automatic generation of IaC code (UC3) and its verification (UC4) as well as bug prediction and correction (UC5) and static optimization

(UC15) aiming at improving application performance. Of course these activities may lead to some reiteration in the mentioned use cases until the point in which, as part of the IaC code generation, AOEs generate the needed runtime images (UC16). Then AOEs can trigger the execution of provisioning, configuration and deployment (UC6), start the application (UC7) and start monitoring the execution (UC8) with the purpose of checking that everything is working well and, in case of problems, of identifying possible refactoring and deployment improvement options (UC9). As a result of this identification, they can go back to the modeling and IaC generation/verification/optimization phases and, at this point, trigger a partial redeployment of the system (UC10).

While Figure 1 summarizes the scope of the project and allows the reader to get a high-level idea at a first glance, the next subsections describe each use case in detail. In the next, each use case description uses SD and SR to refer to SODALITE components foreseen at Design-time, and Run-time, respectively.

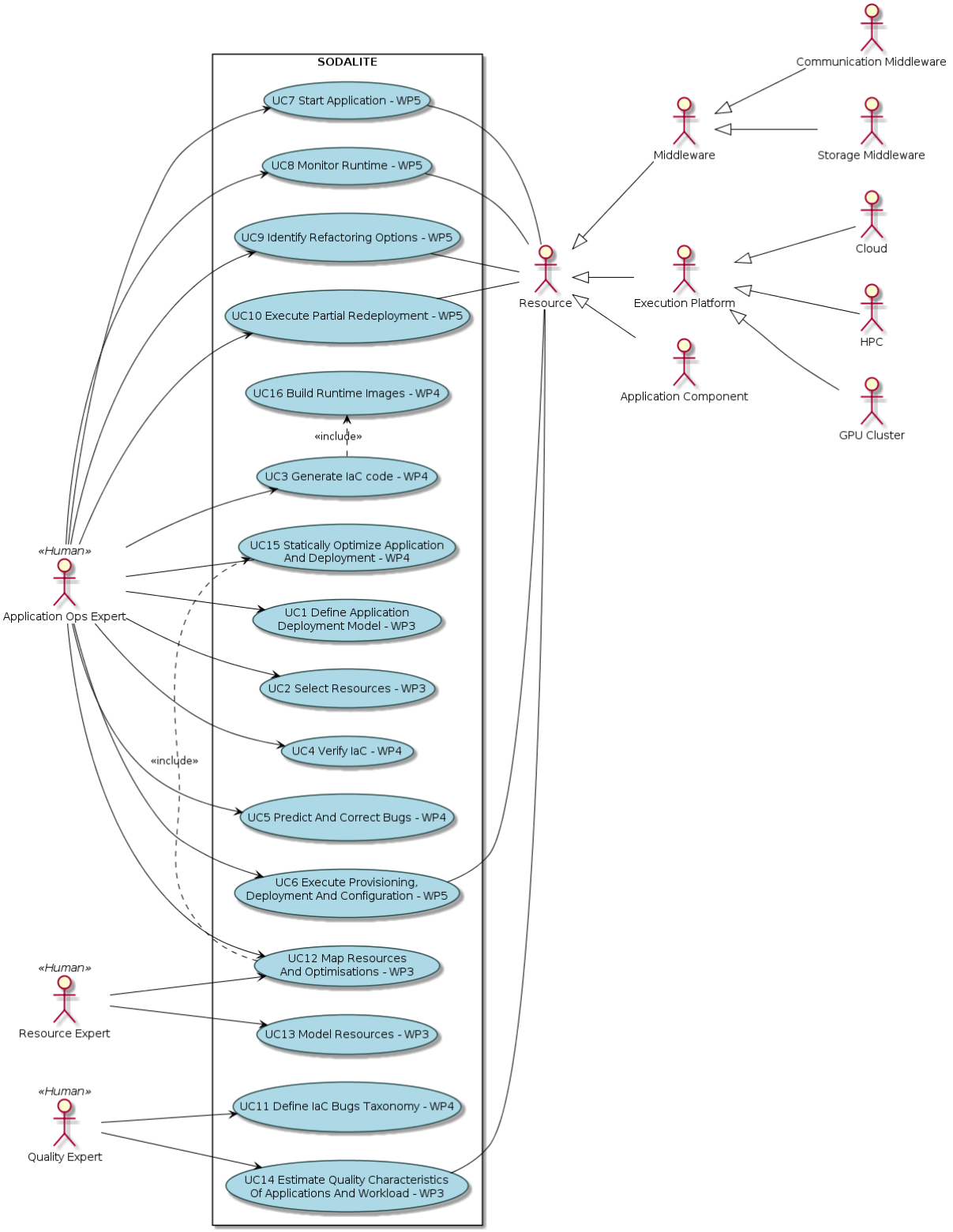

Figure 1: Use cases and actors.

2.2.1. UC1: Define Application Deployment Model (WP3)

Actors:	• Application Ops Expert (AOE)
Entry condition:	Triggered by Application Ops Expert

Flow of events:	The UC sequence is described by the following steps: ● The Application Ops Expert (AOE) initiates the SODALITE Design-time (SD) in order to define the application deployment model. ● The AOE uses a DSL (Domain Specific Language) editor or a graphical interface (e.g., UML diagrams) within the SD to describe application topology/constraints: components/services, inter-component bounds, etc. ● The AOE uses a DSL editor or a graphical interface within the SD to express optimization requirements/constraints: at application, component, and environment level (network, storage). ● In case of graphical input, the SD translates the input into DSL syntax. ● The SD validates the DSL input for syntax validity. ● The SD consults the Semantic Reasoner for the qualitative integrity of the input. The Semantic Reasoner should check for known inconsistencies (e.g., incompatibilities, anti-patterns, etc.) ● All resulted issues/errors are presented to the AOE in design time, so that she can perform refinements. ● The SD consults the Semantic Reasoner for recommendations sourcing from known patterns, requirements, etc, and presents those to the AOE with confidence scores. ● The AOE browses/accepts/rejects recommendations and injects the desired ones into the model. ● After looping through the validation and recommendations steps, the AOE submits the required, valid application deployment model to the SD. ● The SD uploads the model to the semantic knowledge base. ● The SD should then present specific resources/services/products to the AOE (i.e., the process continues to the UC2)
Exit condition:	Application Deployment Model (ADM) created and published on Semantic Knowledge Base (KB)
Exceptions:	Lack of confident system-generated suggestions (can be tackled with a confidence level threshold)

Requirements associated with use case UC1

Id.	Title	Description

UC1.R1	The SODALITE Design-time environment requires an API to the application/Infrastructure abstract pattern repository	Application/Infrastructure abstract patterns can be stored/retrieved to/from this repository through this API from clients such as the Application/Infrastructure Developer Editor (IDE)	
Rationale		**Scope**	**Use Case**
The specification of application/infrastructure abstract models in the IDE is based on the re-utilization of these patterns		Application Components Library	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R2	DSL: specification of application patterns and models	Definition/Implementation of a DSL (both metamodel/semantic) for describing application delivery concerns, including deployment units, dependencies, interactions, configuration, non-functional requirements, etc. This DSL should abstract concrete deployment approaches such as Ansible, Chef, Puppet	
Rationale		**Scope**	**Use Case**
Both application abstract patterns and models (use case specific) are conforming instances of this DSL (metamodel, ontology)		Abstracted Application Tuple	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R3	Authoring of application abstract models (part of abstract tuple)	Developers are assisted by the SD editor to create abstract models describing their application delivery (deployment) requirements according to the deployment (partitioning) strategy for their applications. These abstract models are conforming to the application DSL.	
Rationale		**Scope**	**Use Case**

The specification of application abstract models should be assisted by the editor, which helps developers to reuse modeling patterns (from the repository) as well as on the use of the DSL modeling constructions.		Application Developer Editor	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R4	Integration of Application Developer Editor with SODALITE SD	The specification of the application abstract models takes place within the same IDE that the developer uses for designing and implementing her application.	
Rationale		**Scope**	**Use Case**
The specification of the application abstract model gets simplified from the application information collected from the IDE automatically or by user input that can be obtained from her IDE.		Application Developer Editor	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R5	IntelliJ IDEA IDE extension	Application developer editor extensions as plugin in IntelliJ IDEA IDE	
Rationale		**Scope**	**Use Case**
The Vehicle IoT use case depends heavily on the Android ecosystem, including Android Studio (which no longer supports Eclipse, but is itself derived from IntelliJ) and related IDEs (IntelliJ IDEA, GoLand). An individual plugin would allow for use across all related IDEs, and would further allow frontend and backend modeling to be logically separated.		Application Developer Editor	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R6	Description of application and standard build and run options	The details of the application that will enable its compilation into IaC artifacts suitable for the target architecture must be defined. Any input files needed for test run should be available. Methods to test validity of a successful run should be specified to verify correctness.	
Rationale		**Scope**	**Use Case**

The optimization of the application can occur only if these pieces of information are available		Application Optimiser	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R7	Support for microservice-oriented architecture	SODALITE must support deployment of microservice-oriented architecture developed artifacts	
Rationale		**Scope**	**Use Case**
SODALITE should promote service oriented application design as most suitable for distributed platforms and applications		Runtime	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R8	Abstractions and Mechanisms for Enforcing Performance, Security, and Privacy	The application designs should have abstractions and mechanisms that are required to enforce the optimization decisions (related to performance, privacy, and security).	
Rationale		**Scope**	**Use Case**
We may need specific abstractions and mechanisms in the application to implement some optimization decisions. For example, the traffic (data or message) flows need to be intercepted, rate-controlled (queue/scheduling), and applied privacy enforcement operators.		Deployment Improvement	Define Application Deployment Model (WP3)
Id.	**Title**	**Description**	
UC1.R9	Augment Application Models, IaC Models, and Infrastructure Models for Predicting Control Objectives	The design models of a cloud native application should include the information necessary to support estimating quality attributes to be optimized.	
Rationale		**Scope**	**Use Case**
The design models of the application should have the information necessary to enable quantifying the quality attributes (Performance, Privacy, and Security) of the application.		Application Developer Editor	Define Application Deployment Model (WP3)

Id.	Title	Description	
UC1.R10	Modeling language allowing modeling of all the necessary information to enable the generation of deployable IaC	The deployment preparation tool assumes the SODALITE modeling language will provide all the necessary elements to model heterogeneous infrastructures and applications. The modeling language should provide means to specify all that is needed in order to generate deployable artifacts.	
Rationale		**Scope**	**Use Case**
There must exist suitable inputs to the deployment preparation tool allowing the generation of deployable artifacts.		Infrastructure Operator Editor	Define Application Deployment Model (WP3)

2.2.2. UC2: Select Resources (WP3)

Actors:	• Application Ops Expert (AOE)
Entry condition:	UC1 terminates and the corresponding application deployment model is stored in the KB
Flow of events:	• The SODALITE IDE (SD) retrieves compatible target resources. • [OPTIONAL] The AOE applies filters to available target resources. • The AOE interacts with the SD to receive suggestions about the resource to use. • The SD provides suggestions based on the information provided by the AOE in the application deployment model. • The AOE selects the resources to be used and maps them to the application components. • The SD validates the selection or provides suggestions for changes. In this case, the process goes back to the previous step.
Exit condition:	The ADM is complete as it maps all application components to concrete suitable resources. It is stored back in the KB.
Exceptions:	In case no suitable resource is available, SD returns an error message to AOE

Requirements associated with use case UC2

Id.	Title	Description

UC2.R1	DSL: specification of optimization patterns and models	Definition/Implementation of a DSL (both metamodel/semantic) for describing optimization patterns. This DSL should extend existing standards for abstracting optimization strategies	
Rationale		**Scope**	**Use Case**
The optimization patterns adopted within the abstract application tuple connects application abstract models with infrastructure abstract models and enables the optimal selection of infrastructure capacities		Abstracted Application Tuple	Select Resource (WP3)
Id.	**Title**	**Description**	
UC2.R2	Concretization of abstract models into deployment/configuration plans	The abstract tuple (application, infrastructure, optimization) should be concretized into a deployment/configuration plan, compliant to the standards requested by WP4 (e.g., Ansible, etc)	
Rationale		**Scope**	**Use Case**
The abstract tuple is not a deployable descriptor and needs to be concretized for the target delivery multi-platform, by resolving optimization patterns and the non-functional requirements.		Application Builder	Select Resource (WP3)
Id.	**Title**	**Description**	
UC2.R3	OpenWhisk modeling for serverless computing actions	Modeling should consider not only physical infrastructure, but also serverless computing.	
Rationale		**Scope**	**Use Case**
OpenWhisk is a popular and highly scalable serverless computing / cloud functions platform that allows for functional logic to be written and triggered in response to events or directly via a REST API. These functions play a key role in modern mobile application deployment, and must be managed and utilized alongside other types of conventional infrastructure.		Application Components Library	Select Resource (WP3)
Id.	**Title**	**Description**	

UC2.R4	SLURM/Torque modelling	Modelling must support SLURM/Torque for HPC.	
Rationale		**Scope**	**Use Case**
Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained.		Application Components Library	Select Resource (WP3)
Id.	**Title**	**Description**	
UC2.R5	OpenStack modelling	Modelling must support besides container based deployments also Bare Metal and VM abstractions such as OpenStack.	
Rationale		**Scope**	**Use Case**
OpenStack is an open source cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.		Application Components Library	Select Resource (WP3)
Id.	**Title**	**Description**	
UC2.R6	Use context-aware search and discovery, matchmaking and reuse of cloud applications and infrastructures	The existing application designs (or components) and infrastructure should be able to be dynamically discovered and used when optimizing the application.	
Rationale		**Scope**	**Use Case**
The SD needs to use the rule-based semantic reasoning techniques that are developed by CERTH for context-aware search and discovery, matchmaking and reuse of cloud applications and Infrastructures. The deployment improvement module should be able to query and update the semantic repository.		Deployment Improvement	Select Resource (WP3)

2.2.3. UC3: Generate IaC code (WP4)

Actors:	• Application Ops Expert
Entry condition:	Abstract, optimal, error free ADM is created and source code provided optionally

Flow of events:	• SD parses the ADM. • SD uses the DSL to match optimal IaC node descriptors, relationships, requirement capability based on a repository. • SD builds the IaC blueprint based on matching IaC node description. • SD returns to the AOE the IaC Blueprint.
Exit condition:	IaC blueprints are validated and error free or a list of errors/recommendations is supplied to the user
Exceptions:	If there are errors in the process of creating IaC Blueprints, SD stops the generation of IaC Blueprints, and returns the list of errors found in ADM to AOE

Requirements associated with use case UC3

Id.	**Title**	**Description**	
UC3.R1	SODALITE Runtime (SR) should support Ansible playbooks and TOSCA node definitions for application deployment in public cloud	Implement setup/teardown deployable playbooks for public clouds (e.g., Amazon, Google Cloud)	
Rationale		**Scope**	**Use Case**
SODALITE must address various architectures		Application Components Library	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R2	SR should support Ansible playbooks and TOSCA node definitions for application deployment in HPC environment	Implement setup/teardown deployable orchestrator playbooks for HPC (Torque)	
Rationale		**Scope**	**Use Case**
SODALITE must address various architectures		Application Components Library	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R3	SR should support Ansible playbooks and TOSCA node definitions for application deployment on edge	Implement setup/teardown deployable orchestrator playbooks for REST enabled edge	

Rationale		Scope	Use Case
SODALITE must address various architectures		Application Components Library	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R4	SR should support Ansible playbooks and TOSCA node definitions for application deployment in fog	Implement setup/teardown deployable orchestrator playbooks for fog (network equipment)	
Rationale		**Scope**	**Use Case**
SODALITE must address various architectures		Application Components Library	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R5	Support for SODALITE DSL	The deployment preparation tool must be able to understand and process as input infrastructure specifications defined according to the SODALITE DSL.	
Rationale		**Scope**	**Use Case**
SODALITE will provide a unified language for infrastructure and application specification.		Application Builder	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R6	Generation of correct, complete and deployable IaC artifacts	The deployment preparation tool must be able to generate IaC artifacts that are syntactically correct and runnable. Moreover the deployment preparation tool must be able to generate corresponding IaC artifacts for all the necessary elements and constructs of the SODALITE modeling language.	
Rationale		**Scope**	**Use Case**
SODALITE must produce deployable IaC artifacts.		Application Builder	Generate IaC code (WP4)
Id.	**Title**	**Description**	

UC3.R7	Generation of IaC which exploits heterogeneous architectures	The deployment preparation tool must be able to generate IaC artifacts targeting different architectures (CPUs, HPC, GPUs).	
Rationale		**Scope**	**Use Case**
SODALITE must address heterogeneous architectures.		Application Builder	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R8	Reporting of errors in input models which prevent IaC generation	The deployment preparation tool must report meaningful messages regarding errors encountered during the processing that should help users in fixing issues.	
Rationale		**Scope**	**Use Case**
Error messages must be informative.		Application Builder	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R9	Generation of IaC enabling configuration of runtime components (monitoring, optimization and refactoring) as well as of runtime management policies (refactoring policies, security policies, etc.).	The deployment preparation tool must be able to generate IaC artifacts properly instrumented with configurations and policies needed for runtime application management (monitoring configurations, deployment refactoring policies, security policies, etc.)	
Rationale		**Scope**	**Use Case**
SODALITE must support infrastructures subject to various runtime policies.		Application Builder	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R10	Generation of IaC which exploits serverless computing artifacts (cloud functions)	The deployment preparation tool should be able to generate artifacts targeting pre-defined Cloud Functions in a serverless computing environment.	
Rationale		**Scope**	**Use Case**

SODALITE should be able to handle infrastructure heterogeneity spanning across hardware/software resources, including pre-defined cloud functions available for the application to make use of.		Application Builder	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.R11	Orchestrator input	The Orchestrator takes as input a Concrete Deployment Plan, specified in TOSCA, defining the artifacts that compose an application, their relationships and infrastructure dependencies of each artifact.	
Rationale		**Scope**	**Use Case**
The Concrete Deployment Plan should be a defined instance of a DSL specified by the Application Ops expert, so it benefits from a known and established model.		Runtime	Generate IaC code (WP4)

Assumptions associated with use case UC3

Id.	**Title**	**Description**	
UC3.A1	The modeling approach will be complete enough to offer the possibility of modeling the deployment of complex systems on heterogeneous architectures	The deployment preparation tool assumes that its starting point, i.e., an application deployment model, will contain all pieces of information needed to generate an effective and complete IaC.	
Rationale		**Scope**	**Use Case**
If this assumption does not hold, it will not be possible to generate an executable IaC.		Runtime	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.A2	The modeling approach will be complete enough to offer the possibility to define runtime management policies	The deployment preparation tool assumes that the application deployment model will contain all that is needed to generate IaC artifacts instrumented and configured with runtime management policies.	
Rationale		**Scope**	**Use Case**

If this assumption does not hold, it will not be possible to generate an executable IaC with runtime management policies.		Runtime	Generate IaC code (WP4)
Id.	**Title**	**Description**	
UC3.A3	The modeling approach will be complete enough to offer the possibility to define the configuration of runtime tools such as the monitoring infrastructure	The deployment preparation tool assumes that, as part of the application deployment model, it will be possible to define information that will enable the proper configuration and subsequent execution of the various runtime tools that will be developed in the scope of SODALITE.	
Rationale		**Scope**	**Use Case**
If this assumption does not hold, it will be more difficult to exploit supporting runtime tools during the execution of a complex application.		Runtime	Generate IaC code (WP4)

2.2.4 UC4: Verify IaC (WP4)

Actors:	• Application Ops Expert
Entry condition:	AOE submits the IaC blueprints and the IaC code scripts generated by UC3.
Flow of events:	• SD receives the IaC blueprints and the IaC scripts. • SD parses blueprints and IaC scripts, and builds the formal models required for validating them. • SD checks if the constraints on the nodes, the relationships between nodes, and their properties in the application topology described by the blueprints are satisfied. • SD checks if the constraints on the provisioning workflow described in the IaC scripts are satisfied. • SD provides/displays the validation results to AOE.
Exit condition:	The verification of the blueprints and IaC scripts are completed. The validation results are ready.
Exceptions:	If there are parse errors, SD stops the validation process, and returns the errors to AOE.

Requirements associated with use case UC4

Id.	Title	Description

UC4.R1	Predictive and Corrective (Defect) Analysis of IaC Scripts	The defects in IaC should be predicted, and the resolutions for the identified defects should be recommended.	
Rationale		**Scope**	**Use Case**
Before executing IaC scripts, we need to identify and fix their defects that can cause performance, security, and privacy vulnerabilities.		Deployment Improvement	Verify IaC (WP4)
Id.	**Title**	**Description**	
UC4.R2	Prepare An Infrastructure Code Quality Framework	Infrastructure Code Quality Framework will include the software quality metrics that can be used to assess the quality of IaC. The relevant metrics should be able to be measured and used to predict the defects in IaC.	
Rationale		**Scope**	**Use Case**
The quality factors such as low-coupling, high cohesion, Chidamber and Kemerer metrics can indicate the defects (security vulnerabilities, privacy vulnerabilities, and performance vulnerabilities). We can build the required vulnerability predictors using machine learning techniques.		Deployment Improvement	Verify IaC (WP4)

2.2.5. UC5: Predict and Correct Bugs (WP4)

Actors:	• Application Ops Expert
Entry condition:	AOE submits the IaC artefacts verified by UC 4
Flow of events:	• SD receives the verified IaC artefacts including the IaC blueprints and the IaC scripts. • SD parses the IaC artifacts, and builds the predictive models required for predicting the defects/bugs in them. The defects are anti-patterns, design smells, and code smells for security, privacy and performance. • SD finds defects in the application topology described in the blueprints. • SD finds the defects in the provisioning workflows described in the IaC scripts as well as those in the script source codes. • SD provides/displays the defects and the potential corrections for each identified defect to AOE, which selects the corrections. • SD applies selected corrections.

	• SD can train a machine learning model required for predicting defect-proneness of IaC artifacts based on IaC quality metrics. • SD can predict the defect-proneness index for the IaC artifacts of the application for the machine learning model.
Exit condition:	SD cannot find further defects. AOE decides to stop the defect finding process.
Exceptions:	If there are parse errors, SD stops the defect prediction process, and returns the errors to AOE.

Requirements associated with use case UC5

Id.	**Title**	**Description**	
UC5.R1	Predict and Correct Performance Defects in Designs	Given an application design and the corresponding infrastructure design, the performance defects should be predicted and corrected.	
Rationale		**Scope**	**Use Case**
We need to ensure that the cloud native application to be deployed does not have known performance defects.		Deployment Improvement	Predict and Correct Bugs (WP4)
Id.	**Title**	**Description**	
UC5.R2	Predict and Correct Privacy Defects in Designs	Given an application design and the corresponding infrastructure design, the privacy defects should be predicted and corrected.	
Rationale		**Scope**	**Use Case**
We need to ensure that the cloud native application to be deployed does not have known privacy defects.		Deployment Improvement	Predict and Correct Bugs (WP4)
Id.	**Title**	**Description**	
UC5.R3	Predict and Correct Security Defects in Designs	Given an application design and the corresponding infrastructure design, the security defects should be predicted and corrected.	
Rationale		**Scope**	**Use Case**

We need to ensure that the cloud native application to be deployed does not have known security defects.	Deployment Improvement	Predict and Correct Bugs (WP4)

2.2.6. UC6: Execute Provisioning, Deployment and Configuration (WP5)

Actors:	• Application Ops Expert (AOE) • Resources
Entry condition:	• HPC and Cloud infrastructures must support Lightweight Runtime Environments (LRE), e.g., for Cloud, Kubernetes must be installed, and for HPC container engine, Singularity must be installed on the compute nodes. If VMs are instantiated, then firstly LRE must be installed (alternatively, images containing LRE are booted) • Corrected and verified (UC4, UC5) TOSCA blueprints and deployment playbooks are generated (UC3), and AOE starts the deployment

Flow of events:	• AOE selects the execute provisioning, deployment and configuration option.
	• SODALITE Runtime (SR) receives IaC (blueprints and playbooks).
	if Execution Platform is Cloud:
	• SR provisions Resources according to the IaC.
	• SR deploys application components onto the Cloud infrastructure within LRE (e.g., application container image is deployed from SODALITE image repository).
	• SR configures application (e.g., creates application component endpoints, attaches volumes).
	if Execution Platform is HPC or GPU cluster:
	• SR uploads image into the workspace of the user (e.g., home directory) on the front-end (head) node.
	• SR uploads a job script to be run (e.g., Torque job script with PBS directives).

Exit condition:	Initial application deployment is completed and successful
Exceptions:	• No resources are available at the time of deployment execution. • Resources needed for particular application component are not available. • Deployment is unsuccessful, e.g., corrupted container image, bad container port mapping (port is in use issue), other container related issues (e.g., installed container engine version does not support requested feature, failures to attach volume). • Infrastructure failures, e.g., disk, network equipment failures, system shutdown, human factors.

Requirements associated with use case UC6

Id.	**Title**	**Description**	
UC6.R1	SODALITE Runtime supporting various architectures	Implement SODALITE Runtime to support various architectures.	
Rationale		**Scope**	**Use Case**
SODALITE Runtime needs to address various architectures (public and private clouds, HPC, Edge/Fog nodes) and enable them for resource provisioning in WP5.		Application Components Library	Execute Provisioning, Deployment and Configuration (WP5)
Id.	**Title**	**Description**	
UC6.R2	Support for extension plugins	SR must be extendible to various target architectures. A plugin solution can be envisaged.	
Rationale		**Scope**	**Use Case**
SODALITE must support the development of plug-ins that will use the target system management interface (ranging from OpenStack, Kubernetes, to HPCs Torque).		Application Components Library	Execute Provisioning, Deployment and Configuration (WP5)
Id.	**Title**	**Description**	
UC6.R3	SR should support Ansible playbooks and TOSCA node definitions for application deployment in private cloud.	Implement setup/teardown deployable orchestrator playbooks for private cloud (OpenStack, Kubernetes).	
Rationale		**Scope**	**Use Case**
SODALITE must address various architectures		Application Components Library	Execute Provisioning, Deployment and Configuration (WP5)
Id.	**Title**	**Description**	
UC6.R4	SR plugin supporting Docker Compose	SR should Implement a plugin for multi-container deployments through Docker Compose.	
Rationale		**Scope**	**Use Case**

SODALITE must be able to support a range of plugins and deployment patterns, including those already in use by the use cases.		Application Components Library	Execute Provisioning, Deployment and Configuration (WP5)
Id.	**Title**	**Description**	
UC6.R5	Heterogeneous infrastructure	SR must support, at the same time, different kinds of computing infrastructures. The actual list of infrastructures depends on the case studies requirements, but at least: HPC, Openstack, Kubernetes.	
Rationale		**Scope**	**Use Case**
The SODALITE computing infrastructure is heterogeneous.		Runtime	Execute Provisioning, Deployment and Configuration (WP5)

2.2.7. UC7: Start Application (WP5)

Actors:	• Application Ops Expert (AOE) • Resource
Entry condition:	• Deployment has occurred successfully (UC6). • Monitoring utilities must be preinstalled on the Resources. • AOE initiates the start (it could be an automatic step after deployment, depending on the type of application).

Flow of events:	if Execution Platform is Cloud:
	• SR runs application (could be running already after the deployment).
	if Execution Platform is HPC or GPU Cluster:
	• SR submits a job script to the workload manager.
	• SR waits for the job to start.
	Initiates collection of monitoring metrics

Exit condition:	Application is started
Exceptions:	Application fails at runtime

Requirements associated with use case UC7

Id.	Title	Description
UC7.R1	Lightweight open source Message oriented middleware (MOM) for intra-service communication	The platform should provide communication mechanisms to be reused by the applications.

Rationale		Scope	Use Case
The SR should provide common communication components to be reused in different configurations.		Runtime	Start Application (WP5)
Id.	**Title**	**Description**	
UC7.R2	Smart application scheduling	Multiple heterogeneous applications (streaming, batch processing, microservices, HPC) share the SODALITE infrastructure. In case of resource contention, the platform must provide a mechanism to schedule the execution of applications according to different criteria (e.g., priority, available resources, QoS requirements)	
Rationale		**Scope**	**Use Case**
Different type of applications must be managed according to their different requirements and priority.		Runtime	Start Application (WP5)

2.2.8. UC8: Monitor Runtime (WP5)

Actors:	• Application Ops Expert (AOE) • Resource
Entry condition:	Monitoring agents are running on various entities (e.g., hosts); (1) collecting pre-defined statistics; (2) waiting for requests to perform specialized (not pre-defined) data collection activities.
Flow of events:	• AOE regularly receives pre-defined statistics from monitoring agents. Types of monitored data: network latency, network throughput, Resource utilization metrics (e.g., CPU, memory). Repeat until satisfied (or give up): • AOE requests to see (not pre-defined) performance statistics of specified entities. • Request is translated into requests to SR to obtain the requested statistics. This may result in some new statistics being tracked for some entities. • SR collects the requested statistics. • Updated statistics are presented to AOE. • AOE requests to stop collecting special performance statistics of specified entities.
Exit condition:	Information is received by the requester

Exceptions:	• Monitoring platform is down. • Requested component is down. If request times out, make decisions based on available info and/or return an error indication.

Requirements associated with use case UC8

Id.	**Title**	**Description**	
UC8.R1	IDE Infrastructure dashboard (monitoring, deployment, reconfiguration)	Online runtime application behaviour monitoring, gathered from the target delivery platform, is required to create and update patterns describing the behavior of the underlying infrastructure. Besides, deployment status information and online reconfiguration must be provided to end-users in the dashboard (IDE).	
Rationale		**Scope**	**Use Case**
Online runtime feedback mechanism must close back the loop so that infrastructure (and optimization) abstract patterns could be updated based on learnt observations of application runtime behavior		Infrastructure Operator Editor	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R2	Collect network metrics	Collect network usage between tracked entities (hosts, VMs, containers, switches, etc)	
Rationale		**Scope**	**Use Case**
SR makes deployment decisions based on application specifications and monitoring information. The network monitoring information needs to be imported and understood in order to make re-deployment decisions.		Runtime	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R3	Collect host metrics (CPU, memory)	Collect CPU and memory usage of tracked entities (hosts, VMs, containers, etc)	
Rationale		**Scope**	**Use Case**

SR makes deployment decisions based on application specifications and monitoring information. The CPU and memory monitoring information needs to be imported and understood in order to make re-deployment decisions.		Runtime	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R4	Monitor Overprovisioning (Performance), Security, and Privacy Metrics	The management module should be able to collect the overprovisioning (performance), security, and privacy metrics	
Rationale		**Scope**	**Use Case**
In order to detect and predict the potential violations of the performance (here, avoiding over-provisioning), security, and privacy objectives, we need to use suitable metrics. The application execution environment should support collecting these metrics.		Deployment Improvement	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R5	Monitoring levels	The Monitoring must collect metrics from different levels: - application - runtime environment - infrastructure	
Rationale		**Scope**	**Use Case**
The metrics from multiple levels allows the monitoring system to offer to its users (both human beings and SODALITE adaptation components) a complete view on the system status.		Runtime	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R6	Monitoring infrastructures	The monitor must gather metrics from different kind of infrastructures, specifically HPC and Cloud systems. The actual list of infrastructures depends on the case studies requirements, but at least: HPC, Openstack, Kubernetes	
Rationale		**Scope**	**Use Case**

The SODALITE computing infrastructure is heterogeneous		Runtime	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R7	End-to-end audit logging	Platform must be able to provide end-to-end logging across the entire data life-cycle (e.g., Istio sidecars + Prometheus across the microservice mesh)	
Rationale		**Scope**	**Use Case**
Due to compliance and audit requirements, end-to-end logging across the entire data life-cycle is necessitated.		Runtime	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R8	Visualization of service deployment and adaptations	Platform should be able to support visualization of the running service deployment through a dashboard view (e.g., Grafana)	
Rationale		**Scope**	**Use Case**
The Vehicle IoT use case will deal with the deployment of various services, each of which will contain its own deployment artifacts that should be able to be visually represented and monitored.		Runtime	Monitor Runtime (WP5)
Id.	**Title**	**Description**	
UC8.R9	Absorb Skydive metrics	SkyDive produces network monitoring metrics. These metrics must be absorbed by the Orchestrator to make redeployment decisions.	
Rationale		**Scope**	**Use Case**
SR makes deployment decisions based on application specifications and monitoring information. The network monitoring information coming from Skydive needs to be imported and understood in order to make re-deployment decisions. SkyDive provides a REST API with JSON output.		Runtime	Monitor Runtime (WP5)

2.2.9 UC9: Identify Refactoring Options (WP5)

Actors:	• Application Ops Expert • Resource
Entry condition:	Via runtime monitoring (UC8), SR detects one or more conditions (e.g., QoS violation, under/over utilization of resources, defects, new refactoring options) for initiating a refactoring/adaptation process
Flow of events:	• SR maintains the runtime instance model and the knowledge about the available set of the refactoring options up-to-date. • SR finds defects in the runtime instance model topology and removes the defects. • SR applies patterns to the topology to improve privacy, security, and resource provisioning. • SR replaces one or more current instances of the applied patterns with some other instances to improve privacy, security, and resource provisioning. • SR decides to refactor the deployment architecture (topology) for the application (globally) or to change the resources used by one or more nodes in the current deployment model (locally). • SR finds a valid (consistent and mutually compatible) set of refactoring options while considering refactoring overhead and time, workload dynamics, and system stability. • SR generates an adaptation plan for the selected set of refactoring options. • SR stores the generated adaptation plan in the Semantic Reasoner.
Exit condition:	Once the adaptation plan is generated.
Exceptions:	If a new deployment cannot be found to resolve the situation that triggered the refactoring, send an alert to AOE.

Requirements associated with use case UC9

Id.	**Title**	**Description**	
UC9.R1	Model Control/Optimization Objectives (Performance, Privacy, and Security)	The control or optimization objectives should be represented in a way that enables making trade-offs among them.	
Rationale		**Scope**	**Use Case**
The deployment optimization needs to consider the three key SODALITE quality attributes (Performance, Privacy, and Security). We may need to make trade-offs among them.		Deployment Improvement	Identify Refactoring Options (WP5)

Id.	Title	Description	
UC9.R2	Model Design (Adaptation) Choices	The domain expert's knowledge about the design (or adaptation) choices should be represented.	
Rationale		**Scope**	**Use Case**
To optimize an application design, we have to apply some design options to alter the design. To make trade-off between design alternatives, we need to represent them (the whole design choices) in a suitable form (e.g., a feature model, and decision tree).		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R3	Find an Optimal Design Solution Considering Control Objective Tradeoffs	An optimal design for the application and its infrastructure should be found considering the desired control/optimization objectives (and their tradeoffs)	
Rationale		**Scope**	**Use Case**
Given an initial application design and the corresponding initial infrastructure design, we need to come with an optimal design solution. We may need to make the trade-off between optimization objectives.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R4	Forecast Workload (Multi-class/tenant)	The application workload should be able to be forecasted.	
Rationale		**Scope**	**Use Case**
When determining performance violations and overprovisioning, and optimizing the running application, the control algorithms should use the forecasted workload.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R5	Forecast Infrastructure Dynamics	The states and dynamics of the infrastructure should be able to be forecasted.	
Rationale		**Scope**	**Use Case**

Similar to the above explanation		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R6	Predict Violations of Control Objectives (Performance, Security, and Privacy)	The potential violations of control objectives (performance, security, and privacy) should be able to be predicted.	
Rationale		**Scope**	**Use Case**
The potential (or detected) violations of control objectives drives the adaptation decisions. The adaptation policies can be reactive and/or proactive.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R7	Generate Application and Infrastructure Adaptation Plans	Based on the differences between the current running system and the desired system (the identified optimal design solution), the adaptation plans should be generated.	
Rationale		**Scope**	**Use Case**
In order to move the current system to the desired system, we need to modify the current system. The required modifications are defined as an adaptation plan (an ordered set of adaptation commands). The runtime models kept in the control module (models@runtime) can be used to generate these plans.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R8	Enact Application and Infrastructure Adaptation Plans	The adaptation plans should be scheduled and enacted gracefully.	
Rationale		**Scope**	**Use Case**
The adaptation plans should be executed in order to propagate the desired changes. Ideally, the performance violations and system instability during the execution of the adaptation plans should be prevented and minimized.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	

UC9.R9	Detect and Correct Defects at Runtime	The defects in the running application (instance model) should be predicted and corrected.	
Rationale		**Scope**	**Use Case**
When the application is running, new defects may be discovered (compared with what found during the design time). The runtime changes (adaptations) may also introduce new defects (e.g., create a security anti-pattern). Thus, we need to predict and correct the defects at runtime too. These will use the relevant techniques developed in Task4.4.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R10	Static Provisioning of Heterogeneous Resources	The platform must be able to statically predict the amount of resources to allocate to an application to 1) avoid violations of the Quality of Service guaranteed to the users (latency, throughput, deadlines) 2) improve efficiency (minimize the resource consumption). The allocation could mix different types of resources (HPC, GPU, VMs) since the SODALITE infrastructure is heterogeneous.	
Rationale		**Scope**	**Use Case**
The SODALITE computing infrastructure is heterogeneous, thus, resources must be efficiently provisioned.		Runtime	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R11	Elastic Provisioning of Heterogeneous Resources	The platform must be able to change dynamically the amount of resources allocated to an application to 1) avoid violations of the Quality of Service guaranteed to the users (latency, throughput, deadlines) 2) improve efficiency (minimize the resource consumption). The allocation could mix different types of resources (HPC, GPU, VMs) since the SODALITE infrastructure is heterogeneous.	
Rationale		**Scope**	**Use Case**

The SODALITE computing infrastructure is heterogeneous, resources must be efficiently managed at runtime.		Runtime	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R12	TOSCA inputs to SR	SR must receive TOSCA-based adaptation plans.	
Rationale		**Scope**	**Use Case**
As SR supports TOSCA-based deployment, the adaptation plans must be in the same format.		Deployment Improvement	Identify Refactoring Options (WP5)
Id.	**Title**	**Description**	
UC9.R13	Dynamic Policy-based restrictions on resource access from the Edge	SR must be able to limit resource access based on dynamic policy changes from the Edge.	
Rationale		**Scope**	**Use Case**
This requirement is very important from the viewpoint of the IoT use case: 1. A backend instance is pushed down to the Edge (in our case, a Vehicle), which may be travelling between different jurisdictions – subjecting it to different compliance requirements based on locality. This issue is further conflated by the case that the Cloud-backend and the Edge instance of the server may not be in the same legal jurisdiction, subjecting the entire set of dataflows to a superset of compliance requirements. 2. The end-user (Driver) of the application, may dynamically change things like their privacy preferences and consent settings at any time during the deployment. Based on these changes, we need to re-evaluate what this means for the infrastructure, and determine whether we need to re-deploy, enable/disable features, update access control policies, etc.		Runtime	Identify Refactoring Options (WP5)

2.2.10 UC10: Execute Partial Redeployment (WP5)

Actors:	• Application Ops Expert (AOE) • Resources

Entry condition:	• The refactoring options have been identified (UC9) and the deployment model has been successfully updated • TOSCA blueprints and Ansible playbooks are generated and Application Ops Expert starts the redeployment

Flow of events:	• SR receives IaC (blueprints and playbooks) referring to the current deployment.
	• SR derives the differences between current and updated deployment that needs to be applied to get the updated (desired) state.
	• SR applies these differences until the current state of deployment will be the updated (desired) state.
	if Resource is Cloud:
	• SR executes possible refactoring options: scale in/out/up/down, migrate to another Resource, deploy new components, remove current components.
	if Resource is HPC/GPU:
	• SR executes possible refactoring options: migrate to another Resource, deploy new components, remove current components.

Exit condition:	Redeployment is completed, application is run, and monitoring is established with a "healthy" status
Exceptions:	• SR fails to derive the differences, although the deployment model was updated • No resources are available at the time of redeployment • Redeployment is unsuccessful, e.g., failures in scaling in/out, horizontally/vertically • Infrastructure failures, e.g., disk, network equipment failures, system shutdown, human factors

Requirements associated with use case UC10

Id.	**Title**	**Description**	
UC10.R1	Create and Maintain Runtime Models	SODALITE SR (the control loop) must maintain a replica (a virtual copy) of the deployed application and the infrastructure using the models@runtime paradigm. The runtime models should also include the models representing control objectives (see UC9.R1)	
Rationale		**Scope**	**Use Case**

In order to plan and simulate the runtime changes (e.g., refactoring) to the actual system, we need a model representing the actual system. This model maintains a live (descriptive and prescriptive) connection with the actual system. This will also enable the consistency checking of the adapted application.		Deployment Improvement	Execute Partial Redeployment (WP5)
Id.	**Title**	**Description**	
UC10.R2	Horizontal Resource Scalability	The platform should provide means to horizontally scale heterogeneous resources (GPU, VMs, Containers). Tools (e.g., Kubernetes) must be extended or modified to allow the desired behavior	
Rationale		**Scope**	**Use Case**
Enables UC9.R11		Runtime	Execute Partial Redeployment (WP5)
Id.	**Title**	**Description**	
UC10.R3	Vertical Resource Scalability	The platform should provide means to vertically scale heterogeneous resources (GPU, VMs, Containers). Tools (e.g., Kubernetes) must be extended or modified to allow the desired behavior.	
Rationale		**Scope**	**Use Case**
Enables UC9.R11. Vertical scalability is an emerging technique that avoids the overhead of booting new virtual assets and allows for a faster and fine-grained resource allocation.		Runtime	Execute Partial Redeployment (WP5)

2.2.11. UC11: Define IaC Bugs Taxonomy (WP4)

Actors:	• Quality Expert (QE)
Entry condition:	The descriptions of defects and resolutions in the literature (such as IaC code smells, design smells, anti-patterns, and design patterns) from a domain analysis
Flow of events:	• QE identifies, defines, and organizes a vocabulary of the key concepts used to describe defect and resolutions in terms of IaC (code and design) smells and anti-patterns/patterns found in the literature.

	• QE uses the vocabulary to classify, relate, and combine all defects and resolutions, and builds a taxonomy. • QE formally codifies the taxonomy using ontologies (logics). • QE formally codifies the rules to apply on the models of IaC artifacts to detect each defect, and to recommend the relevant resolutions, all based on the taxonomy. • SD stores the ontological representation of the taxonomy, and the rules.
Exit condition:	When the taxonomy is sufficiently complete.
Exceptions:	

Requirements associated with use case UC11

Id.	**Title**	**Description**	
UC11.R1	Create a Taxonomy of Infrastructure Bugs and Resolutions	A taxonomy of the bugs/defects and resolutions should be created for infrastructure as codes. This provides a basis for detecting defects in IaC scripts and resolving them (static analysis). The taxonomy needs to be built using a literature review and qualitative analysis of IaC scripts. We need to consider the defects related to the three key quality attributes of SODALITE (performance, security, and privacy).	
Rationale		**Scope**	**Use Case**
We can use a taxonomy to characterize the IaC defects and bugs, and their potential resolutions. With a rule-based approach or an ontological approach, we can use the taxonomy to automatically detect the defects in the IaC scripts. We can also use the taxonomy (knowledge) in machine learning algorithms or evolutionary algorithms for defect prediction in IaC.		Deployment Improvement	Define IaC Bugs Taxonomy (WP4)

2.2.12. UC12: Map Resources and Optimisations (WP3)

Actors:	• Resource Expert (RE)
Entry condition:	Triggered by Resource Expert
Flow of events:	• The Resource Expert (RE) initiates the SODALITE IDE (SD) in order to define resource patterns and optimisations (they could regard application patterns, infrastructure patterns, etc).

	• The RE assembles a pattern using known resource types and relationships (as defined by the Semantic Knowledge Base - KB). This can be done via a DSL editor or a graphical interface. • The RE selects relevant optimisation approaches from a given UI menu in the SD. • The SD retrieves metadata and dependencies from the Execution Platform and available targets from the IaC Model repository. • The SD validates the relevance of the given set of optimisations. • The RE reviews and submits the newly created pattern and optimisation details and they are stored to the KB. • [OPTIONAL] The RE corresponds sets of compatible target resources for her new pattern, using the known resources in the KB that were created as part of UC13 and UC14. • [OPTIONAL] the RE assigns performance metrics to each target resource. • [OPTIONAL] The SD stores the defined target resources to the KB, so that they can be used within UC2
Exit condition:	The RE submits a new application/resource pattern and [OPTIONAL] compatible target resources
Exceptions:	A problem that might occur is the RE trying to assign target resources that are incompatible with each other for a) technical reasons as defined in UC13 b) non-technical reasons (e.g., competing infrastructure vendors, etc.) The SD should prevent the submission of such cases to the KB.

Requirements associated with use case UC12

Id.	**Title**	**Description**	
UC12.R1	Select Optimisations for Application and Infrastructure targets	SODALITE should select the optimisation options that can be applied to the application based on the target resources available. These options are made available for the Application Ops Expert to select from.	
Rationale		**Scope**	**Use Case**
This task takes an HPC view of application and infrastructure modelling with the intention of providing tools to enable performance abstractions - that is software tools and patterns that enable explicit or implicit performance decisions.		Application Optimiser	Map Resources and Optimisation (WP3)

2.2.13. UC13: Model Resources (WP3)

Actors:	• Resource Expert
Entry condition:	Triggered by Resource Expert: A new resource needs to be added to the SODALITE Semantic Knowledge Base (KB)
Flow of events:	• The Resource Expert (RE) initiates the SODALITE IDE (SD) in order to define new resources. • The SD retrieves and presents to the RE the known abstract resource types that exist in the ontology schema. • The RE decides whether she wants to add a new abstract resource type within the schema or a new target resource. • In case of a new resource type, the RE needs to define where it should be put in the ontology (i.e., define its super classes). • [OPTIONAL] The RE defines relationship/dependency types between resource types (classes). • [OPTIONAL] The RE defines relationships/dependencies between target resources (instances). • [OPTIONAL] The RE defines optimization approaches/patterns. • The RE submits to the SD every new definition she makes. • The SD validates the integrity of these changes. • The SD communicates the changes to populate the KB.
Exit condition:	The new resource is now in the ontology and can be used as part of an application model.
Exceptions:	• If SD is unable to extend the DSL it will inform the user with an error message • If SD will not be able to plug in the new model transformation code, it will raise an error and will signal an error if the user will try to use that resource

Requirements associated with use case UC13

Id.	**Title**	**Description**	
UC13.R1	Docker Modelling	Modelling must support container runtime: Docker RunC and ContainerD	
Rationale		**Scope**	**Use Case**
One of the runtime environments will be containers (most popular is Docker), so we need to be able to model this type of infrastructure.		Application Components Library	Model Resources (WP3)
Id.	**Title**	**Description**	

UC13.R2	Kubernetes Modelling	Modelling must support abstractions of Kubernetes: Pods, deployment, network policy, Service, Ingress, DaemonSet, etc.	
Rationale		**Scope**	**Use Case**
Kubernetes is a popular cluster manager that is used in many deployments, so we need to be able to model this type of infrastructure.		Application Components Library	Model Resources (WP3)
Id.	**Title**	**Description**	
UC13.R3	Istio Modelling	Modelling should support Istio entities	
Rationale		**Scope**	**Use Case**
Istio provides a Service Mesh on top of a cluster manager such as Kubernetes.		Application Components Library	Model Resources (WP3)
Id.	**Title**	**Description**	
UC13.R4	Ontology Serialization	The semantic model (i.e., ontology) serialization should be compliant to the OWL 2 ontology language.	
Rationale		**Scope**	**Use Case**
Compliance with OWL 2 offers semantic interoperability with other third-party ontologies, and encourages the adoption of our ontology byother parties.		Application Components Library	Model Resources (WP3)
Id.	**Title**	**Description**	
UC13.R5	TOSCA Compliance	The semantic model (i.e., ontology) should be compliant to the TOSCA standard.	
Rationale		**Scope**	**Use Case**
TOSCA is an OASIS standard that defines the interoperable description of services and applications hosted on the cloud and elsewhere, thereby enabling portability and automated management across cloud providers regardless of underlying platform or infrastructure; thus expanding customer choice, improving reliability, and reducing cost and time-to-value.		Application Components Library	Model Resources (WP3)

Id.	Title	Description	
UC13.R6	Authoring of infrastructure abstract models (part of abstract tuple)	Infrastructure vendors are assisted by the SODALITE SD to create abstract models describing their infrastructures. These abstract models are conforming to the infrastructure DSL	
Rationale		**Scope**	**Use Case**
The specification of infrastructure abstract models should be assisted by the editor, which helps vendors to reuse modeling patterns (from the repository) as well as on the use of the DSL modeling constructions.		Infrastructure Operator Editor	Model Resources (WP3)
Id.	**Title**	**Description**	
UC13.R7	IasS Modelling	Modelling must support abstractions of IaaS	
Rationale		**Scope**	**Use Case**
IaaS is one of the SODALITE target execution platforms		Application Components Library	Model Resources (WP3)
Id.	**Title**	**Description**	
UC13.R8	IaC deployment management Modelling	Modelling must support abstractions of IaC.	
Rationale		**Scope**	**Use Case**
This to enable the IaC generation components to generate effective infrastructural code.		Application Components Library	Model Resources (WP3)
Id.	**Title**	**Description**	
UC13.R9	Description of the available hardware	SODALITE should have details of the available hardware on which the applications will be deployed. Details of the compute, memory, storage, network should be available within IaC artifacts. In case of HPC resources, the available queues for test/optimization run should be listed. If there are more than one single architecture (CPU/GPU, DRAM/HBM), valid mapping of several available options should be listed.	

Rationale	Scope	Use Case
Support for heterogeneous hardware is envisaged	Application Optimiser	Model Resources (WP3)

Assumptions associated with use case UC13

Id.	Title	Description	
UC13.A1	RDF Triplestore	An RDF-based graph database like GraphDB needs to be deployed.	
Rationale		**Scope**	**Use Case**
The triplestore will host the semantic models (ontologies) produced within WP3 and act as the semantic repository. It will enable the population/retrieval of data (e.g. patterns) to/from the ontologies and the application of semantic reasoning.		Application Components Library	Model Resources (WP3)

2.2.14. UC14: Estimate Quality Characteristics of Applications and Workload (WP3)

Actors:	• Quality Expert (QE)
Entry condition:	A new application or resource must be added to KB to then enable optimizations and (initial) resource provisioning.
Flow of events:	• QE creates a simulated environment or a mockup to acquire the information of interest (performance, security, privacy metrics). • QE runs the profiling execution/simulation and stores the data of interest in KB. • QE uses SD to relate data, applications, and resources.
Exit condition:	Sufficient data are available and can be used as part of UC12
Exceptions:	• QE cannot set a proper test environment. • Test execution exceeds foreseen resources and time. • Data are not sufficiently accurate.

Requirements associated with use case UC14

Id.	Title	Description
UC14.R1	Estimate Performance of Designs	Given an application design and the corresponding infrastructure design, the performance of the application should be estimated/calculated.

Rationale		Scope	Use Case
In order to determine an application is optimal with respect to performance, we first need to be able to estimate the performance of the application based on its design models.		Deployment Improvement	Estimate Quality Characteristics of Applications and Workload (WP3)
Id.	**Title**	**Description**	
UC14.R2	Estimate Security Level of Designs	Given an application design and the corresponding infrastructure design, the degree of security of the application should be estimated/calculated.	
Rationale		**Scope**	**Use Case**
In order to determine an application is optimal with respect to security, we first need to be able to estimate the security level of the application based on its design models.		Deployment Improvement	Estimate Quality Characteristics of Applications and Workload (WP3)
Id.	**Title**	**Description**	
UC14.R3	Estimate Privacy Level of Designs	Given an application design and the corresponding infrastructure design, the degree of privacy of the application should be estimated/calculated.	
Rationale		**Scope**	**Use Case**
In order to determine an application is optimal with respect to privacy, we first need to be able to estimate the privacy level of the application based on its design models.		Deployment Improvement	Estimate Quality Characteristics of Applications and Workload (WP3)
Id.	**Title**	**Description**	
UC14.R4	Assess the Impact of a Design Choice	The impact of a design choice on the performance, security, and privacy of the application should be able to be estimated.	
Rationale		**Scope**	**Use Case**
In order to refactor the application and infrastructure designs by applying design choices, we need to be able to estimate the impact of each such choice with respect to the three key quality attributes of SODALITE.		Deployment Improvement	Estimate Quality Characteristics of Applications and Workload (WP3)

2.2.15. UC15: Statically Optimize Application and Deployment (WP4)

Actors:	• Application Ops Expert (AOE)
Entry condition:	Optimisation is enabled
Flow of events:	Based on specified target architecture and selected optimisation options, the application will be optimised and the deployment model will be updated. • SR checks and filters for optimisation support for specified target architecture. If none is available it exits. • SR checks and filters for validity of selected optimisation options. If none exists, it exits. For the selected optimisation options • SR applies optimisation. • SR validates optimisation. • SR updates deployment for any runtime changes. • Optimisation report sent to AOE.
Exit condition:	Successful optimisation and validation
Exceptions:	In case of failure at any stage of the optimisation process, the default or the input will be returned with no changes. The expectation is that the entire workflow will work without the optimiser (Optional)

Requirements associated with use case UC15

Id.	**Title**	**Description**	
UC15.R1	Delivery of optimized application	The optimized application (based on the selected optimisations) as an executable or container along with its runtime environment (JIT compiler) will be passed to the deployment manager for deployment.	
Rationale		**Scope**	**Use Case**
Application executable can be optimized based on the mapping in UC12		Application Optimiser	Statically Optimize Application and Deployment (WP4)
Id.	**Title**	**Description**	
UC15.R2	Optimise Application and Deployment	Platforms should optimise Application and its deployment based on the optimisation selected by the Application Ops Expert	

Rationale	Scope	Use Case
The framework for performance optimizations of three types are developed in this task. The Hardware Abstract Interface allows code to be developed in higher-level abstract interfaces that are both hardware neutral but enabling to some degree the machine performance optimisation. Native optimisations are provided for CPU, memory and I/O options that optimise a task for the given hardware. Runtime parameters are assessed empirically through monitoring and provide the best combination of runtime parameters based on search optimisation.	Application Optimiser	Statically Optimize Application and Deployment (WP4)

Assumptions associated with use case UC15

Id.	Title	Description	
UC15.A1	Optimization options for Memory	The optimization process assumes that the required level of memory optimization and the available memory hardware in the infrastructure are available. Default safe option can be assumed by the optimiser if no value available.	
Rationale		**Scope**	**Use Case**
Native optimizations are provided for CPU, memory and I/O options that optimize a task for the given hardware. Runtime parameters are assessed empirically through monitoring and provide the best combination of runtime parameters based on assessed empirically through monitoring and provide the best combination of runtime parameters based on search optimization.		Application Optimiser	Statically Optimize Application and Deployment (WP4)
Id.	**Title**	**Description**	
UC15.A2	Optimization options for IO	The level of storage optimization should be available. Default safe option can be assumed by the optimizer if no value available.	
Rationale		**Scope**	**Use Case**

Native optimizations are provided for CPU, memory and I/O options that optimize a task for the given hardware. Runtime parameters are assessed empirically through monitoring and provide the best combination of runtime parameters based on assessed empirically through monitoring and provide the best combination of runtime parameters based on search optimization.		Application Optimiser	Statically Optimize Application and Deployment (WP4)
Id.	**Title**	**Description**	
UC15.A3	Optimization options for Network	The level of network optimization should be available. Default safe option can be assumed by the optimizer if no value available.	
Rationale		**Scope**	**Use Case**
Native optimisations are provided for CPU, memory and I/O options that optimise a task for the given hardware. Runtime parameters are assessed empirically through monitoring and provide the best combination of runtime parameters based on assessed empirically through monitoring and provide the best combination of runtime parameters based on search optimisation.		Application Optimiser	Statically Optimize Application and Deployment (WP4)
Id.	**Title**	**Description**	
UC15.A4	Optimisation options for Autotuning	The level of autotuning optimisation should be available. Default safe option can be assumed by the optimiser if no value available.	
Rationale		**Scope**	**Use Case**
Native optimisations are provided for CPU, memory and I/O options that optimise a task for the given hardware. Runtime parameters are assessed empirically through monitoring and provide the best combination of runtime parameters based on assessed empirically through monitoring and provide the best combination of runtime parameters based on search optimisation.		Application Optimiser	Statically Optimize Application and Deployment (WP4)
Id.	**Title**	**Description**	

UC15.A5	Optimisation options for Compute	The level of compute optimisation should be available. Default safe option can be assumed by the optimiser if no value available.	
Rationale		**Scope**	**Use Case**
Native optimisations are provided for CPU, memory and I/O options that optimise a task for the given hardware. Runtime parameters are assessed empirically through monitoring and provide the best combination of runtime parameters based on assessed empirically through monitoring and provide the best combination of runtime parameters based on search optimisation.		Application Optimiser	Statically Optimize Application and Deployment (WP4)

2.2.16. UC16: Build Runtime Images (WP4)

Actors:	• Application Ops. Expert
Entry condition:	Target execution environment is defined, artefact binary executables with dependencies and configurations exist in predefined locations
Flow of events:	• SR reads the definition of Target Environment • SR chooses the best matching template for the process of image building based on supported target environment image container technology (eg. HPC, Cloud etc.) • SR gathers all the artefacts with dependencies and configuration for the execution environment • SR builds the image and stores it.
Exit condition:	The image is stored properly in the SODALITE framework
Exceptions:	Failure in the image building process (e.g., errors in definitions of target environment, binary executable artefact with dependencies and configurations and similar): If there are errors in the process of creating the artefacts runtime image, SD stops the build process, and returns the list of errors found in the process referring to the application deployment model and makes this available to AOE.

Requirements associated with use case UC16

Id.	**Title**	**Description**	
UC16.R6	Lightweight application base images	The platform must provide light base images for the applications to be reused.	
Rationale		**Scope**	**Use Case**

The SR must have minimum footprint on application and resources usage.	Runtime	Build Runtime Images (WP4)

2.3. Summary of use cases

In the previous sections we have described the SODALITE UML use cases. Each of them involves at least a human actor and may include also the interaction with some resources which are to be considered as external actors as well.

The table below provides a summary of all use cases and highlights the main human actors involved in them and whether they are mandatory steps for a proper usage of SODALITE or whether they can be considered optional. These latter ones concern steps that can either be triggered by the actors or can be skipped. They are:

- UC2: Select Resources since default resources can be assigned to an application if no one is selected
- UC5: Predict and Correct Bugs as the AOE may be willing to exclude this automated correction and may want to take care of bugs by himself/herself.
- UC8: Monitor runtime as, while monitoring is highly beneficial, it may introduce an overhead that users may want to exclude. Of course, excluding monitoring implies that UC9 and UC10 (refactoring and redeployment) cannot be performed.
- UC9: Identify Refactoring Options, UC10: Execute Partial Redeployment and UC15: Statically Optimize Application and Deployment. Even if these represent the most advanced features offered by SODALITE, the user can still exploit the platform without exploiting them.

Use Case	Main target users	Notes
UC1 Define Application Deployment Model (WP3)	Application Ops Experts	Mandatory
UC2 Select Resources (WP3)	Application Ops Experts	Optional
UC3 Generate IaC code (WP4)	Application Ops Experts	Mandatory
UC4 Verify IaC (WP4)	Application Ops Experts	Mandatory
UC5 Predict and Correct Bugs (WP4)	Application Ops Experts	Optional
UC6 Execute Provisioning, Deployment and Configuration (WP5)	Application Ops Experts	Mandatory
UC7 Start Application (WP5)	Application Ops Experts	Mandatory
UC8 Monitor Runtime (WP5)	Application Ops Experts	Optional but mandatory for refactoring and redeployment
UC9 Identify Refactoring Options (WP5)	Application Ops Experts	Optional (Mandatory for Vehicle IoT UC)

UC10 Execute Partial Redeployment (WP5)	Application Ops Experts	Optional (Mandatory for Vehicle IoT UC)
UC11 Define IaC Bugs Taxonomy (WP4)	Quality Experts	Mandatory for QE
UC12 Map Resources and Optimisations (WP3)	Resource Experts	Mandatory for RE
UC13 Model Resources (WP3)	Resource Experts	Mandatory for RE
UC14 Estimate Quality Characteristics of Applications and Workload (WP3)	Quality Experts	Mandatory for QE
UC15 Statically Optimize Application and Deployment (WP4)	Application Ops Experts	Optional
UC16 Build Runtime images (WP4)	Application Ops Experts	Mandatory

3. Architecture

The SODALITE platform is divided into three main layers, each covered by a separate work package. These layers are the Modeling layer (WP3), the Infrastructure as Code layer (WP4), and the Runtime layer (WP5). Figure 2 shows these layers together with their relationships defined in terms of offered and used interfaces. The Modeling layer exploits the interfaces offered by the other two layers to offer to the end users (Application Ops Experts, Resource Experts and Quality Experts) the needed information concerning the application deployment configuration and the corresponding runtime. In turn, it offers to the other layers the possibility to access the ontology and the application deployment model through the SemanticReasoningAPI. The Infrastructure as Code Layer offers to the modeling layer the APIs for preparing the deployment, for verifying the IaC and for predicting defects. Finally, the Runtime Layer offers the APIs for controlling the orchestration of an application deployment and for monitoring the status of the system. In turn, this layer relies on the interfaces offered be the underlying technologies with particular reference to the ones shown in the figure.

In the next sections, we describe the main components in each of these layers and how they fit together. For each component, we specify its functional description, inputs, outputs, programming languages/tools used, dependencies, and critical factors (such as risks that can affect the implementation of the component). The sequence diagrams for the previously defined use cases fill in the details of the data flow, while referring to additional sub-components not shown in the high-level diagrams.

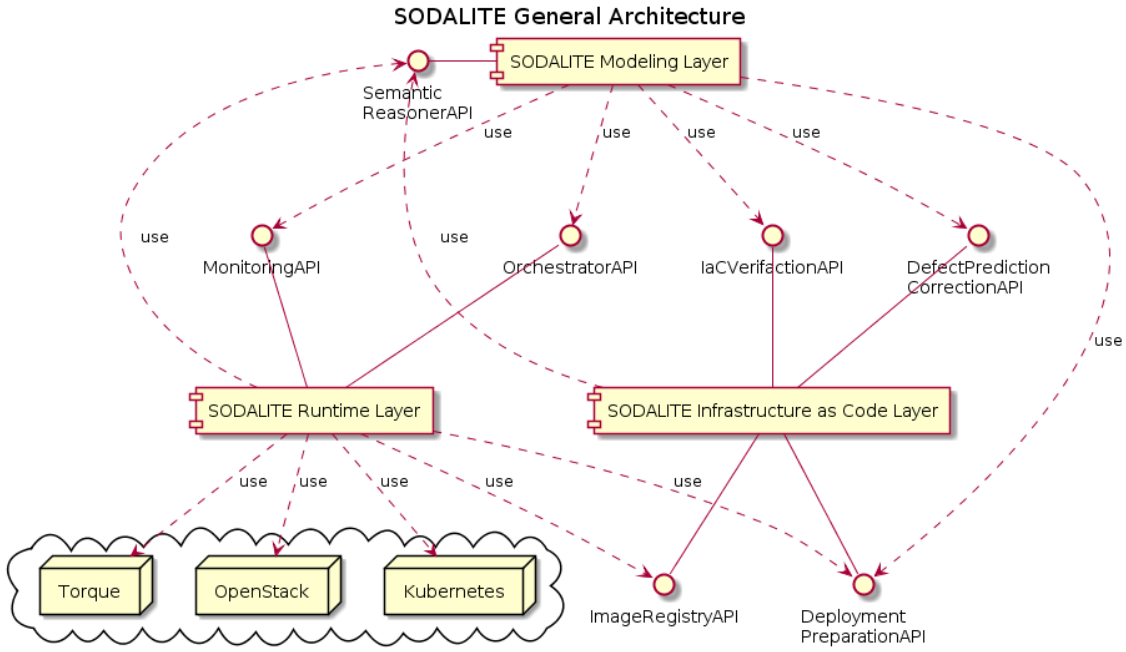

Figure 2: SODALITE Overall Architecture.

3.1. WP3 Modelling layer

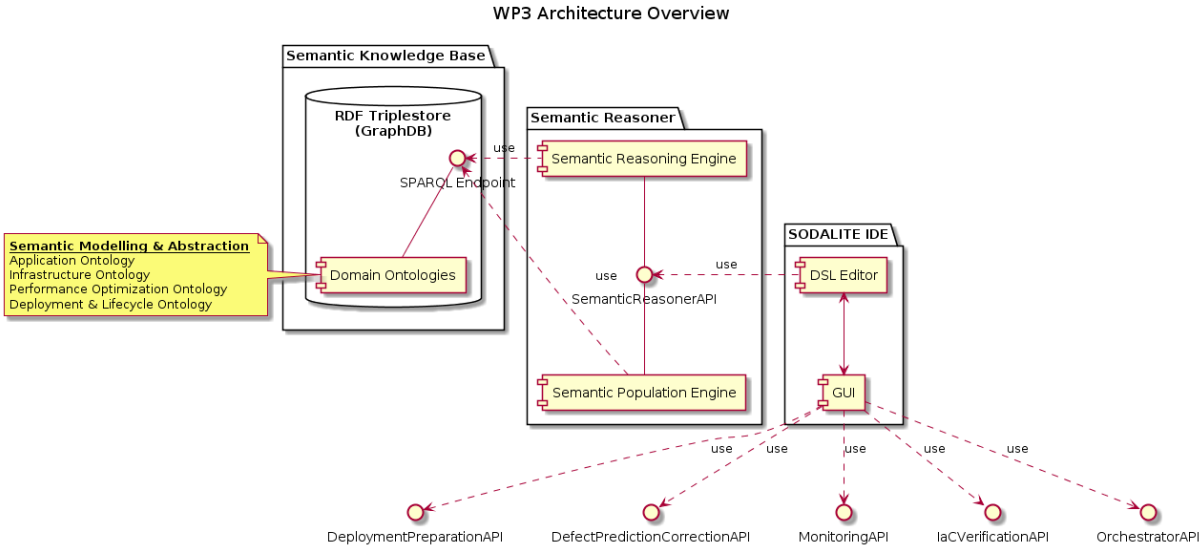

Figure 3: WP3 Modelling Layer Architecture.

Figure 3 shows the internal architecture of the SODALITE Modelling Layer. The used interfaces offered by other components are also highlighted. A set of SODALITE domain ontologies, resulted by the abstract modelling of the related domains (applications, infrastructure, performance optimization and deployment), will be hosted in a SPARQL-served RDF Triplestore (GraphDB), constituting SODALITE's Semantic Knowledge Base. A dedicated middleware (Semantic Reasoner) will enable the exploitation of this repository, mediating for the population of data and the application of rule-based Semantic Reasoning. Last but not least, an IDE will provide a user interface with a DSL editor, for the design of deployment models using knowledge retrieved from the Semantic Reasoner. The IDE will also communicate with other system APIs for the monitoring of the deployment lifecycle.

3.1.1 Component descriptions

3.1.1.1 SODALITE IDE

Functional Description:

The SODALITE IDE provides complete support for the authoring lifecycle of abstract application deployment models (AADM in the following). The IDE enables Application Ops Experts (AOE in the following) to create AADMs for their applications. AADMs are instances of the SODALITE DSL metamodel.

The IDE assists AOEs in the textual/graphical authoring of the AADM thanks to features such as: a) syntax highlighting, b) autoformatting, c) autocompletion and quick fixes, d) validation/error checking, e) scoping (cross-references), f) outlining, etc.

AOEs can describe in the AADM the application topology in terms of components and services, their constraints and inter-component boundaries, and also express optimization requirements or constraints.

The IDE checks the AADM for DSL conformance (syntactic validation) and relies on the Semantic Reasoner for semantic validation (i.e., inconsistencies and/or recommendations). They are presented to the AOE in the IDE for further inspection. Eventually, the AOE can refine/amend the AADM based on them.

Additionally, the IDE can request to the Semantic Reasoner for node resources compatible with application deployment needs. Matching resources are presented to the AOE in the IDE.

Upon completion, the AOE sends the AADM to Abstract Model Parser in order to build an IaC blueprint.

Eventually, the AOE can store an AADM pattern into the Semantic KB in order to be incorporated into the set of reusable patterns for recommendations.

Input:

Authoring of AADM (as instances of SODALITE DSL) requires the following inputs in the SODALITE IDE:

1. AOE manual inputs: the AADM consists on a set of statements (in case of textual editing) or subgraphs (in case of graphical editing), which are compliant with the SODALITE DSL, that are manually introduced by the AOE, based on his/her knowledge of his/her app.
2. Libraries of predefined SODALITE reusable types (and patterns) (which are compliant with the SODALITE DSL) and that can be imported from the SODALITE KB or from the local IDE workspace.
3. Semantic Reasoner inputs. They are Reasoner responses to IDE requests. Possible Reasoner inputs are:
    a. qualitative validation results (inconsistencies, anti-patterns)
    b. node target matches
    c. suggestions/recommendations
    d. Optimisations

Output:

1. An AADM instance to be sent to the Abstract Model Parser
2. An AADM pattern to be sent to the Semantic KB for storage and eventual sharing in other AADM authoring processes.

Programming languages/tools:

- SODALITE DSL: XText, EMF
- SODALITE IDE: Eclipse, Web (Orion)
- SODALITE IDE DSL Editor: XText, Sirius, Java

Dependencies:

1. Semantic Reasoner REST API
2. Semantic Reasoner query language and OWL notation
3. Semantic Reasoner response schema (JSON)
4. Alignment between SODALITE DSL and Semantic KB Schema

Critical factors:

The latency accessing the SODALITE KB (and retrieving request responses) from the IDE may prevent AADM Editor to present real time recommendations, node targets, etc in the code assistance.

AADM patterns need to be serialized in the selected OWL notation before being submitted to the Semantic KB for sharing/reutilization. Therefore, SODALITE DSL and KB Schema must be semantically compatible.

Eclipse DSL technology (XText, EMF, Sirius) might not be fully compatible with a full-fledged Web-based IDE.

3.1.1.2 Semantic Reasoner (Knowledge Base Service - KBS)

Functional Description:

The KBS is middleware facilitating the interaction with the semantic knowledge base (KB). In particular, it provides an API to support the insertion and retrieval of knowledge to/from the KB, and the application of rule-based semantic reasoning over the data stored in the KB.

Input:

1. Requests from the SODALITE IDE for the insertion of domain knowledge from Application Ops Experts and Resource Experts (abstract and target resource types, resource patterns, dependencies, inconsistencies, etc.).
2. Requests from the SODALITE IDE for knowledge retrieval in order to present appropriate content in the IDE, to assure alignment with the DSL, etc.
3. Requests from the SODALITE IDE for the qualitative validation of user input (with the help of semantic reasoning).
4. Requests from the SODALITE IDE for recommendations based on the user requirements.

Output:

1. Domain knowledge (abstract and target resource types, resource patterns, dependencies, inconsistencies, etc.)
2. Detected inconsistencies in a given deployment model.
3. Generated recommendations based on user requirements.

Programming languages/tools: Python, REST API, SPARQL query language

Dependencies:

1. Alignment with the SODALITE IDE and its DSL should be established.
2. The required API services need to be defined.

Critical factors: The imprecise definition of the inputs' structure.

3.1.1.3 Semantic Knowledge Base (KB)

Functional Description: The KB is SODALITE's semantic repository that will host the models (ontologies) created in WP3. The ontologies will be populated with domain knowledge, i.e., abstract and target resource types, resource patterns, deployment patterns, dependencies, inconsistencies, etc. This component will interact with the KBS and will offer capabilities for knowledge storage and manipulation.

Input: Queries from the KBS for the insertion, update, deletion and retrieval of knowledge. More complex queries will also allow the execution of rule-based semantic reasoning and the inference of recommendations and/or inconsistencies.

Output: Requested domain knowledge, recommendations and inconsistencies.

Programming languages/tools:

1. Semantic triplestore with SPARQL support (GraphDB Free version).
2. SPARQL query language.

Dependencies: The semantic models (ontologies) will provide the data storage schema for the KB. Thus, the development of the models directly affects the KB functionality.

Critical factors: The triplestore's scalability needs to be studied, as performance issues might occur upon a great increase in data and querying load.

3.1.2 Use Case Sequence diagrams

3.1.2.1 UC13: Model Resources

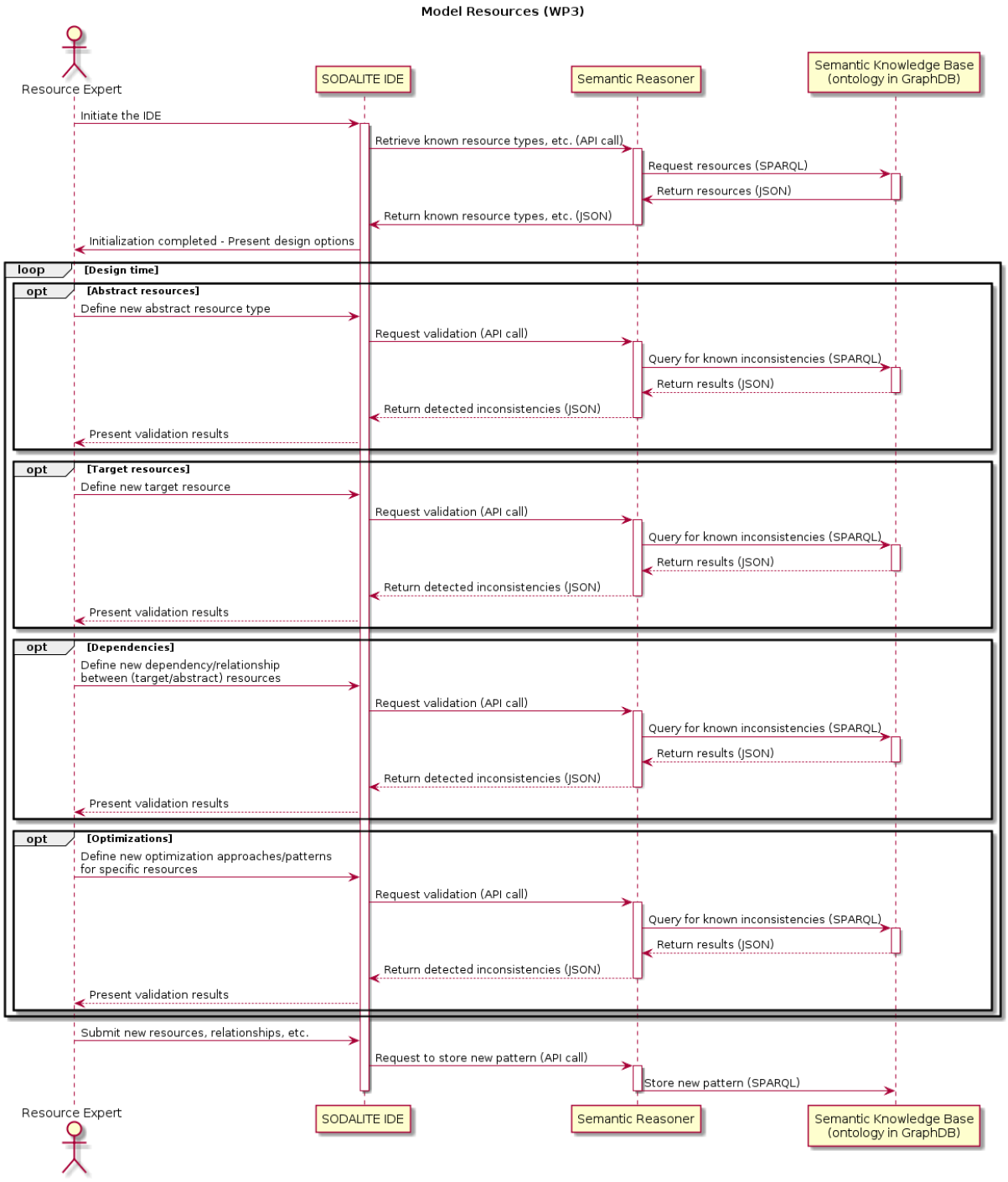

Figure 4: Sequence diagram for UC13.

Figure 4 describes how the SODALITE components cooperate to implement the features offered as part of UC13 - Model Resources. This use case is initiated by the Resource Expert in order to populate and enrich the KB with new definitions of resource types. New knowledge could regard abstract and/or specific resource types, relationships between known entities (e.g., dependencies between resources), patterns and optimisation approaches. The whole process takes place with the use of

the SODALITE IDE and its DSL, assisted by the Semantic Reasoner for the qualitative validation of input and the interaction with the KB.

3.1.2.2 UC1: Define Application Deployment Model

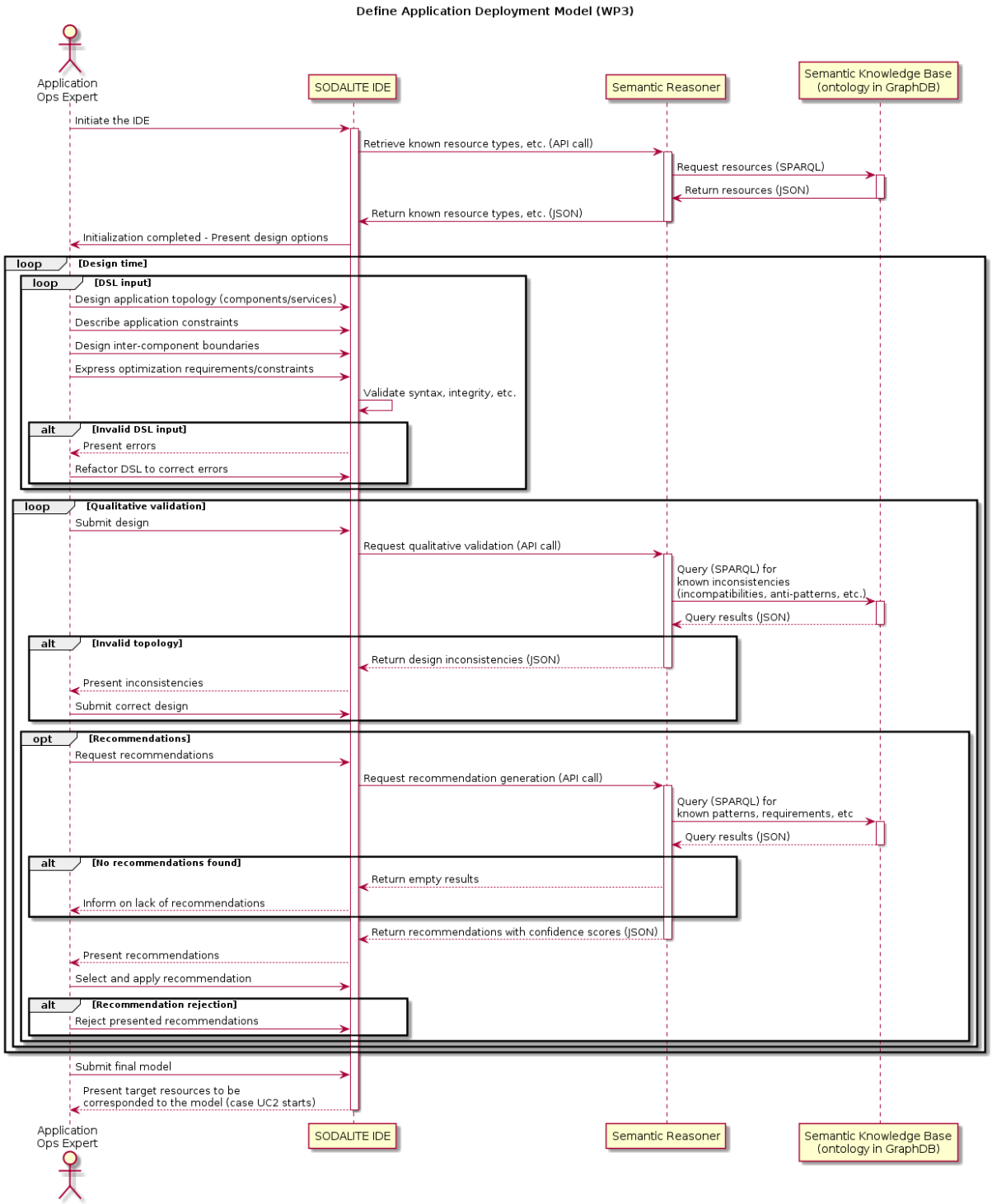

Figure 5: Sequence diagram for UC1.

Figure 5 models the collaboration between the SODALITE components to implement the features required in UC1. The Application Ops Expert (AOE) uses the SODALITE IDE in order to define an application deployment model (ADM). The IDE is charged with presenting existing knowledge (e.g. resource types), validating user DSL input by detecting inconsistencies, and generating

recommendations. The required interaction with the KB is served by the Semantic Reasoner component. The use case output is a valid ADM.

3.1.2.3 UC2: Select Resources

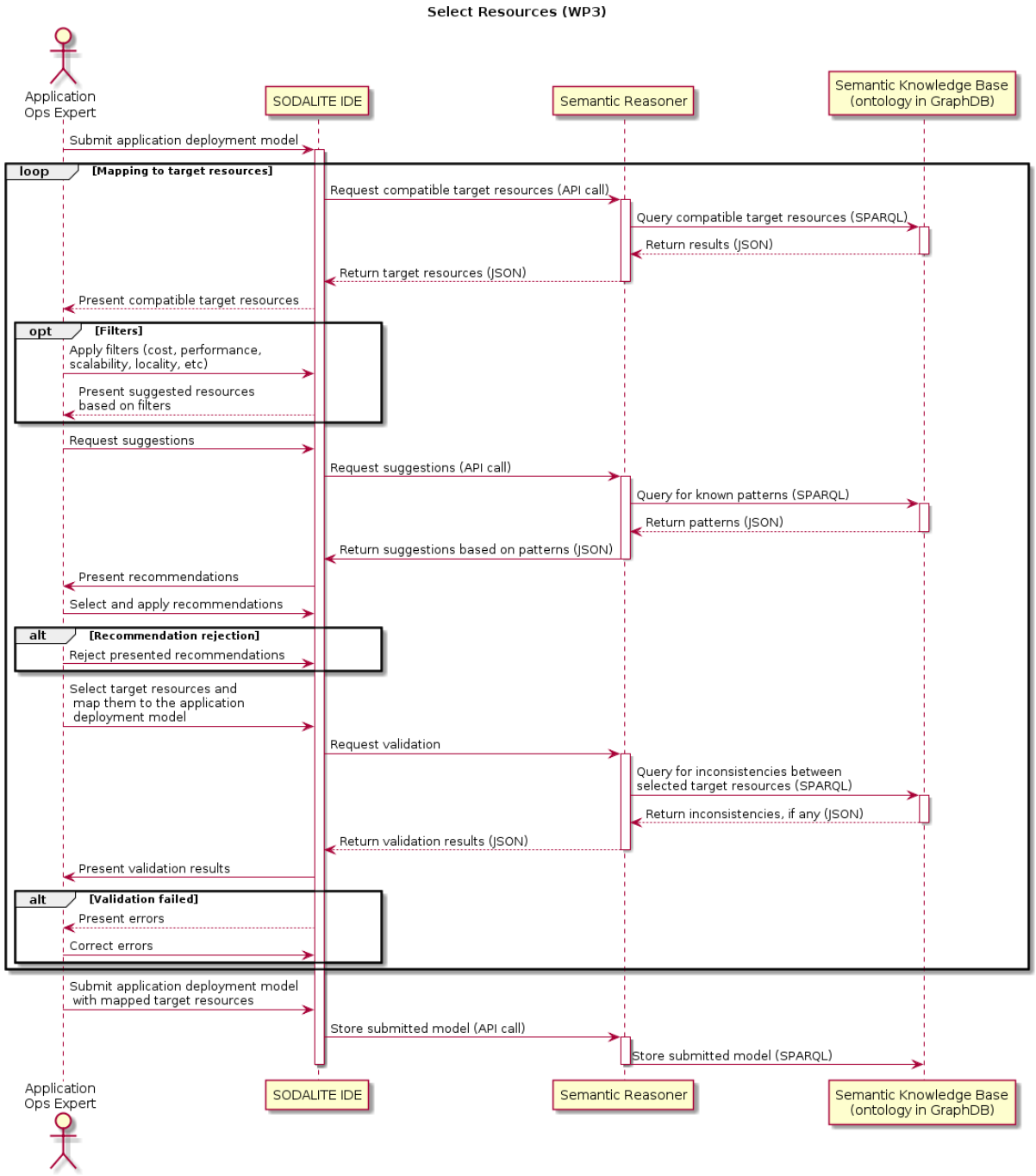

Figure 6: Sequence diagram for UC2.

Figure 6 models the interaction between the SODALITE components when implementing the features offered within UC2 - Select Resources. As soon as an application deployment model, incorporating abstract resource types, has been defined, a selection of target resources needs to be made and mapped to the abstract types, in order to enable the deployment process. This flow includes the generation of suggestions regarding compatible resources and patterns - to which the

user will be able to apply filters - and the validation of provided input, with the support of the Semantic Reasoner and information stored in the Semantic Knowledge Base.

3.1.2.4 UC12: Map Resources and Optimisations

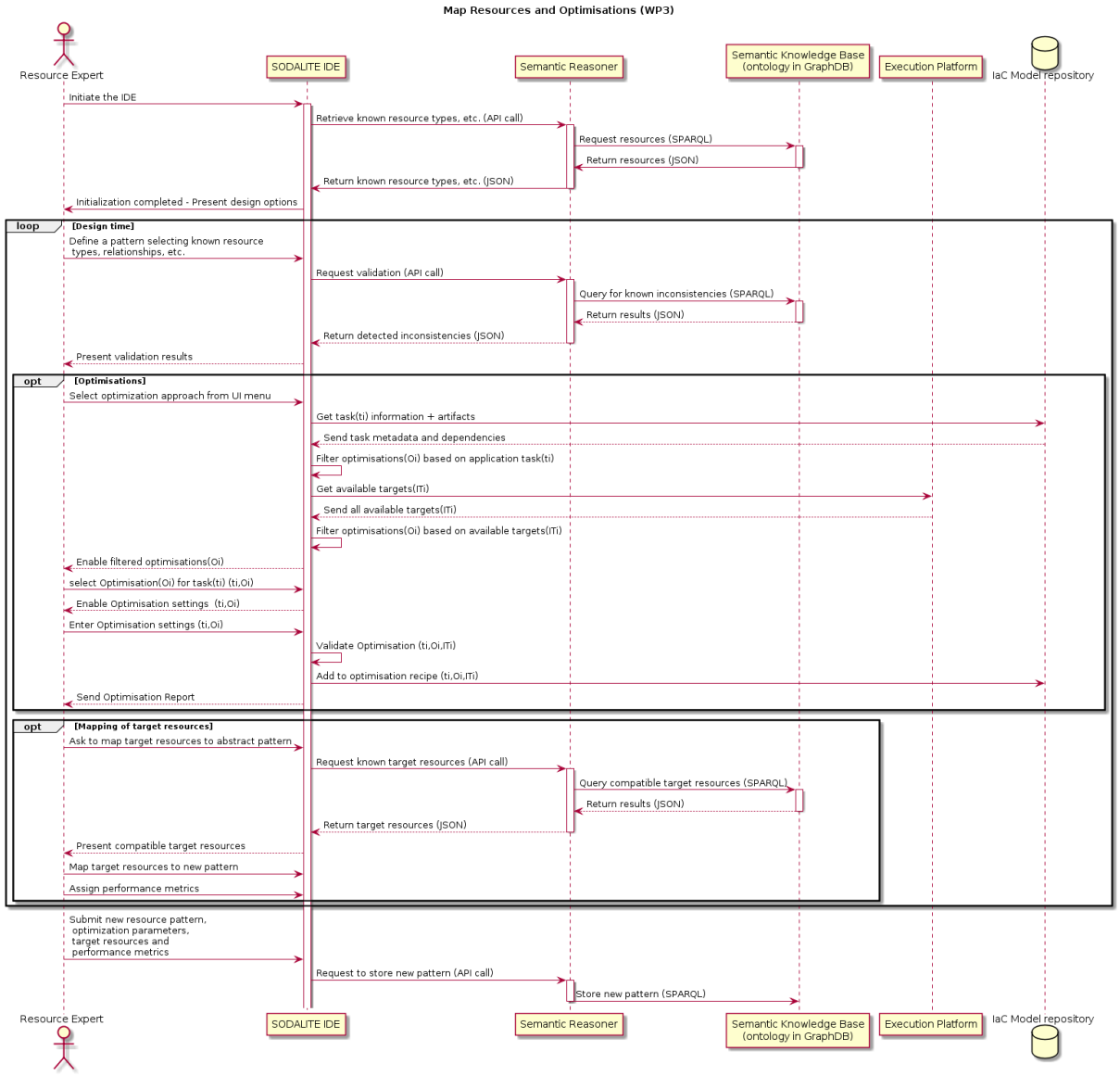

Figure 7: Sequence diagram for UC12.

Figure 7 describes the interaction between the SODALITE components while implementing UC12 - Map Resources and Optimizations. This use case describes the process of defining abstract resource patterns by a Resource Expert (RE). Additionally, actual (target) resources can be mapped to these patterns. To these ends, the SODALITE IDE retrieves and presents known resource types using the Semantic Reasoner. Finally, the newly generated knowledge is stored in the Semantic Knowledge Base and becomes available in related use cases, such as the aforementioned Select Resources.

Moreover, based on the application and available resource types, different optimizations will be enabled for the RE to select from. The RE will also have to enter the settings for any selected optimisation. This will be stored in the IaC Model repository.

3.1.2.5 UC14: Estimate Quality Characteristics of Applications and Workload

We do not include a separate sequence diagram for this use case as the Quality Expert in this case performs the quality assessment experiments. In doing so, he/she exploits the whole SODALITE framework to define the Application Deployment Model (UC1) associated with the experimental prototypes used in the assessment:

- select the resources he/she wants to assess for performance (UC2),
- generate the IaC code (UC3) and possibly verify it (UC4),
- execute provisioning, deployment and configuration (UC6),
- start the prototype (UC7),
- run the monitor to collect data (UC8) and, finally,
- edit the resource and application models (UC13) and (UC1) to include additional information about performance.

Alternatively, the Quality Expert could run the experiments in a simulated environment outside the SODALITE framework and then exploit UC13 and UC1 to update the corresponding models in SODALITE.

3.2 WP4 Infrastructure as Code layer

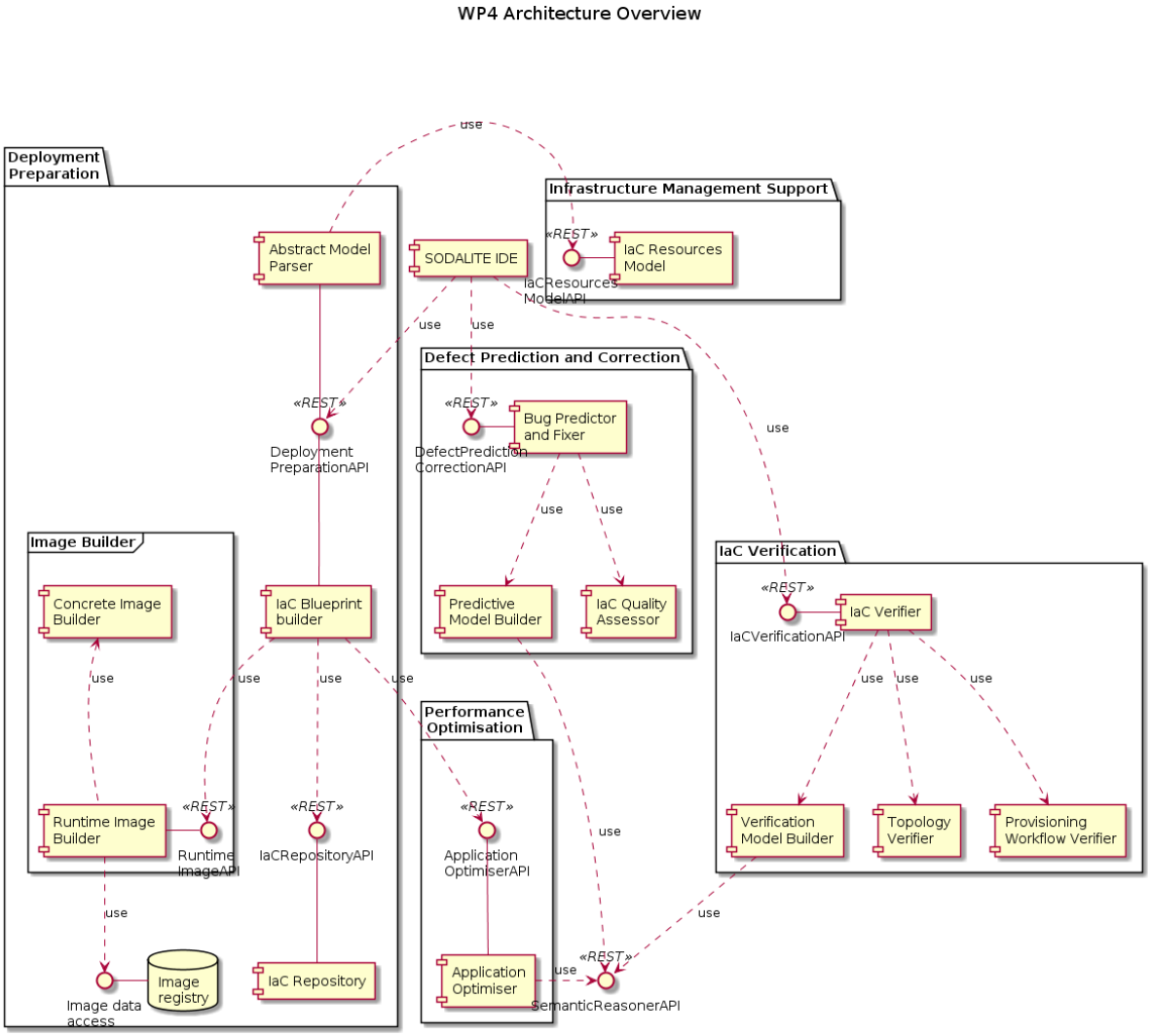

Figure 8: Infrastructure as Code Layer Architecture.

The Generation of Infrastructure as Code (IaC) blueprint builds on the abstract application definition and deployment model from the WP3 Modelling layer and uses the tuple of matching IaC node

definition and abstract application artifact definition with functional and nonfunctional requirements for preparing an optimal IaC blueprint and runtime artifacts for subsequent deployment in WP5 Runtime Layer. The Infrastructure as Code Layer orchestrates the parsing, building, verifying the IaC blueprint with topology and application optimisation in focus, still keeping reference to the underlying abstract model source at all times. Figure 8 shows the internal architecture of this layer, highlighting the way this layer is interfaced with the other two. In the following each component is described.

3.2.1 Component Descriptions

3.2.1.1 Abstract Model Parser

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

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

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

The component allows the SODALITE IDE to:

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

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

Programming languages/tools: Java

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

- IaC Blueprint Builder
- IaC Resources Model

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

3.2.1.2 IaC Blueprint Builder

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

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

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

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

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

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

Input: Abstract application deployment model, IaC Resources Model

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

Programming languages/tools: Java, Python

Dependencies:

- SODALITE IDE
- Abstract Model Parser
- IaC Resources Model
- Application Optimiser
- IaC Repository

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

3.2.1.3 IaC Resources Model

Functional Description: This component holds the IaC Node Definitions and Actuation scripts (playbooks) for the deployment of the application through a set of predefined templates. It is structured as a repository and allows for best matching search.

Input: Definition of abstract node model and constraints.

Output: IaC Node definition and playbooks for actuation

Programming languages/tools: Python/Java

Dependencies: Resource Experts supply infrastructure node definition and relationships with binding to preferred execution platforms. QA Experts supply prefered application binding to infrastructure definition.

Critical factors: N/A

3.2.1.4 Runtime Image Builder

Functional Description: Runtime image builder builds the runtime images

Input: Target architecture and artifact definition

Output: A runtime image equipped with configuration, artifact executable binary, configuration metadata, monitoring artifact. The image is released to the Image repository for deployment.

Programming languages/tools: Python

Dependencies: Concrete Image Builder

Critical factors: N/A

3.2.1.5 Concrete Image Builder

Functional Description: Implementation of concrete image builder for the execution platform to handle specifics regarding configuration, deployment, monitoring.

As it seems there can be significant differences between the images built targeting HPC/Cloud/Kubernetes, Concrete Image Builder implements an adapter pattern to satisfy and bridge the different approaches for targeting the above-mentioned execution platforms.

The built image also includes monitoring artefacts allowing the post deploy configuration by the Orchestrator.

Input: Runtime Image Builder configuration and definition of binary runtime.

Output: Runtime Image

Programming languages/tools: Yaml (Docker, Kompose, HPC container technology), Python

Dependencies: Runtime Image Builder

Critical factors: N/A

3.2.1.6 Application Optimiser

Functional Description: The Application Optimiser Optimises application for a given target platform based on the optimisation options selected.

Input: Optimisation recipe containing the set of Application tasks, Infrastructure targets and optimisations. This recipe will be retrieved from the IaC Model repository.

Further refined inputs are listed below:

1. Artifacts and dependencies for Application tasks
2. Hardware information for Infrastructure targets
3. Optimisation settings for optimisations.

Output: Optimised executable or container, Changes to runtime/deployment and job scripts for submission to resource.

Programming languages/tools: Python, Ruby, CRESTA Autotuning framework

Dependencies: IaC model repository, Runtime Image Builder, Execution Platform

Critical factors: Overhead time for optimisation an application. Validation of Optimisation may require support from the execution platform

3.2.1.7 IaC Verifier

Functional Description: This component coordinates the processes of verification of the application deployment topology and provisioning (deployment) workflow/plan.

Input:

- Abstract IaC models
- Correctness criteria such as well-structuredness, soundness, and application specific constraints

Output:

- Verification Errors (for invalid artifacts)
- Verification Summary (for valid artifacts)

Programming languages/tools: Java / Eclipse

Dependencies:

- SODALITE IDE
- Verification Model Builder
- Topology Verifier
- Provisioning Workflow Verifier

Critical factors: N/A

3.2.1.8 Verification Model Builder

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

Input: Abstract IaC models; Verification knowledge (from Semantic Knowledge Base)

Output: Verification Models

Programming languages/tools: Java

Dependencies:

- Semantic Knowledge Base
- Topology Verifier

Critical factors: N/A

3.2.1.9 Topology Verifier

Functional Description: This component verifies the deployment topology of the application against given correctness criteria and application specific constraints.

Input:

- Formal model of the topology
- Correctness criteria
- Application specific constraints

Output:

- Topology Verification Errors (for an invalid topology)
- Topology Verification Summary (for a valid topology)

Programming languages/tools: Java

Dependencies:

- IaC Verifier
- Verification Model Builder

Critical factors: N/A

3.2.1.10 Provisioning Workflow Verifier

Functional Description: This component verifies the deployment (provisioning) workflow of the application against given correctness criteria and application specific constraints.

Input:

- Formal Model of the Provisioning Workflow
- Correctness criteria
- Application specific constraints

Output:

- Topology Verification Errors (for an invalid provisioning workflow)
- Topology Verification Summary (for a valid provisioning workflow)

Programming languages/tools: Java

Dependencies:

- IaC Verifier
- Verification Model Builder

Critical factors: N/A

3.2.1.11 Bug Predictor and Fixer

Functional Description: This component is responsible for predicting bugs in abstract IaC models (topology and provisioning workflow), suggesting corrections or fixes for each bug, and correcting the bug applying the fix selected by the Application Ops Expert.

Input: Abstract IaC models

Output: Bugs, Fixes

Programming languages/tools: Java

Dependencies:

- SODALITE IDE

- Semantic Knowledge Base
- Predictive Model Builder
- IaC Quality Assessor

Critical factors: N/A

3.2.1.12 Predictive Model Builder

Functional Description: This component builds the ontological predictive model for finding bugs (in topology and provisioning workflow) and suggesting corrections. The bugs are anti-patterns, derivations from design patterns, and other code and design smells. This component also builds a machine learning model that can predict bug proneness index of the IaC artifacts based on IaC metrics.

Input:

- Abstract IaC models or Concrete IaC artifacts
- Bug and resolution knowledge (ontology and rules)
- IaC datasets
- IaC metrics

Output:

- Ontological Predictive Models
- Machine Learning based Predictive Model

Programming languages/tools: Java

Dependencies:

- Bug Predictor and Fixer
- Semantic Knowledge Base

Critical factors: N/A

3.2.1.13 IaC Quality Assessor

Functional Description: This component can calculate different IaC metrics for IaC artifacts.

Input: IaC artifacts

Output: IaC metrics

Programming languages/tools: Java

Dependencies:

- Bug Predictor and Fixer

Critical factors: N/A

3.2.1.14 IaC Model Repository

Functional Description: IaC Model Repository will store the optimisation recipe that will be used by the Application Optimiser to optimise the application and deployment for a target.

Input: Query by Application id or Target id

Output: Add/Modify/Select Optimisations for a selected application or target or both.

Programming languages/tools: Python

Dependencies: None

Critical factors: Model Repository inaccessible to SODALITE IDE or Application Optimiser

3.2.1.15 Image Registry

Functional Description: 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. This

registry and the images will be accessible through a docker-like interface describing the access to a specific image through filtering, labels, image IDs.

Input: Image ID, label or similar

Output: The runtime image of the artefact

Programming languages/tools: Python, Docker

Dependencies: Runtime Image Builder, Orchestrator

Critical factors: accessibility from Runtime Image Builder, Orchestrator.

3.2.2 Sequence Diagrams

3.2.2.1 UC3: Generate IaC Code

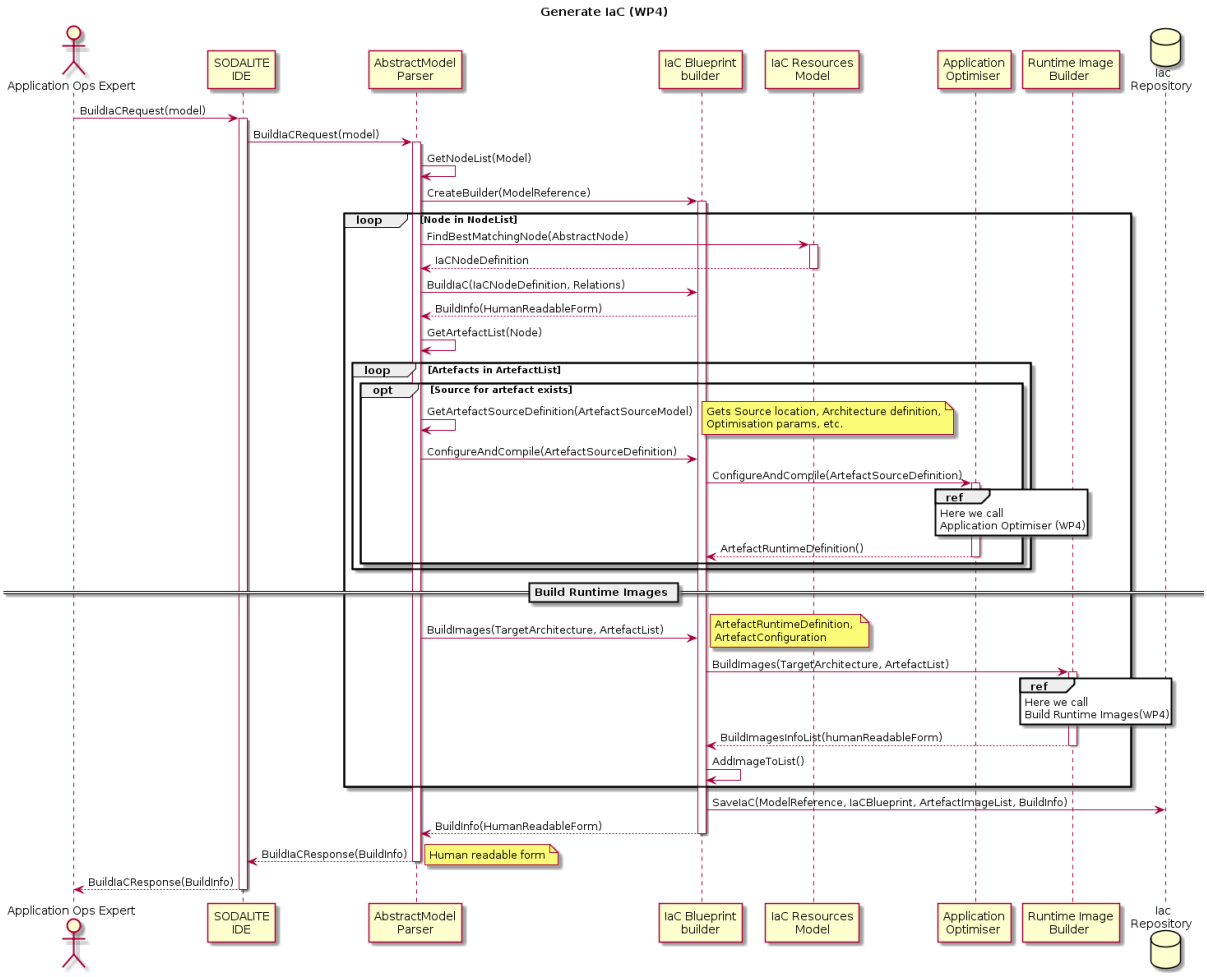

Figure 9: Sequence diagram for UC3.

Figure 9 describes the interaction between the SODALITE components while implementing UC3 - Generate IaC Code. The prerequisites for the IaC blueprint to be built are a well defined abstract application deployment model and definition of artifacts, be it source (scripts) or executable binaries with configuration, to be deployed on the infrastructure. Application Ops Experts initiates the generation of the IaC blueprint through SODALITE IDE with the reference to the model definition. Abstract Model Parser parses the model and replaces the abstract node definitions and relations with optimal IaC node definition from the IaC Resource Model which is built into the IaC blueprint by the IaC Blueprint Builder. Each step is tracked and recorded for subsequent IaC changes reflecting the model. For each node, artifacts definitions with source code are then optimally compiled by the

Application Optimiser component into an executable binary. Immediately after this step the executables are packed into Runtime Images by the Build Runtime Image component. The Runtime Images are stored in the registry for later deployment and the IaC Blueprint is stored in the IaC Repository. The build-time results are returned back to the Application Ops Expert.

3.2.2.2 UC4: Verify IaC

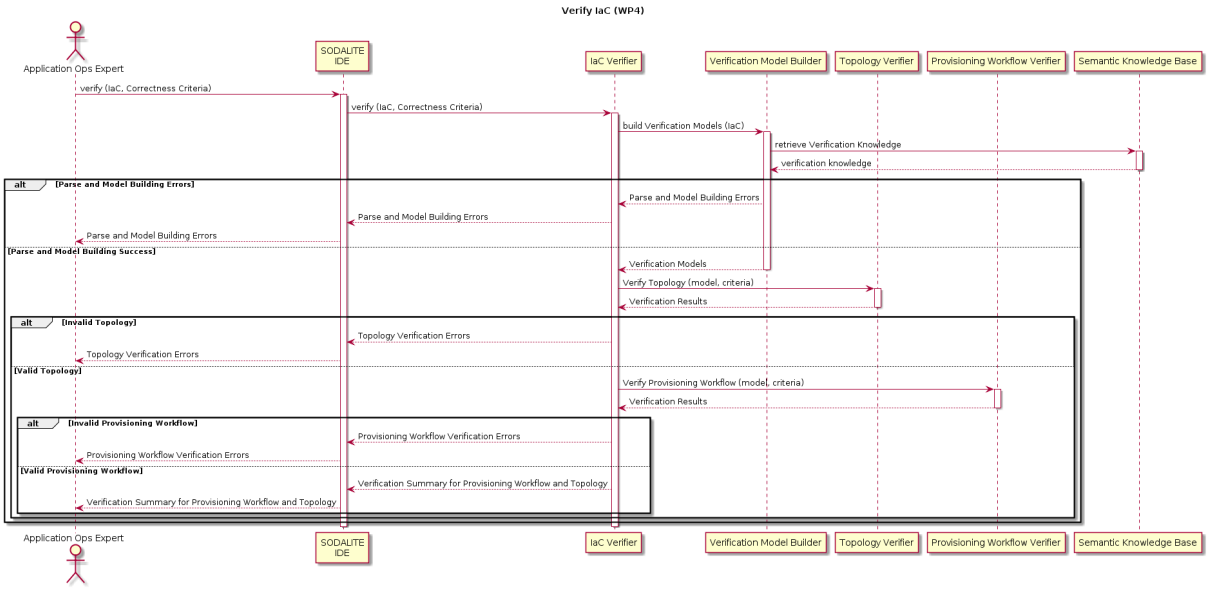

Figure 10: Sequence diagram for UC4.

Figure 10 describes the interaction between the SODALITE components while implementing UC4 - Verify IaC. Application Ops Expert provides the abstract IaC modeling artifacts to the IaC Verifier to formally verify the artifacts with respect to given correctness criteria. Both the deployment model and the provisioning workflow of the application needs to be verified. The provisioning workflow includes the provisioning and configuring of the infrastructure, deployment of the application components on the infrastructure, and configuring the infrastructure and the application components. Verification Model Builder builds the formal verification models using the verification knowledge from the Semantic Knowledge Base. Topology Verifier verifies the topology whereas the Provisioning Workflow Verifier verifies the provisioning workflow. The verification results are returned back to the Application Ops Expert.

3.2.2.3 UC5: Predict and Correct Bugs

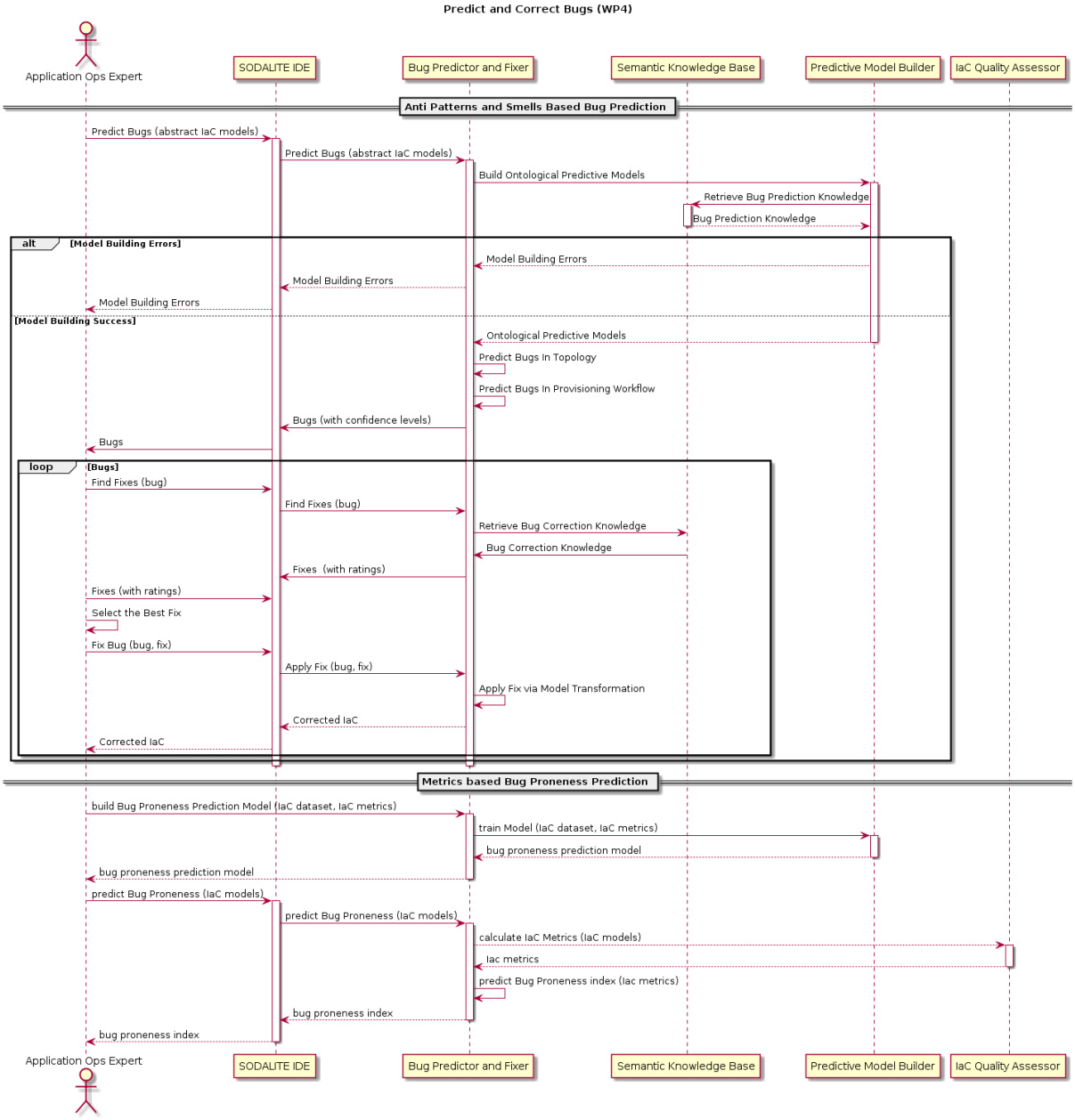

Figure 11: Sequence diagram for UC5.

Figure 11 describes the interaction between the SODALITE components while implementing UC5 - Predict and Correct Bugs. Application Ops Expert submits the abstract IaC models via SODALITE IDE to Bug Predictor and Fixer for detecting the bugs in the application topology and the provisioning workflow. Bug Predictor and Fixer uses Predictive Model Builder to parse the received IaC models, and builds the predictive models required for predicting the bugs in them. The bugs are anti-patterns, design smells, and code smells for security, privacy and performance. The bug prediction results are shown in SODALITE IDE. Application Ops Expert can select one or more bugs, request potential fixes for each selected bug, and choose and apply the desired fixes. The Semantic Knowledge Base contains the knowledge required to predict bugs and to recommend corrections/fixes. Bug Predictor and Fixer can also assess the quality of concrete IaC artifacts in terms of IaC quality metrics, and use the IaC metrics to predict the defect-proneness indices in the IaC artifacts.

3.2.2.4 UC11: Define IaC Bugs Taxonomy

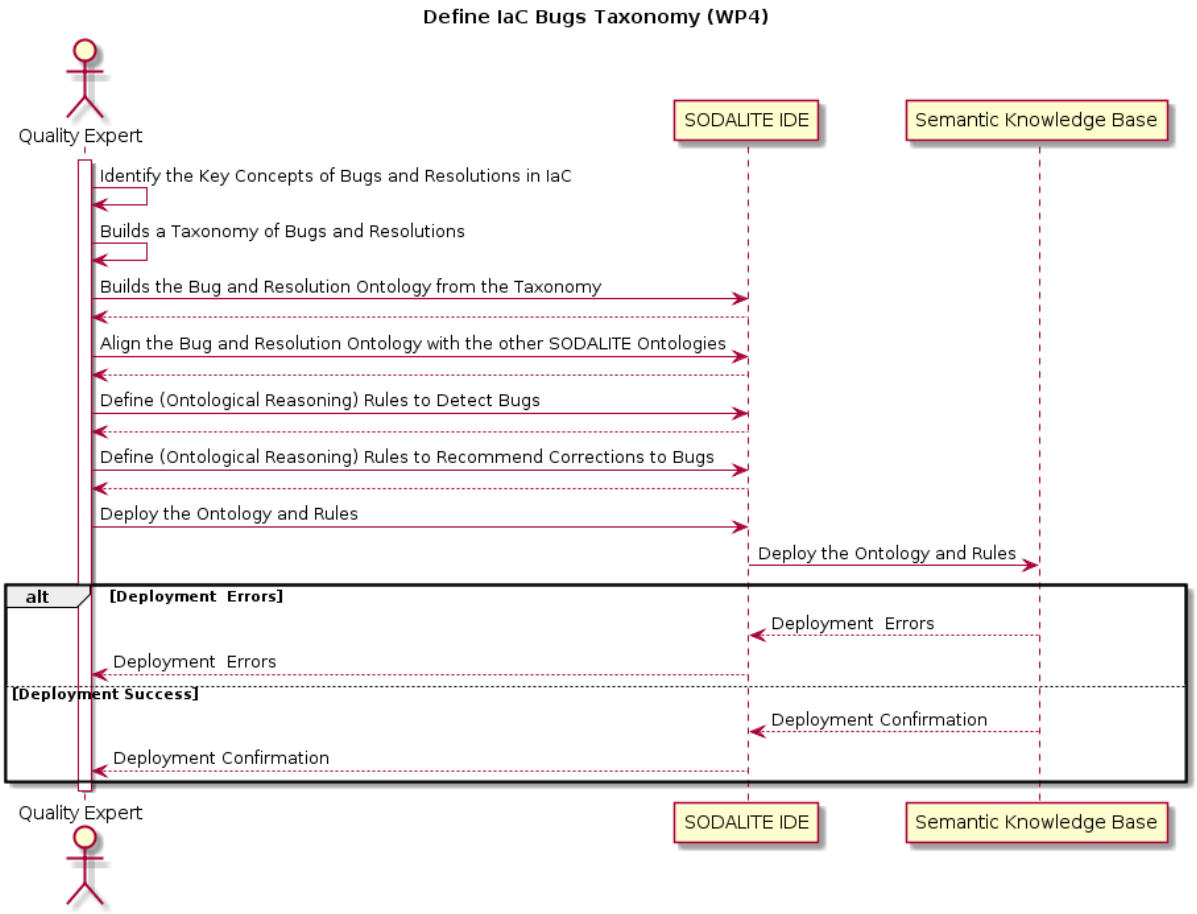

Figure 12: Sequence diagram for UC11.

Figure 12 describes the interaction between the SODALITE components while implementing UC11 - Define IaC Bugs Taxonomy. Quality Expert identifies, defines, and organizes a vocabulary of the key concepts used to describe bugs and their resolutions for IaC. Next, Quality Expert uses the vocabulary to classify, relate, and combine all bugs and resolutions, and builds a taxonomy. Then, the taxonomy is defined formally as an ontology. The ontological reasoning rules required for detecting bugs and suggesting fixes for bugs are also defined. Finally, Quality Expert deploys the ontology and rules in Semantic Knowledge Base.

3.2.2.5 UC15: Statically Optimize Application and Deployment

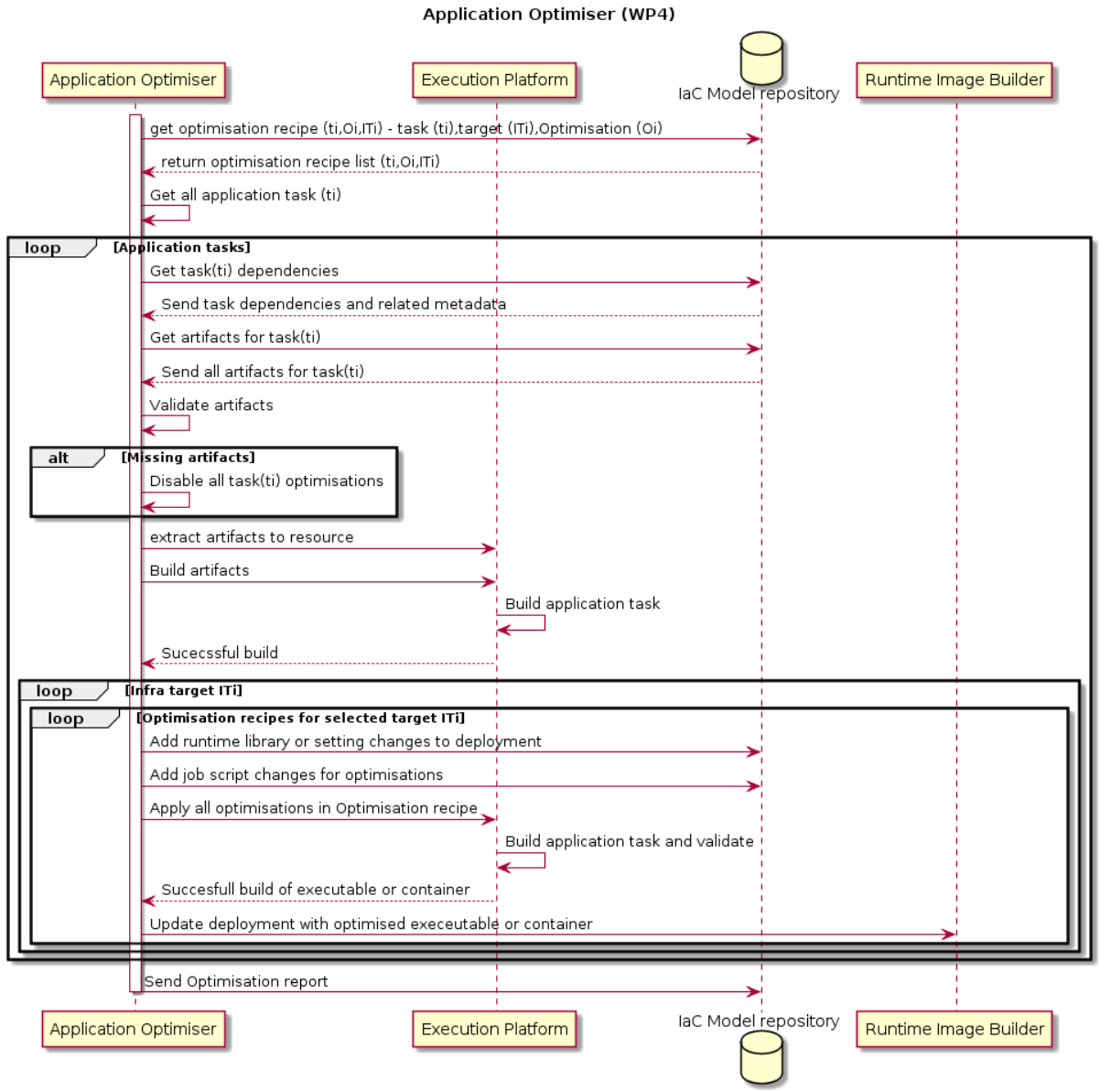

Figure 13: Sequence diagram for UC15.

Figure 13 describes the interaction between the SODALITE components while implementing UC15 - Statically Optimize Application and Deployment. This use case describes the process for optimising the Application and deployment statically. Static optimisation refers to the optimisation before deployment of the application. The input for this use case is the optimisation recipe created as part of the Map resources and optimisation (WP3) use case and the output will be an optimised application executable or a container. The optimization recipe stored in the IaC Model repository is retrieved and extracted. For all the tasks in the recipe, the tasks are optimised for different targets based on the optimisations selected. An optimisation report will be made at the end of this process.

3.2.2.6 UC16: Build Runtime images

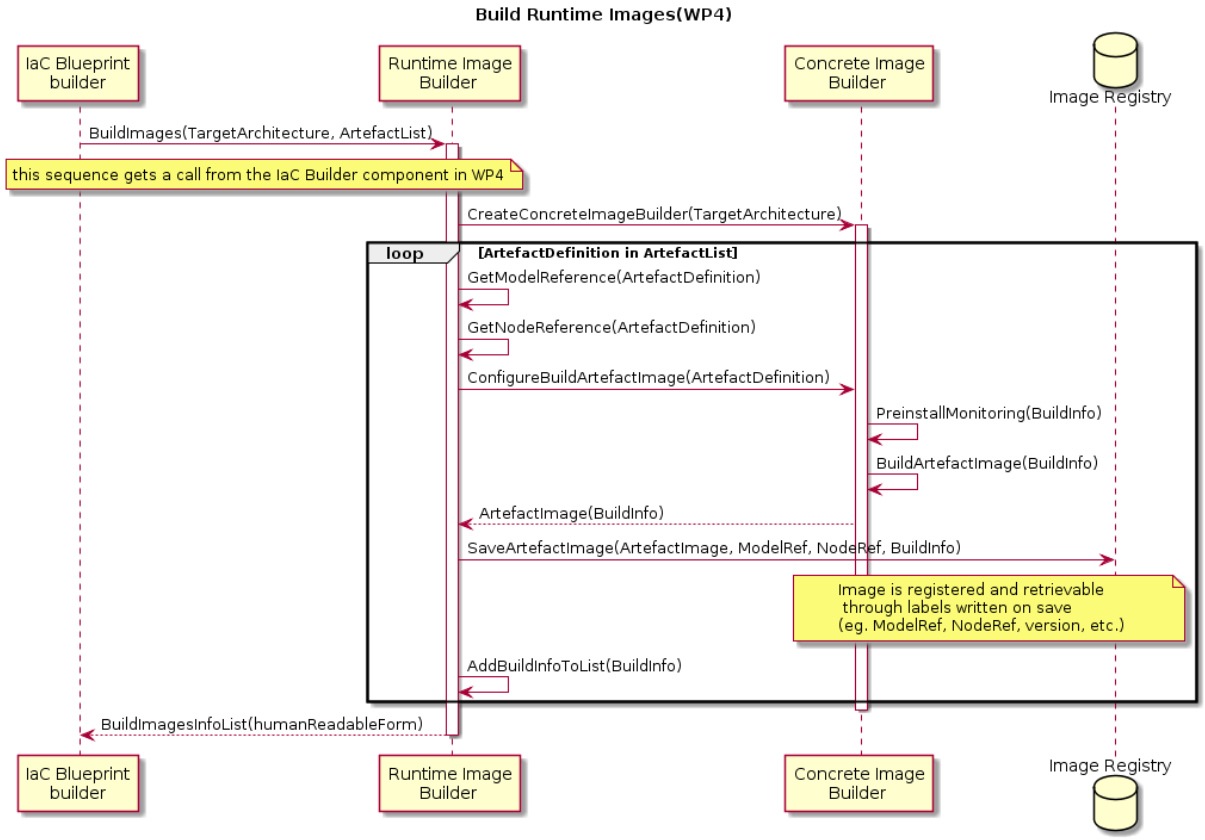

Figure 14: Sequence diagram for UC16.

Figure 14 describes the interaction between the SODALITE components while implementing UC16 - Build Runtime Images. This is an internal process initiated in UC3 - Generate IaC. Runtime Image Builder builds a runtime image based on tuple definition of target architecture and artifact list for that architecture. Runtime Image Builder activates a specific Concrete Image Builder based on target architecture to prepare a runtime image of the artifact and its configuration with added SODALITE monitoring artifact. The built runtime image is then stored in the Image registry for later deployment. The build-time information is returned to the calling component IaC Blueprint builder.

3.3. WP5 Runtime layer

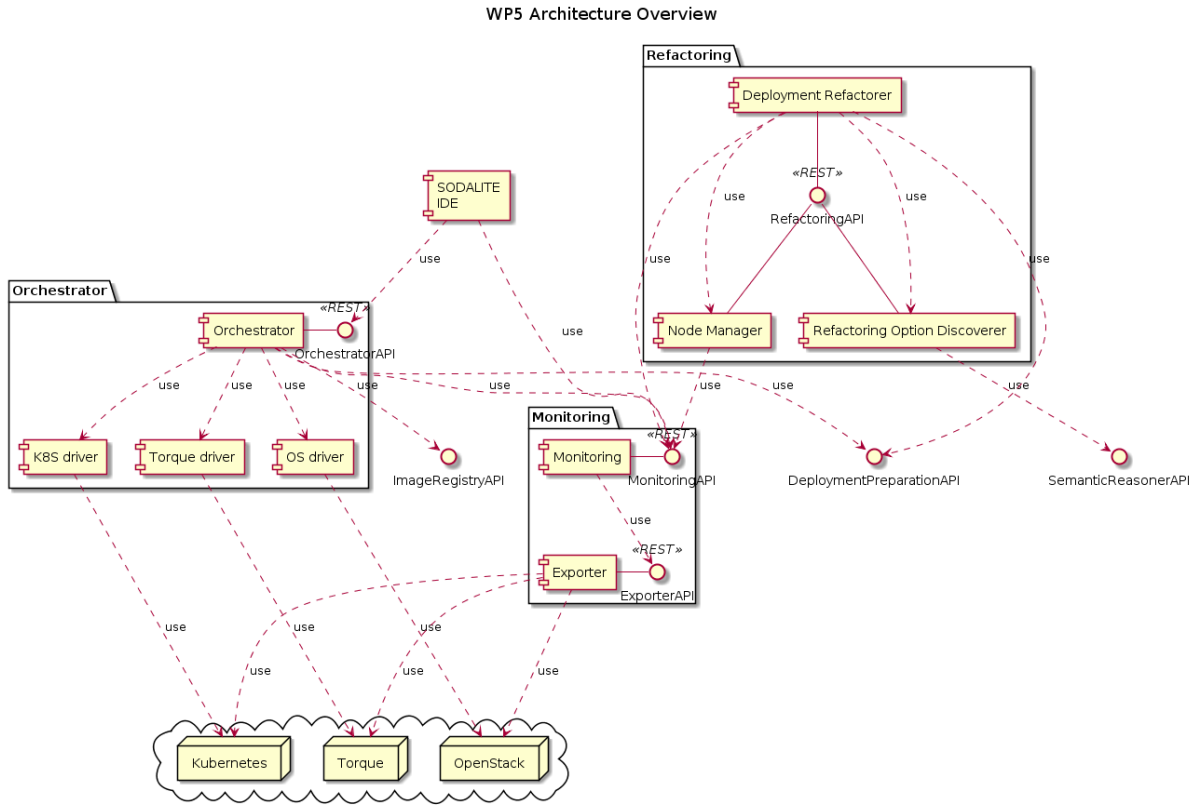

Figure 15: Runtime Layer Architecture.

The Runtime layer of SODALITE (see Figure 15) 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. It is composed of the following main blocks:

- Orchestrator. It receives the application to be deployed or re-deployed as a blueprint expressed in TOSCA, deploying the application components on the appropriate infrastructure.
- Monitoring. It monitors the application components and the infrastructure where they are deployed to be used by Refactoring and the interested SODALITE actors.
- Refactoring. It is able to propose a new application model to fulfill the application goals. When it modifies the model in the Semantic Reasoner, it calls the Deployment Preparation, which will trigger the generation of a new blueprint that arrives to the Orchestrator to initiate the redeployment.

3.3.1 Component Descriptions

3.3.1.1 Orchestrator + Drivers

Functional Description: The Orchestrator manages the lifecycle of an application deployed in heterogeneous infrastructures.

Input: Deployment plan

Output: Commands to target infrastructures

Programming languages/tools: Python, xOpera, Cloudify.

Dependencies: Target infrastructures: HPC, Kubernetes, OpenStack…

Critical factors: Each orchestrator has its own limitations. This results in limitations concerning the possibility to apply certain actions on the managed application. For instance, since Cloudify does not support SubstitutableNodes, it does not support the Refactoring feature.

3.3.1.2 Monitoring

Functional Description: Gathers metrics from the heterogeneous infrastructure and application execution, allowing query and aggregation on them.

Input: Heterogeneous infrastructure

Output: Metrics

Programming languages/tools: Prometheus, Prometheus query language and API

Dependencies: Monitoring agents in each of the infrastructure components. We'll need to probe each agent according to its specific API or we'll need intermediate connectors that translate between Prometheus style APIs to the APIs of the specific monitoring agent.

Critical factors: Certain metrics could be difficult to get in some infrastructures.

3.3.1.3 Deployment Refactorer

Functional Description: This component refactors the deployment model of an application in response to violations in the application goals. It also derives the node-level goals from the application goals. The goals are monitored at runtime by collecting the necessary metrics. A machine learning based predictive model is used to select a valid set of refactoring options to derive a valid variant of the deployment model (the new deployment model). The new abstract deployment model is transformed to TOSCA and IaC Scripts using the Deployment Preparation module. Optionally, an adaptation plan can be generated based on the differences between the new deployment model and the current deployment model. The new refactoring options as well as the changes to the existing refactoring options can be discovered at runtime.

Input:

- IaC topology model
- Refactoring option model
- Application goals
- QoS metrics

Output: (Topology) Adaptation Plan or New Deployment Model (in TOSCA and IaC Scripts)

Programming languages/tools: Java

Dependencies:

- Refactoring Option Discoverer
- Node Manager
- Deployment Preparation
- Semantic Reasoner
- Monitoring Agent

Critical factors: N/A

3.3.1.4 Node Manager

Functional Description: This component is responsible for managing node resources including the overall node capacity/throughput while maintaining the node goals assigned by the Deployment Refactorer. The node goals are monitored at runtime by collecting the necessary metrics. The node may have policies for auto-scaling and availability.

Input: Node goals, QoS metrics

Output: Node Resource Allocation Decisions (e.g., limit admission of requests, change CPU allocations, etc.)

Programming languages/tools: Java

Dependencies:

- Deployment Refactorer
- Orchestrator
- Semantic Reasoner
- Monitoring Agent

Critical factors: N/A

3.3.1.5 Refactoring Option Discoverer

Functional Description: This component is responsible for discovering new refactoring options and changes to existing refactoring options based on design patterns and anti-patterns (in general, topology level defects). For example, a new instance of a pattern that may offer better performance or security may be found.

Input: Search criteria (patterns, anti-patterns, QoS goals, and constraints)

Output: Refactoring options

Programming languages/tools: Java

Dependencies:

- Deployment Refactorer
- Monitoring Agent
- Semantic Reasoner

Critical factors: N/A

3.3.2 Sequence Diagrams

3.3.2.1 UC6: Execute Provisioning, Deployment and Configuration

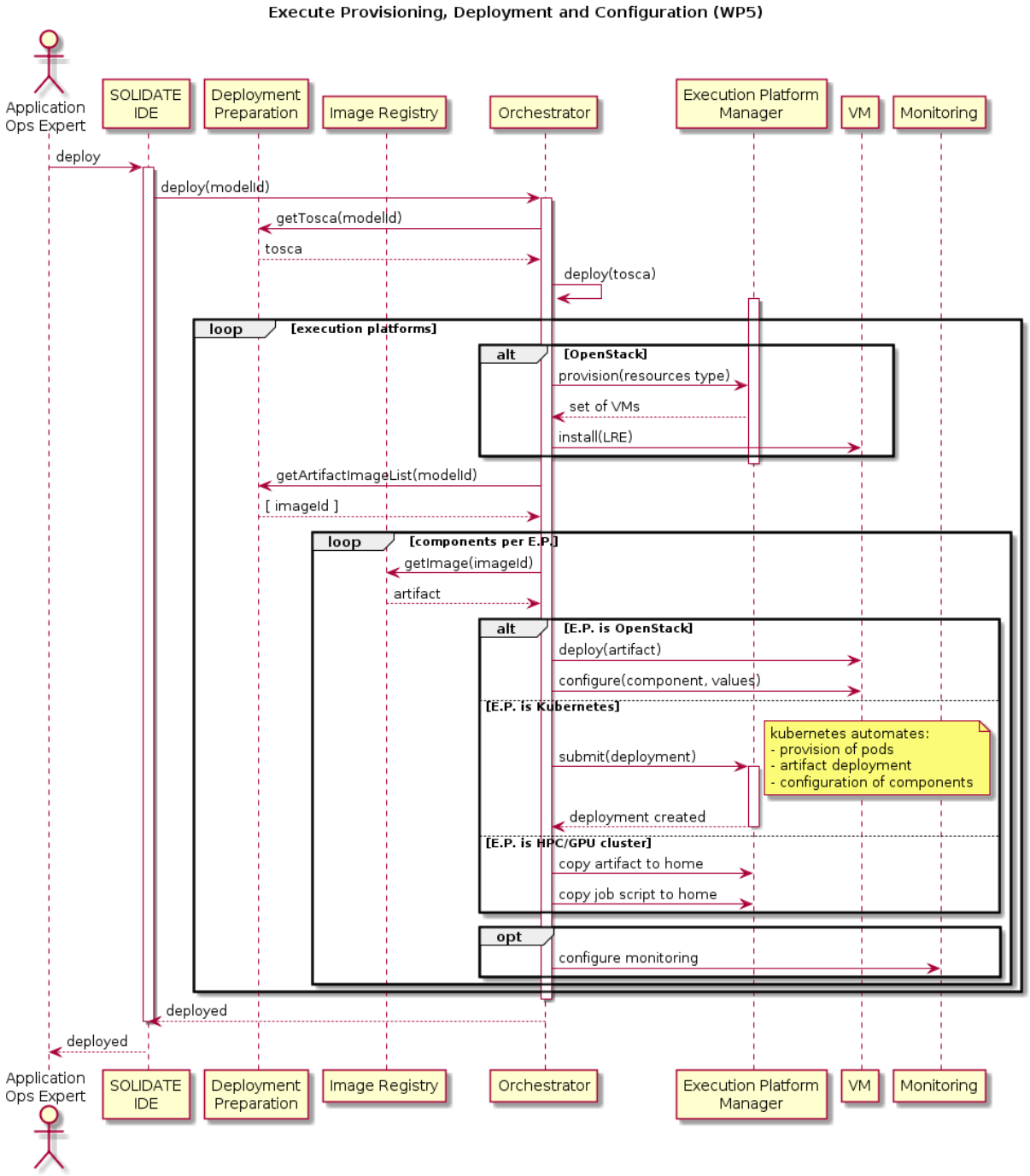

Figure 16: Sequence diagram for UC6.

Figure 16 describes the interaction between the SODALITE components while implementing UC6 - Execute Provisioning, Deployment and Configuration. Application Ops Expert initiates the deployment via SODALITE IDE, which in turn provides IaC blueprints and playbooks to the Orchestrator. Upon this, the Orchestrator interacts with the Execution Platform Manager to provision Resources for the deployment and configure the application components. In this sequence diagram, the following Execution Platform Managers are considered: OpenStack as the virtual

infrastructure provisioner, Kubernetes as a provisioner of the infrastructure for containerized applications and batch job manager to provide compute Resources within HPC and GPU clusters. All of the considered execution platforms are assumed to provide unified Lightweight Runtime Environment (LRE) to deploy the generated application artefacts in the different execution platforms.

With OpenStack, before the deployment, the virtual Resources must be firstly provisioned. For that, the Orchestrator requests OpenStack to provision a set of virtual machines (VMs) and other resources that the application demands, e.g., network and storage. The LRE is then installed on VMs for the deployment of the application artefacts. Upon the deployment, the Orchestrator configures application components.

Kubernetes automates resource provisioning and application deployment by providing an interface, through which the deployment and configuration descriptions are passed. Hence, the Orchestrator submits these descriptions to Kubernetes, which deploys the application artefacts and configures them afterwards.

When HPC or GPU clusters are selected as the Resources for deployment, the Orchestrator pre-uploads the artefacts and job description script to the user workspace (e.g. home directory of the user) on login (front-end) nodes before submitting the job to the batch job manager.

At this point, the deployment of the heterogeneous application components is performed on different Resources and the application is ready to be started. Optionally, the configuration of a monitoring platform can be additionally executed if such mechanisms are provided to the Orchestrator.

3.3.2.2 UC7: Start Application

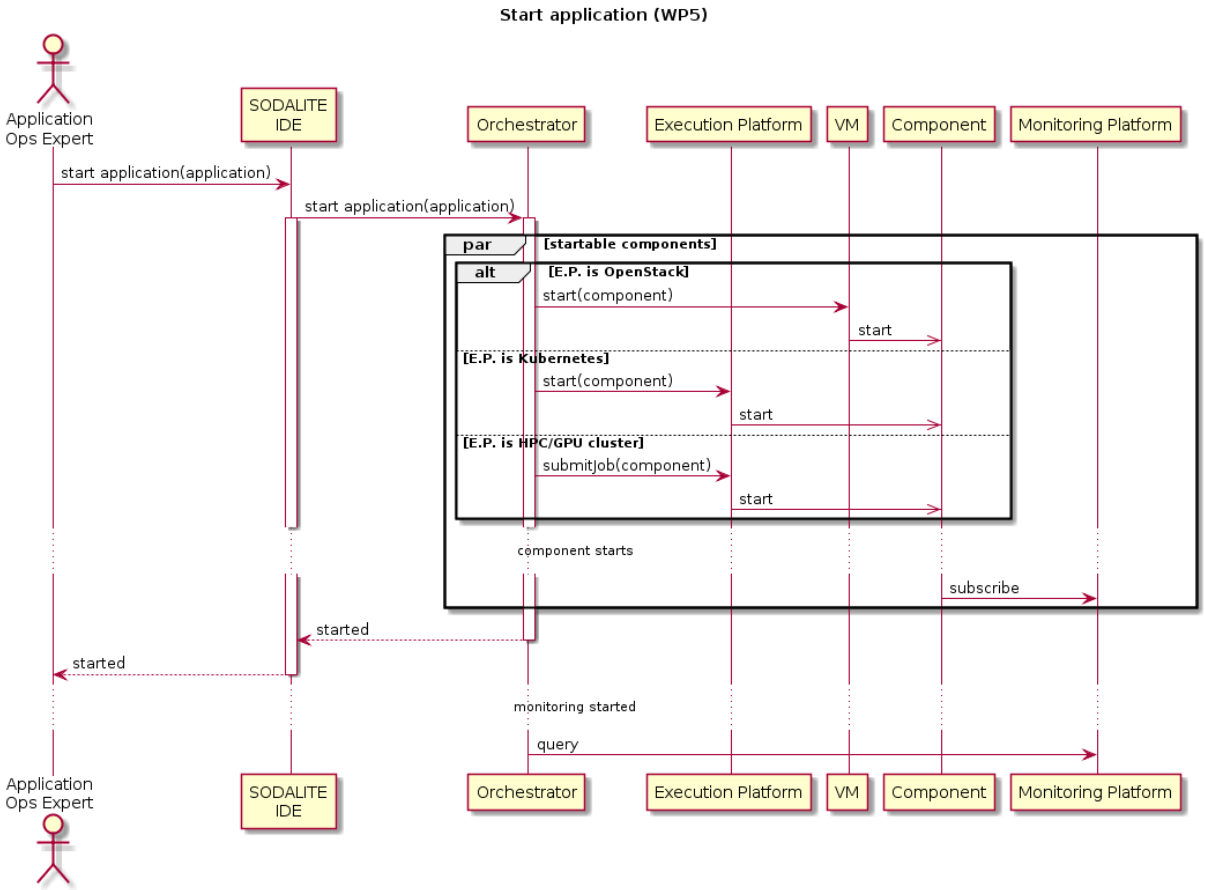

Figure 17: Sequence diagram for UC7.

Figure 17 describes the interaction between the SODALITE components while implementing UC7 - Start Application. This sequence diagram describes the steps performed when an application is started. In case of a service (e.g., web application, REST service), the application is started after the deployment and runs until its termination. In case of non-services applications - which start, do some processing and end (e.g., HPC application) -, applications may be started several times for a single deployment.

The Application Ops Expert decides to start an application by using the SODALITE IDE. This request arrives to the Orchestrator, which in turn starts the execution of each of the components that compose the application. When the component starts, the execution environment of the application sends a message to Monitoring Platform to indicate that the component is alive and prepared to send metrics. Once the components register to Monitoring, Orchestrator can initiate collection of metrics for its purposes (e.g., check application health).

3.3.2.3 UC8: Monitor Runtime

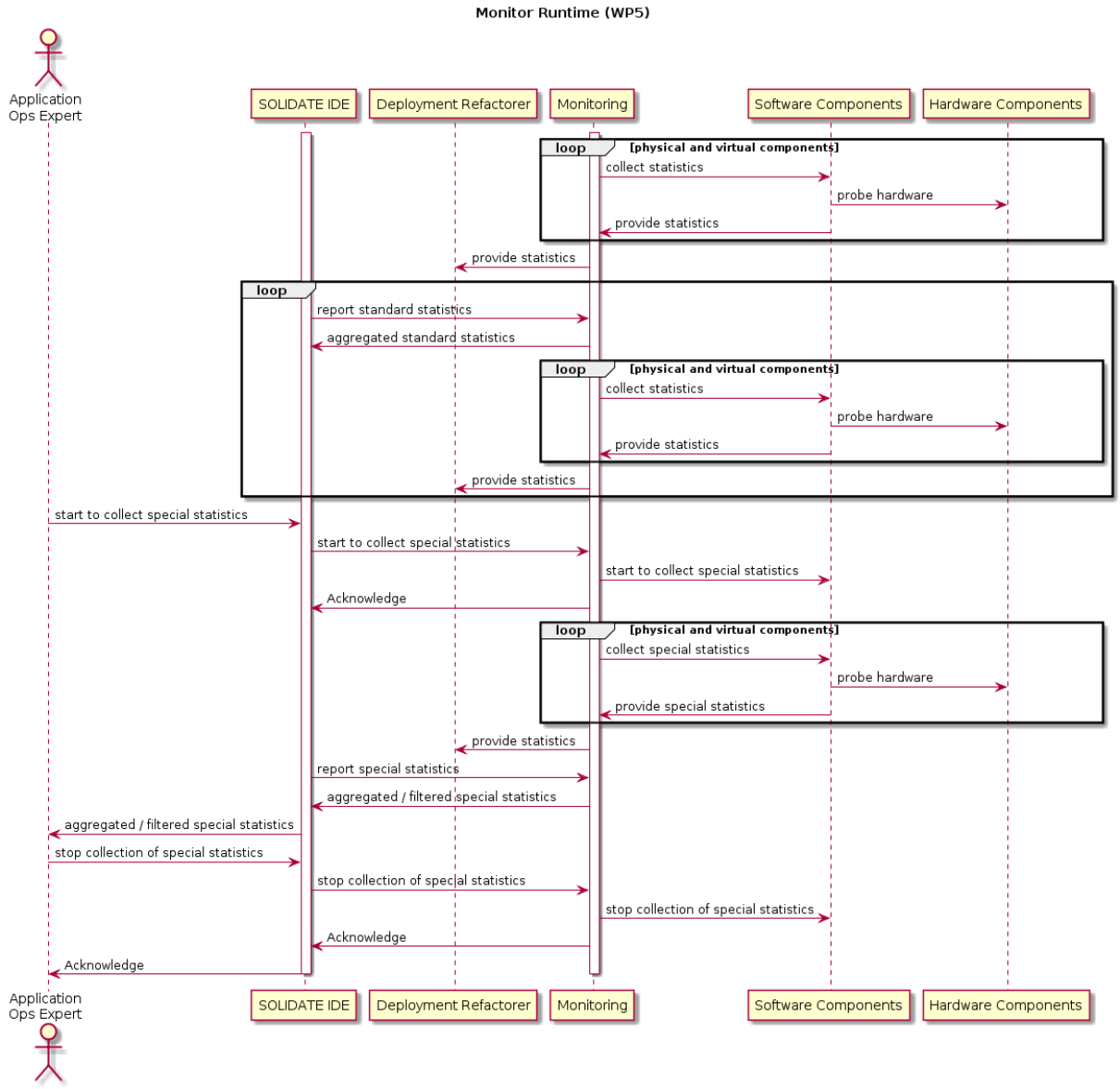

Figure 18: Sequence diagram for UC8.

Figure 18 describes the interaction between the SODALITE components while implementing UC8 - Monitor Runtime. The Monitor component collects system statistics on an ongoing basis. On each host (whether physical or virtual) there is a software component that interacts with the Monitoring component and reports standard system statistics. The statistics are usually collected on each (physical or virtual) host by reading various counters and registers that hold updated system statistics. These combined statistics are collected and periodically reported to a dashboard that is part of the SODALITE IDE. These statistics are also available to be used by the Deployment Refactor component to make placement decisions based on resource usage. In some cases it may be desirable to collect some specific non-standard statistics in order to isolate the cause of some observed anomaly. In this case, the operator can request to collect additional specific statistics. This request is translated by the Monitoring component into requests to the agents running on each (physical or virtual) host. When the operator no longer needs the collection of the non-standard statistics, he informs the Monitoring component to stop the collection of those statistics

3.3.2.4 UC9: Identify Refactoring Options

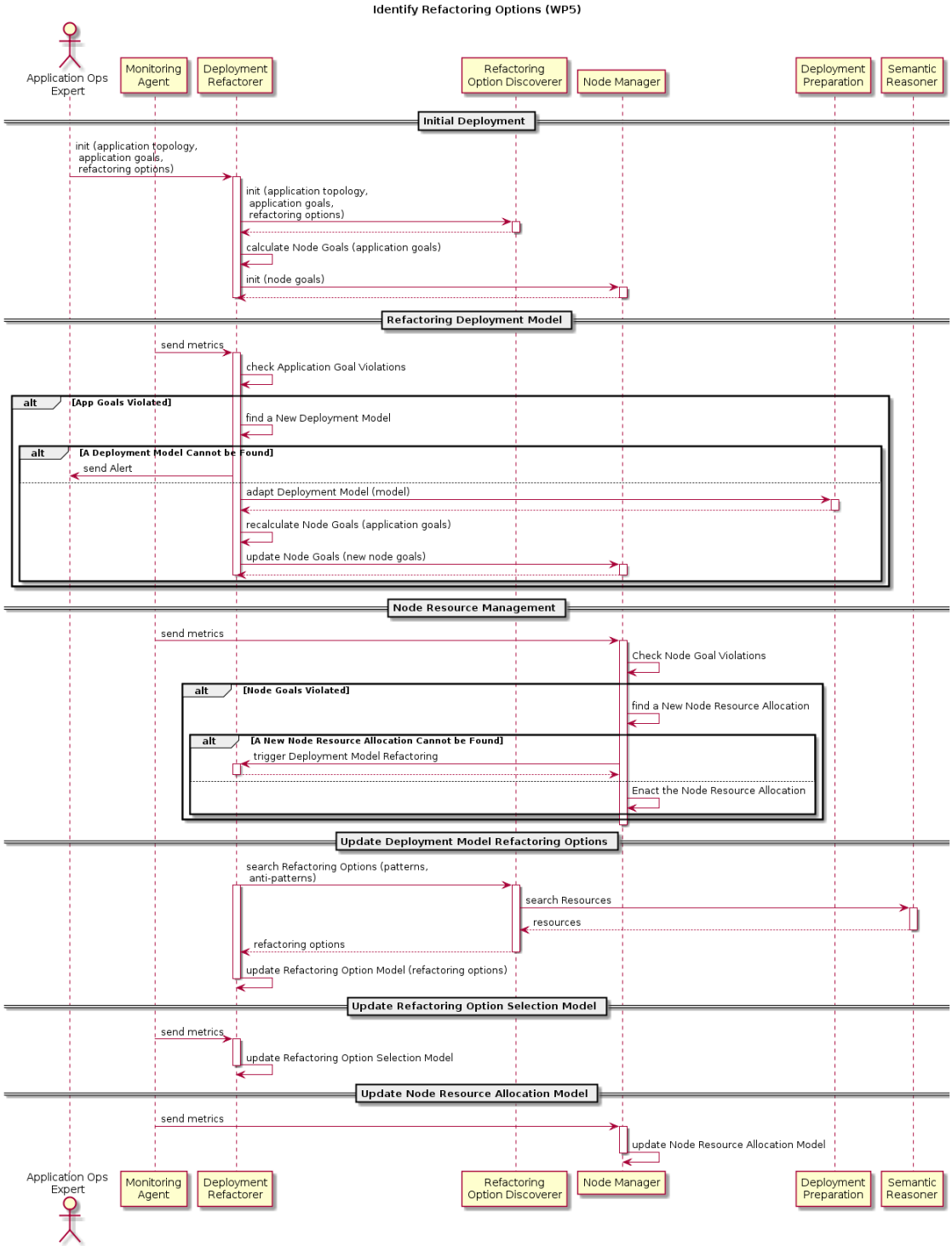

Figure 19: Sequence diagram for UC9.

Figure 19 describes the interaction between the SODALITE components while implementing UC9 - Identify Refactoring Options. Deployment Refactorer is initialized with the IaC models for the initial deployment, the initial set of refactoring options, and application goals. It uses the Node Managers of each of the nodes in the application topology to manage the resources in those nodes. The node resource management is based on the node level goals derived from the application goals. Via Monitoring Agent, Deployment Refactorer monitors the application goals. If application goals violations are detected, Deployment Refactorer tries to find a new deployment model for the

application that can resolve the detected application goal violations. If a new deployment model cannot be found, Application Ops Expert is alerted. The new deployment is enacted via Deployment Preparation component. Deployment Refactorer also may reassign node-level goals as necessary. Refactoring Option Discoverer can find the new refactoring options as well as the changes to the existing refactoring options using patterns and anti-patterns (bugs). It uses Semantic Reasoner for this purpose. Both Deployment Refactorer and Node Manager use the Monitoring Agent to collect data to determine the impacts of the refactoring decisions and to update the predictive models used for refactoring option selection and node resource allocation, respectively.

3.3.2.5 UC10: Execute Partial Redeployment

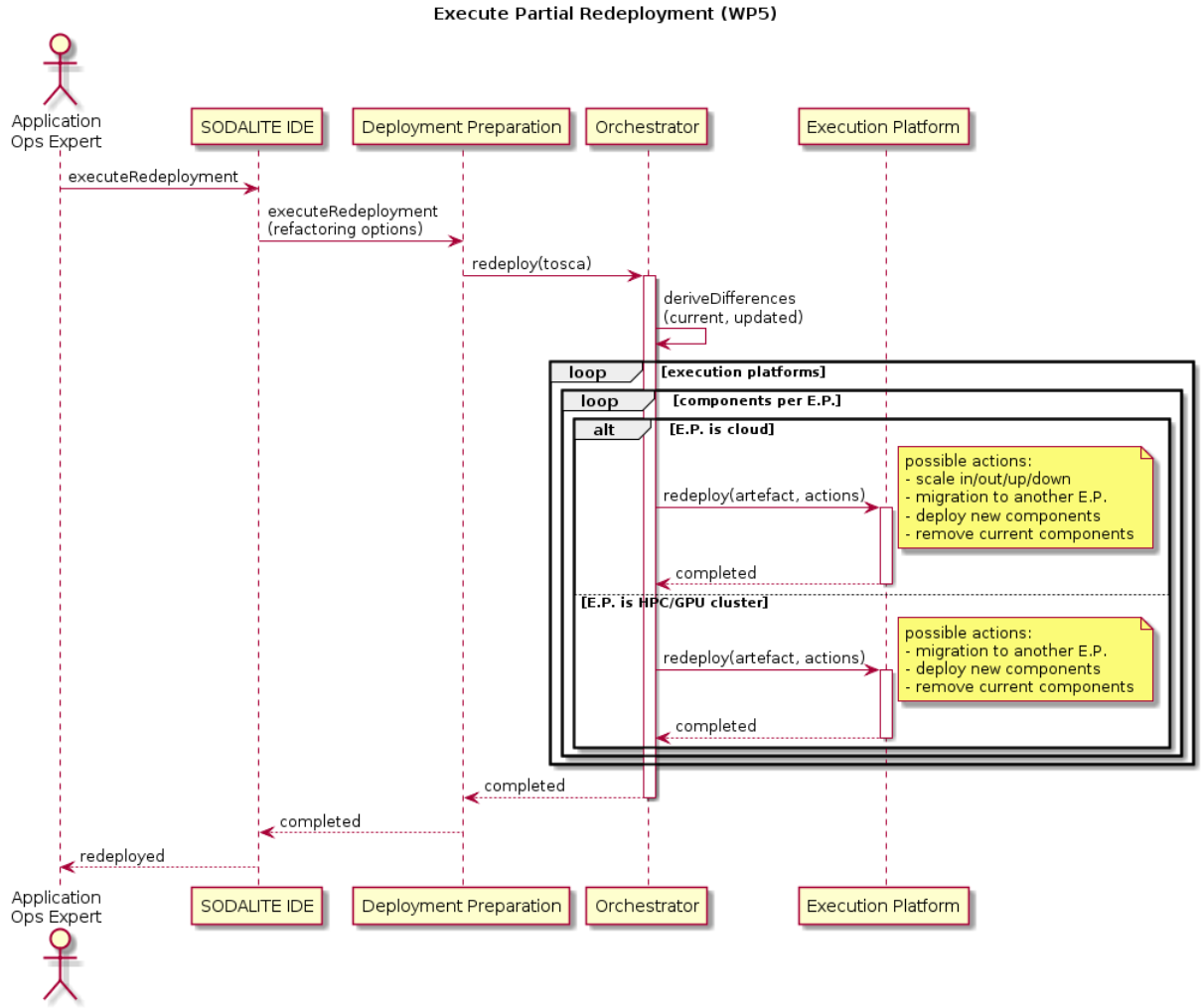

Figure 20: Sequence diagram for UC10.

Figure 20 describes the interaction between the SODALITE components while implementing UC10 - Execute Partial Redeployment. After the Refactoring Options have been identified and Deployment Model has been updated, Application Ops Expert initiates redeployment via SODALITE IDE, which in turn instructs Deployment Preparation Module to provision the IaC blueprints and playbooks of the new deployment. The Orchestrator derives the difference between current and updated deployments and applies adaptation actions until the current state of deployment becomes the updated state.

Such adaptation actions are performed on the Execution Platforms used for the redeployment. If the selected platform is Cloud, the actions that might be performed are the following:

- any form of scaling (in/out/up/down),

- migration to another Execution Platform,
- deployment of the new application components introduced by the Application Ops Expert and removal of current components.

For what concerns HPC and GPU clusters, as the corresponding Execution Platforms lacks flexibility in scaling at runtime, the scaling actions are not present as possible adaptation actions in the sequence diagram; however, all the other actions applicable to Cloud can be executed on these platforms as well (migration, deployment and removal of components).

3.4. Mapping SODALITE Architecture with the SODALITE Exploitable Results

From the Grant Agreement preparation till now the SODALITE consortium understanding of the problem being tackled has evolved and is resulting in a more precise definition of the elements belonging to the SODALITE solution. As a consequence of this, various elements presented in this deliverable were not available in the Grant Agreement and others have changed their names. The purpose of the following table, therefore, is to clarify the mapping between the terminology adopted in the Grant Agreement and the current terminology. Based on the development of the project, the list of the exploitable results might evolve over time. The table below represents the current understanding of the consortium.

	WP/Task	Exploitable results	Description	Corresponding element in D2.1
ER#0	6	SODALITE Stack	a complete set of tools and software components, developed in this project. It joins all SODALITE elements under one brand and allows the project to present its outputs using one name - SODALITE. SODALITE Stack will enable mix and match selection of tools, used for different use cases and different infrastructures.	SODALITE Platform
ER#1	3	Abstraction Components Library	a library of basic component abstractions, along with patterns related to application, infrastructure and performance. Application developers and infrastructure operators will exploit this library to best describe their applications or infrastructures. This library is the central semantic part of the project. SODALITE abstractions are human and machine-readable and are used for the creation of Abstracted Application Tuples.	Semantic Knowledge Base
ER#2	3	Abstracted Application Tuple	A tuple materialises an application modelled by exploiting the components library above. Each tuple comprises an abstract description of the application, its infrastructure, and its non-functional requirements. The tuple is then used to feed the Application Builder and create an Application Pattern Implementation.	Application Deployment Model

ER#3	4	Application Container	This takes the form of a container with the application, its deployment plan and the runtime constraints. This is used by Runtime, to actually deploy the application on the infrastructure.	Runtime Image Builder
ER#4	3	Application Developer Editor	This corresponds to the SODALITE application modelling IDE, used to compose and describe applications, their components, and performance using the Abstraction Components Library.	SODALITE IDE, Semantic Reasoner
ER#5	4	Application Builder	A tool that takes an Abstracted Application Tuple and constructs an Application Pattern Implementation plan for the given abstracted infrastructure. This plan is used to feed the Application Optimiser, which aligns the application with the chosen abstracted infrastructure and takes into account its performance and other non-functional requirements.	Deployment Preparation
ER#6	4	Application Optimiser	This component takes into account the infrastructure, as provided by the application developer and the operator. The purpose is to statically optimise the application, its deployment and configuration, to maximize its performance.	Application Optimiser
ER#7	3	Infrastructure Operator Editor	This corresponds to the SODALITE infrastructure modelling editor, used to compose and describe infrastructures by means of the Abstraction Components Library. The outcome is then used by both Application Optimiser and Deployment Improvement. It also provides dashboards, showing the infrastructure and application state for already deployed applications (monitoring, deployment, operational analytics, and reconfiguration data).	SODALITE IDE
ER#8	5	Deployment Improvement	This component uses the static models and dynamic monitoring data to assess the current state of infrastructure and application. Based on this state it provides dynamic reconfiguration plans to improve the deployment. To this end, it includes the Predictive Deployment Refactoring component, which uses historical data (monitoring, infrastructure state), as well as Operational Analytics and Semantic decision support, to decide how and when	Deployment Refactorer, Refactoring Options Discoverer

			to do the re-deployment and what needs to be changed to improve performance. In short, this steers the application execution during its runtime to achieve better performance.	
ER#9	5	Runtime	This is the central component that takes care of the deployment of the application and of its possible re-deployments, based on outcome from Deployment Improvement component. It also provides the lowest level monitoring and alerting to other components.	Orchestrator and Monitoring

4. Technical KPIs

The following table reports the technical KPIs defined in the Grant Agreement and provides additional information concerning their interpretation, the measurement approach, the involved components and the deadline by which the target lower bound is planned to be achieved. We do not consider in this analysis KPI 6.1 (Uptake of developed work in

the use-cases.) and 6.2 (Dissemination of developed work using the use-cases) as they are not directly related to the technical aspects of the project.

Id and description	Specific meaning	Target	Involved component	Deadline
KPI 1.1: Abstraction of application and infrastructure.		Lower bound is 25% coverage of all application and infrastructure scenarios in the scope of SODALITE case-studies	SODALITE Modeling Layer	M24
KPI 1.2: Abstraction of Infrastructure Performance Patterns.		Lower bound is 80% of all performance patterns found in HPC and Cloud infrastructures	SODALITE Modeling Layer	M33
KPI 1.3: Abstraction of execution constraints and possibilities.		Lower bound is coverage of 80% of execution scenarios	SODALITE Modeling Layer	M33
KPI 2.1: Increase of abstracted application performance on abstracted infrastructure by using Infrastructure performance abstraction patterns	When AOE exploit abstractions in their application code, we expect an increase in performance of 15%	Application performance increased by 15%. The performance metric to be used will depend on the specific case study.	Case studies, SODALITE Modeling Layer, Infrastructure as Code Layer, Runtime, and Case studies	M30
KPI 2.2: Increase of concretized (deployed) application performance running on targeted infrastructure through		Lower bound target is 20% improvement over the baseline	SODALITE Infrastructure as Code Layer, SODALITE Runtime	M30

Predictive Deployment Refactoring.				
KPI 3.1: Reduction in software and/or application development time and cost.	The focus will be on development of deployment descriptions, not application code.	Lower bound target is 10% improvement over the baseline and will be evaluated through external parties where possible. The improvement will be measured by considering the time needed to develop an application manually and then with SODALITE.	SODALITE Modeling Layer	M24
KPI 3.2: Reduction in software management (redeployment, reconfiguration) time and cost.	This reduction is specifically concerning resource management.	Lower bound target is 30% improvement over the baseline and will be evaluated through external parties where possible. The improvement will be measured by changing the way we re-deploy the app.	SODALITE Runtime	M24
KPI 4.1: Component compatibility	Integration of the SODALITE system allows for combined use of all its components.	The target is 95% of SODALITE component compatibility	All components	M33
KPI 5.1: Open source release		Minimum 80% of code released under open-source license	All components	M36
KPI 5.1: Extension of existing projects		Minimum 60% of code extending the existing projects, to be upstreamed	All components	M36

5. Preliminary Evaluation Plan

The main objective of the evaluation plan will be to verify the achievement of the KPIs presented in Section 4. The SODALITE case studies will be exploited in this assessment. Objective metrics such as performance improvement or reduction in required effort will be complemented by subjective feedback from the case study owners. Where possible, external stakeholders will be involved as well, in particular, for KPIs 3.1 and 3.2.

As described in Section 4, the fulfillment of KPIs will be assessed starting from M24 on. In the previous iterations of the project, other testing and validation activities will be performed. More specifically, we will test the UML use cases defined in Section 2 in the context of some of the SODALITE case studies. The Table below shows the planned coverage of UML use cases by case studies and when the intermediate and final versions of the use cases will be ready to be tested by case study owners. The table also includes a column for the testbed providers. They, in fact, will play the role of Resource Expert and Quality Expert. In these two roles, they will provide inputs to all three case studies, in particular, concerning the characteristics of the specific resources they will make available for the purpose of the experimentation.

Additional details concerning the evaluation plan will be made available in the forthcoming releases of this deliverable.

Use Case	**Virtual clinical trial**	**SNOW**	**Vehicle IoT**	**Testbed providers**	**Released at**
UC1 Define Application Deployment Model (WP3)	X	X	X		M12, M18, M24
UC2 Select Resources (WP3)	X				M12, M18, M24
UC3 Generate IaC code (WP4)	X	X	X		M12, M18, M24
UC4 Verify IaC (WP4)	X	X	X		M12, M18, M24
UC5 Predict and Correct Bugs (WP4)	X				M12, M18, M24
UC6 Execute Provisioning, Deployment and Configuration (WP5)	X	X	X		M12, M18, M24, M33
UC7 Start Application (WP5)	X	X	X		M12, M18, M24, M33
UC8 Monitor Runtime (WP5)	X	X	X		M12, M18, M24, M33
UC9 Identify Refactoring Options (WP5)	X	X	X		M18, M24, M30, M33
UC10 Execute Partial Redeployment (WP5)	X	X	X		M18, M24, M33
UC11 Define IaC Bugs Taxonomy (WP4)				X	M12, M18
UC12 Map Resources and Optimisations (WP3)	X	X	X	X	M12, M24

UC13 Model Resources (WP3)		X		X	M12, M22, M30
UC14 Estimate Quality Characteristics of Applications and Workload (WP3)			X		M18, M24, M33
UC15 Statically Optimize Application and Deployment (WP4)	X	X	X		M18, M30
UC16 Build Runtime images (WP4)	X	X	X		M12, M18, M24

References

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

[2] OMG, Unified Modeling Language. Formal Document. Version 2.5.1, December 2017.

[3] ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes - Requirements engineering," in ISO/IEC/IEEE 29148:2018(E) , vol., no., pp.1-104, 30 Nov. 2018, doi: 10.1109/IEEESTD.2018.8559686.

[4] Michael Jackson and Pamela Zave. 1995. Deriving specifications from requirements: an example. In Proceedings of the 17th international conference on Software engineering (ICSE '95). ACM, New York, NY, USA, 15-24. DOI=http://dx.doi.org/10.1145/225014.225016.

[5] XLAB, SOftware Defined AppLication Infrastructures managemenT and Engineering (SODALITE) - Grant agreement number: 825480, AMENDMENT Reference No AMD-825480-4, 2019.