# Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

# D2.3

## Requirements, KPIs, evaluation plan and architecture - final version

**Polimi**

31/07/2021

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

| Deliverable data | | | |
|---|---|---|---|
| **Deliverable** | Requirements, KPIs, evaluation plan and architecture - final version | | |
| **Authors** | Luciano Baresi (POLIMI), Elisabetta Di Nitto (POLIMI), Giovanni Quattrocchi (POLIMI), Dragan Radolović (XLAB), Alexander Maslennikov (XLAB), Alfio Lazzaro (HPE), Jesús Gorroñogoitia (Atos), Jesús Ramos Rivas (Atos), Kamil Tokmakov (USTUTT), Steven Presser (USTUTT), Zoe Vasileiou (CERTH), Nikolaidis Efstathios (CERTH), Dourvas Nikolaos (CERTH), Kalman Meth (IBM), Paul Mundt (ADPT), Indika Weerasingha Dewage (JADS) | | |
| **Reviewers** | Nejc Bat (XLAB), Kalman Meth (IBM) | | |
| **Dissemination level** | Public | | |
| **History of changes** | Name | Change | Date |
| | v0.9 | complete release ready for internal review | 28.7.2021 |
| | v1.0 | finalized for submission | 29.7.2021 |
| | | | |

## Executive Summary

This deliverable is the continuation and the last iteration of deliverables D2.1 and D2.2. It provides the final description of the requirements defined by the consortium and of their final status. It also supplies the final version of the architecture of the SODALITE framework, and the conclusive presentation of KPIs and their evaluation process.

As for requirements, it summarises the requirements completed and deleted over the first two years, those completed, or that will be completed, in the third year, and the deviations we have identified. It refines the architecture of the proposed framework with the novel aspects emerged in the last year and a special emphasis on the security aspects, which are now better integrated with the different use cases. The architecture described here complies with the release of the SODALITE environment at month 30, that is, Milestone MS7 (*Final Architecture*). The work on our KPIs condenses what we did in the past two years, lists all KPIs, updates the evaluation plan, and focuses mainly on the automated assessment of the security vulnerabilities embedded in delivered artifacts.

## Glossary

This section provides a reference for the main terms used in this document. Most of the terms are defined the first time they are used in the document, but their definition is also reported here for the sake of simplicity and speed. Reported terms are classified under seven main categories.

**Acronyms**

| | |
|---|---|
| **AADM** | Abstract Application Deployment Model |
| **AAI** | Authentication and Authorization Infrastructure |
| **ADM** | Application Deployment Model |
| **AM** | Ansible Model |
| **AOE** | Application Ops Expert |
| **CSAR** | Cloud Service Archive |
| **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 |
| **IAM** | Identity and Access Management |
| **JWT** | JSON Web Token |
| **KB** | Semantic Knowledge Base |
| **KPI** | Key Performance Indicator |
| **LRE** | Lightweight Runtime Environment |
| **MOM** | Message Oriented Middleware |
| **NFR** | Non Functional Requirement |
| **OASIS** | Organization for the Advancement of Structured Information Standards |
| **OM** | Optimization Model |
| **OWL** | Web Ontology Language |
| **PBS** | Portable Batch System |
| **QE** | Quality Expert |

| | |
|---|---|
| **RDF** | Resource Description Framework |
| **RE** | Resource Expert |
| **RM** | Resource Model |
| **SD** | SODALITE Design-time |
| **SR** | SODALITE Runtime |
| **Torque** | Terascale Open-source Resource and QUEue Manager |
| **TOSCA** | Topology and Orchestration Specification for Cloud Applications |
| **VPN** | Virtual Private Network |

**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 targeted states of the infrastructure/application 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 with 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). |
| **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 environment. These last ones are also called SODALITE case studies. |

**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 making 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 only the tasks to be executed to the cloud provider, which 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. |
| **Keycloak** | Open source Identity and Access management |
| **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. |
| **MODAK** | The SODALITE Application Optimizer, MODAK, offers a framework that generates the scripts to be executed in an HPC environment to achieve an optimized execution of application components. |
| **OpenStack** | An open source cloud operating system. |
| **OpenFaaS** | 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. |
| **Vault** | A tool that helps secure, store and tightly control access to tokens, passwords, certificates, encryption keys for protecting secrets and other sensitive data. |

**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 OWL2 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

## 1.1 Context

Uniform, standardized infrastructures in computing centres are increasingly provided as services. Thanks to them, applications in multiple domains, including Industry 4.0, smart environments and many others are nowadays easier to manage. There are, however, many other applications that need to adopt ad-hoc and optimized infrastructures to efficiently execute specific categories of jobs or components. For example, consider a web application organized according to a microservice architecture that, among the other features, runs an AI inference algorithm, for instance, to recognize specific objects within some images or to identify the products that a certain user will, likely, prefer. This application would benefit from an heterogeneous setting as the microservices and web server could find their optimal configuration on the cloud, while at least part of the inference algorithm or at least its training phase could run more effectively on an HPC cluster based, for instance, on GPUs. On the one hand, such a configuration can bring several advantages in terms of efficient use of the available resources and effective execution of the system. On the other hand, being able to effectively deploy and operate application components in an heterogeneous environment is not easy and today requires an in-depth knowledge of each target infrastructure, of the execution models each of them support, and of the mechanisms that can be exploited to efficiently enable information exchange between the application parts deployed on different types of resources.

With the aim of simplifying the adoption of heterogeneous infrastructures ensuring the possibility to fine-tune performance, the SODALITE framework offers modeling and runtime features to simplify the creation, the deployment and operation of complex applications that require the adoption of heterogeneous computational environments.

## 1.2 SODALITE goals

The main project goals can be summarized as follows:

- *Allow the SODALITE user to create deployment models in a simple and smart way*. The SODALITE IDE is coupled with an ontology-based reasoning engine that guides the user in the definition of the deployment model, providing context-dependent suggestions, e.g., on missing dependencies and properties to be specified for a certain component, given its type, and assigning proper default and resource-dependent values to the pieces of information that are left underspecified by the user.
- *Support design-time optimization of applications, especially in the case they adopt HPC resources*. To exploit HPC resources in the best possible way, the application code may need to be tuned and/or scaling actions may need to be executed (e.g., increasing the number of cores, accelerating with GPUs or coprocessors, enabling faster storage, etc.). Such actions must be tailored considering the type of application components to be deployed, their QoS requirements and the available resources. The SODALITE Application Optimizer, MODAK, focuses on these issues and offers a framework that, given the specification of a few constraints as part of a deployment model, is able to generate the scripts to be executed in an HPC environment to achieve an optimized execution of application components.
- *Support resource experts in modelling their resources and in automating the process of discovering new resources* and deriving suitable models for them.
- *Support the identification of bug smells in deployment models and of possible reconfiguration options* of running application configurations. Thanks to machine learning, SODALITE analyses the previous history of deployment models that had to be corrected to identify bug smells, thus building a taxonomy of bug smells that is then used to provide suggestions to DevOps experts. A similar learning mechanism is also used at runtime to

identify possible configurations that perform better than others and suggest them to DevOps experts when the monitoring system reveals the presence of problems in the current configuration.

- *Offer light-weight execution environments* that are essentially cross-platform containers that enable the possibility to execute, with different performance, the same application components on heterogeneous resources in a seamless way and allow them to be built automatically.
- *Incorporate monitoring configuration within deployment models* to automate the execution of the monitoring infrastructure and to collect monitoring data from multiple and heterogeneous platforms.
- *Support runtime optimisation of applications* by dynamically scaling in and out computational resources depending on the specific applications being considered.
- *Offer suitable mechanisms to support data placement-aware deployment and data movement between HPC, Cloud and edge resources*. Data placement and movement across memory, storage or across infrastructures is important for application performance. SODALITE aims to optimise data movement at two different levels: single components and compositions of multiple components. Many components use accelerators like GPUs to improve their performance. One of the major bottlenecks in getting efficient performance from GPUs is the data movement across the host CPU and the device GPU. CPUs and GPUs have dedicated memory and the data is usually moved to the GPU for computations and the results are then copied back. In the context of application optimization, we explore asynchronous data transfer and prefetching as a way to address this issue. For what concerns the composition of multiple components, these usually communicate by reading and writing data in a persistent memory like files. If the applications are deployed across different targets, then the data is communicated over the network. In SODALITE we explore the usage of efficient data movement across storage and network to improve the workflow performance.
- *Offer Advanced orchestration features*, including the possibility to reconfigure part of a the infrastructure or the deployed application, the parallelization of deployment execution to make it faster, the possibility to restart or resume a failed deployment from the point of failure as well as the definition of a well-designed REST API and support for the newly introduced orchestration features.
- *Enable secure and privacy-aware operation of the infrastructure***.** Providing proper identity and access management is a crucial part of protecting both user data and sensible project information. There are two different facets we will consider in the scope of SODALITE. The first one concerns the mechanisms that control access to the SODALITE platform itself. This is covered by a role-based Identity and Access Management (IAM) implementation (Keycloak) for SODALITE users and other implementations for secret and credential management (e.g., Vault or similar). The second aspect concerns the possibility to model, as part of the resources made available to the DevOps teams and suitable to support application deployment, those dedicated middleware components, such as VPNs, to deploy and operate applications properly.

## 1.3 Goals of this document

This specific document concludes the series of documents (D2.1 and D2.2) that provide an overview on the SODALITE approach by defining its requirements, its software architecture, and evaluation plan. In particular, this document reports about the implementation status of all SODALITE requirements, extends and ameliorates the overall architecture of the SODALITE framework, and reexamines the evaluation plan, which aims at: 1) guaranteeing that the SODALITE framework addresses the research and technical challenges that have been defined during the project conception, and 2) ensuring a good level of quality of the SODALITE code also from the

security viewpoint. The work presented in this document aims to finalize all these aspects and provide the consortium with a solid revision of all these concepts, which are being finalized in the last phase of the project.

## 1.4 Work performed from the beginning of the project

As already mentioned, this deliverable is the final result of a set of tasks aiming at:

- Enabling all partners to achieve a shared and overall understanding of the challenges to be addressed in the project - this work has led to identification of the main stakeholders for the SODALITE platform and to the definition of several use cases and requirements that have driven all SODALITE development activities. More specifically, with the involvement and collaboration of all partners, we identified three main stakeholders, 16 UML use cases at the beginning of the project and another two use cases in the following years. The analysis of these use cases has generated the definition of 82 requirements in the first project year, to which another 21 have been added in the second project year. In that same year, 8 requirements have been discarded, while one has been discarded in the third year. These numbers are summarized in the table below.

| | M6 | M24 | M30 | Total |
|---|---|---|---|---|
| SODALITE users | 3 | 0 | 0 | 3 |
| UML Use cases | 16 | +1 | +1 | 18 |
| Requirements | 82 | +21 -8 | -2 | 93 |

- Defining the architecture of the SODALITE framework in terms of all its subcomponents and the interaction among them - the resulting architecture has been the element that has enabled the parallel development of multiple components, a continuous integration approach, as well as the possibility to keep the development process under control. The SODALITE architecture, in its final version, includes three main layers (modelling, infrastructure as code, and runtime) supported by a security pillar and is composed of 15 main components. The architectural analysis has required the development of 4 component diagrams focusing on multiple perspectives and 21 sequence diagrams. As mentioned, the definition of the architecture has enabled the proper configuration of our CI/CD environment, which includes the following tools: Jenkins for automating the execution of tests and the creation of dockerised components, DockerHub for storing our components, SonarCloud and Snyk for code analysis.
- Developing a proper evaluation plan that has taken its roots from the KPIs defined in the grant agreement and has applied them in the SODALITE case studies; the evaluation plan has then been extended to include also the procedures and automated tools aiming at keeping the quality of the SODALITE code under control. The assessment of the KPIs has required the definition of a measurement process that also includes some experiments with external users.

While the three types of results achieved by this work package are not directly exploitable, they have been the basis for the understanding, development and assessment of all SODALITE results. The work conducted within WP2 and reported in this deliverable has been disseminated through some presentations and the writing of two papers that provide an overview of the SODALITE framework (the reader can refer to the dissemination deliverable for more information). Additionally, it is driving the writing of a book that provides a synthesis of the SODALITE results.

## 1.5 Progress beyond the state of the art and potential impact

This deliverable is contributing to the progress beyond the state of the art by offering a complete consolidation of the SODALITE requirements and of the corresponding architecture. The SODALITE framework resulting from this implementation represents a novel result in the literature because it simplifies the deployment and operation of complex applications on multiple heterogeneous infrastructures. While there are multiple tools and approaches that support such activities in a purely cloud-based context, the novelty concerns the possibility to target heterogeneous resources. Additionally, the project offers a number of specific innovations that range from the smart editing features offered by the IDE and the supporting ontology to the optimization features offered by MODAK, the dynamic discovery and automated generation of resource models, the verification approach based on the identification of code smells, the refactoring, etc.

As already discussed in D2.1, in the third project year we are undertaking the third development iteration to consolidate the SODALITE platform. Among the other aspects, we are focusing on the mechanisms for data movement, on the integration of the runtime optimisation and reconfiguration mechanisms, and on the usage of FaaS resources especially for what concerns the edge part.

In order to increase the impact of the SODALITE framework, we are paying particular attention to the quality of code and its robustness from the security perspective. Moreover, we are continuing the integration and testing iterations and have decided to have monthly testing of the SODALITE platform by the case study owners. Finally, we are planning the execution of experiments with external end users and the release of part of the SODALITE platform as a service in a sand-box environment.

## 1.6 Relationships with other WPs

This deliverable is one of the cornerstones of the whole project. It defines the requirements and architecture on which the whole SODALITE framework is based. Moreover, it describes the evaluation plan adopted in the project to assess the artifacts developed and the results obtained. As such, this document is supposed to guide the work of both the technical work packages (WP3-WP5) and of the work package in charge of the demonstrators and of the evaluation (WP6).

The evaluation plan described in this deliverable and in its previous versions is executed in WP6 and the results that will be achieved will be reported in deliverable D6.4 (*Final implementation and evaluation of the SODALITE platform and use cases*), which will show the extent to which the SODALITE framework addresses the needs by the case study owners and fulfills the KPIs defined at the beginning of the project.

## 1.7 Structure of the document

This last iteration of the set of documents on requirements, architecture, and KPIs is structured in a way similar to the previous deliverables D2.1 and D2.2:

- Section 2 focuses on requirements. We organize them in several different groups: those fully met at the end of year two, those met at the current M30, those that will be met by the end of the project, and those that have been discarded, together with the motivations for this choice.
- Section 3 provides a coherent and updated view of the architecture. This section gives the final picture of what components are part of the SODALITE offer and how they cooperate to meet the different goals. We want to give a clear definition of the architecture to continue the development, and we have also revised and ameliorated the integration between the way SODALITE deals with authentication and authorization and the other components. This means that we have restructured the architecture, and its description, thoroughly to better explain the particular components and their role in the different scenarios.

- Section 4 provides a short summary of the evaluation plan that has been described in further details in D2.2. Moreover, it highlights the work done on assessing the security issues of delivered artifacts ---by means of automated analysis--- with the objective to deliver a more robust infrastructure.
- Finally Section 5 draws the conclusions.

# 2 Final status of requirement assessment

This section lists all the requirements the project has defined from the beginning and provides a clear summary of whether and how we implemented them. The presentation touches: the new use case defined in the third year of the project, the requirements we implemented, those that were cancelled, and those that were subject of major deviations.

## 2.1 Identified Use Cases

This section proposes the whole set of actors and use cases identified by the consortium. We originally identified 3 actors and 16 use cases after the first requirements elicitation phase, and we identified two new additional use cases in the second and third iteration, respectively. The new use case UC 17, related to resource discovery, was already presented in deliverable D2.2, while the new use case, UC 18, concerns the possibility for our main user to oversee the life-cycle of an application after deployment and is detailed in Section 2.1.3.

### 2.1.1 Actors

Identified actors, as presented in deliverable D2.1 are the following. We also try provide a mapping onto the roles foreseen in ISO/IEC/IEEE standard 12207 Systems and software engineering — Software life cycle processes:

| Actor | Brief description |
|---|---|
| **Application Ops Expert (AOE)** | S/he is in charge of operating the application and of all the aspects that refer to the deployment, execution, optimization and monitoring of the application. This role can correspond to the ISO/IEC/IEEE role in charge of Operation processes and maintenance processes as they focus on the day-by-day operation. |
| **Resource Expert (RE)** | S/he is in charge of dealing with the different resources required to deploy and execute the application. This role can correspond to IEEE roles in charge of Infrastructure management and Configuration management processes, given they are supposed to allocate and manage resources and configurations. |
| **Quality Expert (QE)** | S/he is in charge of the quality of service both provided by the execution infrastructure and required by the executing application. This role can correspond to IEEE roles in charge of Quality Management and Quality assurance processes because they oversee the overall quality of deployed applications and thus of the project itself. |

### 2.1.2 Use cases up to Year Two

The use cases already presented in previous deliverables are:

| ID | Title | Reference WP |
|---|---|---|
| UC1 | Define Application Deployment Model | WP3 |
| UC2 | Select Resources | WP3 |
| UC3 | Generate IaC code | WP4 |
| UC4 | Verify IaC | WP4 |
| UC5 | Predict and Correct Bugs | WP4 |
| UC6 | Execute Provisioning, Deployment and Configuration | WP5 |
| UC7 | Start Application | WP5 |
| UC8 | Monitor Runtime | WP5 |
| UC9 | Identify Refactoring Options | WP5 |
| UC10 | Execute Partial Redeployment | WP5 |
| UC11 | Define IaC Bugs Taxonomy | WP4 |
| UC12 | Map Resources and Optimisations | WP3 |
| UC13 | Model Resources | WP3 |
| UC14 | Estimate Quality Characteristics of Applications and Workload | WP3 |
| UC15 | Statically Optimise Application and Deployment | WP4 |
| UC16 | Build Runtime Images | WP4 |
| UC17 | Platform Resource Discovery | WP4 |

### 2.1.3 New Use Case

This section presents the new UC we added in the third year. This UC enables an AoE to browse his deployments and manage their life-cycle. This UC is associated with the following requirements expressed by UCs: UC8.R1, UC8.R8, Y2_R6.

**UC18: Deployment Governance**

| | |
|---|---|
| **Actors:** | Application Ops Expert (AoE) |
| **Entry condition:** | An IDE-logged AoE needs to manage her application deployments |

| | |
|---|---|
| **Flow of events:** | • AoE browses her application deployments in a dedicated Deployment Governance view of the IDE. This view shows every application (i.e. blueprint in runtime orchestrator terminology) and their deployments.<br>• AoE navigates the tree of her application and deployments and selects one.<br>• The browser shows details of selection.<br>• AoE can open in a Web browser the monitoring dashboards associated with a selected deployment, and browse their monitoring metrics.<br>• AoE can request two possible actions on selection: resume or delete.<br>  ○ AoE may request to resume a deployment if a failed one is selected. Resume can be done either from an initial state or from the first failing node.<br>  ○ AoE can request to delete a deployment. Blueprints not containing deployments can be deleted as well.<br>• AoE can refresh the list of application deployments. |
| **Exit condition:** | AoE ends the management of her application deployments. |
| **Exceptions:** | Runtime orchestrator endpoint could not be correctly configured in IDE. Orchestrator could not be accessible (VPN not set but required, orchestrator down). |

## 2.2 Completed requirements

### 2.2.1 Up to the second year

These are the requirements that were fully implemented by the end of the second year of the project (month 24).

| ID | Brief description | Notes |
|---|---|---|
| **UC1.R1** | The SODALITE Design-time environment requires an API to the application/Infrastructure abstract pattern repository | Application and infrastructure abstract patterns can be stored/retrieved to/from the repository |
| **UC1.R2** | DSL: specification of application patterns and models | All modeling needs of the demonstrating use case have been addressed |
| **UC1.R3** | Authoring of application abstract models (part of abstract tuple) | AADM/RM for all the demonstrating user cases are available |
| **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- |
| **UC2.R4** | SLURM/Torque modelling | Supported |
| **UC2.R5** | OpenStack modelling | Supported |

| | | |
|---|---|---|
| **UC2.R6** | Use context-aware search and discovery, matchmaking and reuse of cloud applications and infrastructures | Supported |
| **UC3.R1** | SODALITE Runtime (SR) should support Ansible playbooks and TOSCA node definitions for application deployment in public cloud | We support EGI and AWS. |
| **UC3.R2** | SR should support Ansible playbooks and TOSCA node definitions for application deployment in HPC environment | Both TORQUE and SLURM deployments are supported |
| **UC3.R5** | Support for SODALITE DSL | Supported |
| **UC3.R6** | Generation of correct, complete and deployable IaC artifacts | Supported |
| **UC3.R7** | Generation of IaC which exploits heterogeneous architectures | Supported |
| **UC5.R2** | Predict and Correct Security and Privacy Defects in IaC Artifacts | Merged with a UC4 requirement. Detection of common security/privacy smells in TOSCA and Ansible are supported |
| **UC6.R3** | SR should support Ansible playbooks and TOSCA node definitions for application deployment in private cloud | Private Openstack cloud supported |
| **UC6.R4** | SR plugin supporting Docker Compose | SR does not support docker compose but, instead, enables modelling and execution of docker hosts, registries, volumes, networks, containers through TOSCA topology models |
| **UC7.R2** | Smart application scheduling | Application requests are efficiently scheduled by the Node Manager on fast GPUs or CPUs according to application needs (e.g., SLA, current workload) |
| **UC9.R2** | Model Design (Adaptation) Choices | The adaptation choices can be modeled using the feature modeling and policy based adaptation language |
| **UC10.R3** | Vertical Resource Scalability | Node Manager is able to vertically scale resources at runtime according to applications' needs |
| **UC13.R1** | Docker Modelling | Container runtime is supported |
| **UC13.R4** | Ontology Serialization | The semantic model is compliant with OWL2 language |

| | | |
|---|---|---|
| **UC16.R1** | Lightweight application base images | Application Ops Expert defines the base image from which to build the image |
| **UC17.R4** | The existing application designs (or components) and infrastructure should be able to be dynamically discovered and used when optimizing the application | The optimization of application deployment due to changes on the infrastructure or other NFRs is based on the option discovery and reconfiguration components for the currently deployed AADM. |
| **Y2_R9** | SODALITE should support the runtime discovery of components | Platform Discovery Service is currently able to discover and prepare the node definition templates for target infrastructures and store the Resource Models (RM) into the semantic knowledge base. It also supports updating the RMs when triggered by monitoring or by any other component in the pipeline. |

### 2.2.2 Completed by Month 30

These are the requirements that were fully implemented by the deadline of this deliverable (month 30).

| ID | Brief description | Notes |
|---|---|---|
| **UC3.R3** | SR should support Ansible playbooks and TOSCA node definitions for application deployment on edge | Fully implemented. |
| **UC3.R4** | SR should support Ansible playbooks and TOSCA node definitions for application deployment in fog | Fully implemented. |
| **UC8.R2** | Collect network metrics | Additional metrics have been added. |
| **UC8.R3** | Collect host metrics (CPU, memory) | Supported |
| **UC8.R5** | Monitoring levels | The API to allow application developers to hook their exporters to the monitoring ecosystem has been developed. |
| **UC8.R9** | Absorb Skydive metrics | Skydive Metrics for networks are collected. |
| **UC9.R10** | Static Provisioning of Heterogeneous Resources | The enforcement of TOSCA policies on scalability has not been developed because scalability is handled by the node manager and refactorer |
| **UC9.R12** | TOSCA inputs to SR (SODALITE Runtime) | The reconfiguration component has been integrated with the orchestrator and with the |

| | | |
|---|---|---|
| | | knowledge base |
| **UC9.R13** | Dynamic Policy-based restrictions on resource access from the Edge | AlertManager integrated into the Edge. |
| **UC10.R1** | Create and Maintain Runtime Models | We have improved the runtime model to support the AADM at runtime. Validated with Snow and Vehicle IoTrefactoring scenarios |
| **UC10.R2** | Horizontal Resource Scalability | Implementing TOSCA policies and triggers would have meant duplicating the work of the refactoring component. This feature is already covered by the functionality of option discovery and reconfiguration at runtime |
| **UC13.R2** | Kubernetes Modelling | Kubernetes Helm, Cluster and Node can be modelled. Also Kubernetes objects and manifests are modeled in AADM/RM |
| **UC13.R5** | TOSCA Compliance | Every DSL specification is TOSCA compliant. The blueprints we generate are 100% TOSCA compliant. We do not support the whole TOSCA specification as there are some aspects that are not needed in our cases and are not supported by our orchestrator |
| **UC13.R6** | Authoring of infrastructure abstract models (part of abstract tuple) | KB-driven content assistance for RM has been implemented. RM DSL has been extended with support for policy definitions |
| **UC13.R7** | IaaS Modelling | All needed infrastructures have been modeled. Some have been automatically discovered |
| **UC14.R1** | Estimate Performance of Designs | Performance design for the latency test cases AI (SnowUC) and MPI (ClinicalUC) benchmarks estimated. |
| **UC15.R1** | Delivery of optimized application | Optimized containers for the TensorFlow training (SnowUC) and the linear solver used in CodeAster (ClinicalUC) provided. |
| **UC15.R2** | Optimize Application and Deployment | We cover optimization for AI (SnowUC) and MPI (ClinicalUC). Also, we show that the improvement on inference, which could be used in VehicleIoT, is marginal |
| **Y2_R5** | SODALITE should support modeling of HPC workflows | Generalised Node types were provided, such that it is now possible to model HPC job workflows in an abstract way without targeting a specific HPC resource manager.<br>Furthermore, the support of parallel deployment |

| | | |
|---|---|---|
| | | execution implemented in SODALITE orchestrator allows one to execute independent HPC jobs in parallel |
| **Y2_R7** | SODALITE should provide easier integration of components by providing guidance and suggestion of integration points for a particular component type | The SODALITE IDE provides suggestions depending on the requirements of a certain component |
| **Y2_R13** | SODALITE should automate the definition of optimization options for a target infrastructure | The benchmarks and calculation of the infrastructure performance model is automated and containerized, and can be run on either PBS or Slurm |
| **Y2_R14** | SODALITE should support the validation/monitoring/alerting that the modelled/discovered components are working properly | Runtime behaviour of deployed application components in target infrastructures is observed by monitoring. AOEs can define rules for detecting anomalous behaviour. Upon detection, alerts are triggered and caught by Refactoring that applies corrective actions |
| **Y2_R16** | SODALITE should support GDPR awareness and compliance | We have implemented a policy-based decision-making and enforcement mechanism in the IaC verification. This verification is triggered at runtime. |

## 2.3 Requirements foreseen to be completed by month 33

These requirements are those on which we are still working at the time of writing this document and that we plan to fully implement by the end of the development activities at month 33:

| ID | Brief description | Notes |
|---|---|---|
| **UC1.R6** | Description of application and standard build and run options | The description of build and run options is achieved through the specification of TOSCA interface operations using the Ansible DSL. The work is ongoing for what concerns the possibility to easily incorporate Ansible modules in the SODALITE modeling framework |
| **UC1.R7** | Support for microservice-oriented architecture | The deployment of microservice-oriented architecture developed artifacts will be supported by M33 |
| **UC1.R10** | Modeling language allowing modeling of all the necessary information to enable the generation of deployable IaC | The integration of the Ansible DSL with the knowledge base will be completed by M33 |

| | | |
|---|---|---|
| **UC2.R1** | DSL: specification of optimization patterns and models | Almost completed, few missing enhancements. The results of this will be presented in D3.4 |
| **UC2.R2** | Concretization of abstract models into deployment/configuration plans | The implementation is completed. It is to be validated onto concrete scenarios from use cases |
| **UC2.R3** | OpenFaas modeling for serverless computing actions | We have started addressing this requirement in the second project year. We still need to define some new OpenFaas container and experiment with it |
| **UC3.R8** | Reporting of errors in input models which prevent IaC generation | Basic support for misconfiguration detection added. We are working on integrating with UC4.R1 and UC4.R2 |
| **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.). | Monitoring rules are implemented as node types which are bound to infrastructural nodes and used to support refactoring and runtime management policies. |
| **UC3.R10** | Generation of IaC which exploits serverless computing artifacts (cloud functions) | We are extending the IaC builder and defining new nodes in the IDE and semantic reasoner |
| **UC3.R11** | Orchestrator input | We are working on making the orchestrator be able to deploy functions on OpenFaaS |
| **UC4.R1** | Verification of deployment descriptions for syntax and semantic errors | We are integrating the validation of IaC after the generation in UC3, which is then called from the IaC builder |
| **UC4.R2** | Verification of provisioning workflows derived from/specified in the deployment model descriptions | We need to implement the M2M transformation process for converting Ansible playbook workflows into Petri-net models and apply control flow checks |
| **UC5.R1** | Predict and Correct Performance Defects in Deployment Models | The performance anomaly detection needs to be integrated with the SODALITE runtime, and evaluated with case studies |
| **UC5.R3** | Build an Infrastructure Code Quality Framework | The IaC quality analysis tool from RADON will be integrated. The control flow metrics will be added |
| **UC6.R1** | SODALITE Runtime supporting various architectures | See UC2.R3 |
| **UC6.R2** | Support for extension plugins | See UC2.R3 |
| **UC6.R5** | Heterogeneous infrastructure | We are still working on OpenFaaS |
| **UC8.R1** | IDE Infrastructure dashboard | Thorough integration tests are still pending |

| | | |
|---|---|---|
| | (monitoring, deployment, reconfiguration) | |
| **UC8.R6** | Monitoring infrastructures | Integration with Vault is still pending |
| **UC8.R7** | End-to-end audit logging | Log-analysis is application specific. SODALITE monitoring, orchestrator, and refactorer provide the necessary information to implement it. |
| **UC8.R8** | Visualization of service deployment and adaptations | Thorough integration tests are still needed |
| **UC9.R1** | Model Control/Optimization Objectives (Performance, Privacy, and Security) | The need to specify information related to security and privacy is to be analysed. |
| **UC9.R3** | Find an Optimal Design Solution Considering Control Objective Tradeoffs | Need to experiment with case studies. |
| **UC9.R4** | Forecast Workload (Multi-class/tenant) | Need to experiment with case studies. |
| **UC9.R5** | Forecast Infrastructure Dynamics | Need to experiment with case studies. |
| **UC9.R6** | Predict Violations of Control Objectives (Performance, Security, and Privacy) | Need to experiment with case studies. |
| **UC9.R9** | Detect and Correct Defects at Runtime | Need to be integrated with the SODALITE runtime. |
| **UC9.R11** | Elastic Provisioning of Heterogeneous Resources | We need to work on the integration between Node Manager and Deployment Refactoring. |
| **UC11.R1** | Create a Taxonomy of Infrastructure Bugs and Resolutions | Need to validate the misconfiguration taxonomy with user surveys |
| **UC12.R1** | Select Optimisations for Application and Infrastructure targets | Few missing enhancements for the optimization DSL |
| **UC13.R8** | IaC deployment management Modelling | The final release for the IaC blueprint builder is planned for M33 |
| **UC13.R9** | Description of the available hardware | Platform Discovery Service already covers the intended targeted infrastructures by describing and storing the node definitions in the semantic knowledge base. We plan to additionally show to the user the definition of these nodes in the IDE to complete this requirement |
| **UC14.R2** | Estimate Security Level of Designs | Added a few more security and code smells for Ansible. We need to identify more security smells. |

| | | |
|---|---|---|
| **UC14.R3** | Estimate Privacy Level of Designs | Need to identify more privacy smells. |
| **UC14.R4** | Assess the Impact of a Design Choice | Need to experiment with use cases. |
| **UC17.R1** | OpenFaaS must be managed and utilized alongside other types of conventional infrastructure. | Still to be tested on FaaS functions |
| **UC17.R2** | Modelling must support SLURM/Torque for HPC. | We are investigating the possibility to add the definition of fast disks for HPC environments or schedulers. |
| **UC17.R3** | Modelling must support, besides container based deployments, also Bare Metal and VM abstractions such as OpenStack. | All needed models are available. We just need to compose a significant example. |
| **Y2_R1** | SODALITE should support alternative distributed deployment configurations for field testing | Alternative distributed deployment configurations will be possible thanks to the versioning support for AADMs. Variants (i.e versions) of an existing AADM can be authored, stored in the KB and deployed using the IaC and Runtime layers |
| **Y2_R2** | SODALITE should support the ability of running a reduced pipeline for debugging or test runs | This will be mostly implemented in the orchestrator/IaC builder. Probably, this can be done only in IDE/Reasoner. |
| **Y2_R3** | SODALITE should support a more general implementation of component connectors to cope with changing workloads | We support and tested in AADM/RM (by M30) the Nifi and data pipeline node types used in the demo (AWS S3 and GridFTP). The remaining connectors: S3-compatible storage (MinIO), Kafka, and the other connectors requested by use case owners will be developed, integrated and tested. |
| **Y2_R4** | SODALITE should support the adaptation of connectors when needed at runtime | This refers to the refactoring: when a redeployment of an application component happens, the connector should also be updated and adjusted such that the updated component would still be connected. We are testing it. |
| **Y2_R6** | SODALITE should support restarting a workflow from a failed component | Testing in concrete scenarios is still pending. |
| **Y2_R8** | SODALITE should support the discovery of heterogeneous resources at the Edge | Plan to support node-level annotations, additional accelerator types. |
| **Y2_R11** | SODALITE should support prebuilt optimized Singularity | More devices will be considered for experimentation purposes |

| | | |
|---|---|---|
| | containers configuration during the IaC generation so that the container can be executed on any infrastructure and can be configured for different optimization | |
| **Y2_R12** | SODALITE should support multi-arch container images. These images also include edge deployment; we could also have multiple variants of containers for different optimizations. | Implementation is currently in progress. |
| **Y2_R15** | SODALITE should support an extension of deployment refactoring for Cloud-to-Edge deployment. In particular, it should support deployment of microservices for hybrid multi-architecture clusters | Cloud-Edge hybrid application refactoring is supported and validated. Multi-architecture configurations are work in progress. |

## 2.4 Cancelled requirements

These are the requirements we stated originally and that then we decided to cancel. The main motivations behind the decision are summarized in the last column (Notes).

| ID | Brief description | Notes |
|---|---|---|
| **UC1.R5** | IntelliJ IDEA IDE extension | SODALITE IDE uses XText technology that supports the migration of the SODALITE AADM/RM textual editors to IntelliJ. However, the lack of resources prevents us from addressing this requirement without compromising others ranked higher. Besides, this support does not ease the development of the other IDE features for IntelliJ |
| **UC1.R8** | Abstractions and Mechanisms for Enforcing Performance, Security, and Privacy | The abstractions such as Load Balancer, Queue, Policy Enforcement Points are implemented by the case studies as necessary and are not part of SODALITE stack. |
| **UC1.R9** | Augment Application Models, IaC Models, and Infrastructure Models for Predicting Control Objectives | We use benchmarking for building performance models. In case of the performance modeling for deployment refactoring, the variants in the deployment models are models represented using the feature modeling ( a separate model). Thus, so far, there is no need for augmenting the IaC models. |
| **UC7.R1** | Lightweight open source | Application owners control and model their own |

| | | |
|---|---|---|
| | Message oriented middleware (MOM) for intra-service communication | inter-component information flow and middleware - this is an internal application design feature |
| **UC8.R4** | Monitor Overprovisioning (Performance), Security, and Privacy Metrics | Other monitoring requirements include this requirement |
| **UC9.R7** | Generate Application and Infrastructure Adaptation Plans | These requirements are not anymore relevant for the refactorer. The refactorer generates a new deployment model variant and sends it to xOpera, which does the actual adaptation. |
| **UC9.R8** | Enact Application and Infrastructure Adaptation Plans | These requirements are no longer relevant for the refactorer. The refactorer generates a new deployment model variant and sends it to xOpera, which does the actual adaptation. |
| **UC13.R3** | Istio Modelling | It can be handled as a Helm chart. |
| **Y2_R10** | SODALITE should support containers optimized for edge resources and should allow users to incorporate in these containers the needed elements (e.g., trained models) so that they can be shipped to the edge | This requirement comprises porting to edge devices primarily. Optimizations are only secondary and beyond the scope of the requirement. As such, it is out of the scope the project. |
| **Y2_R17** | SODALITE should support data privacy | Data privacy is managed by the different applications directly in different specific ways. |

# 3 Architecture

The SODALITE architecture has continued to evolve during the third year of the project. Over time, additional runtime environments have been supported, security has been improved, APIs have been refactored, and some new components have been introduced. The most recent essential updates are:

- Extended Security Pillar details;
- Revised interaction between components;
- Additional interfaces used by IDE (see figures for Modelling Layer and for IaC Layer).

These are reflected in the updated architecture figures and accompanying flow diagrams, in particular: UC6, UC7, UC8, UC11, UC13, and UC17. A new sequence diagram has been added for the new governance use case (UC18).

In order to clarify the role of the Security Pillar in the interaction with the other components, new sequence diagrams have been added in this part, which is now presented in Section 3.2. These new sequence diagrams help the reader understand how the off-the-shelf security components we have incorporated in the architecture work when they are invoked by the other SODALITE components.

The Sections 3.3 to 3.5 present the other component and sequence diagrams that define the SODALITE architecture. For the sake of completeness, also the parts that did not require any change compared to D2.2 have been reported again here.

## 3.1 General architecture

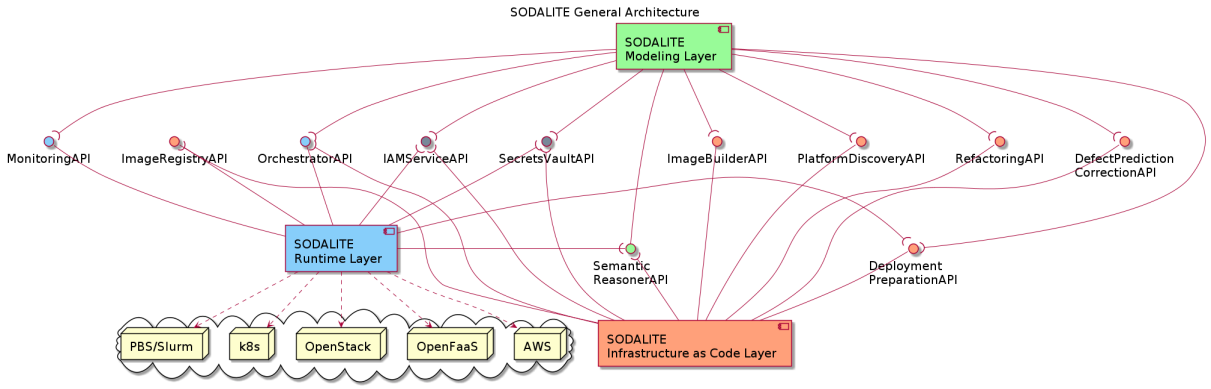



Figure 1 - SODALITE general architecture.

The SODALITE platform is divided into three main layers. These layers are the Modelling Layer, the Infrastructure as Code layer, and the Runtime layer. Figure 1 shows these layers together with their relationships defined in terms of offered and used interfaces. The Modelling 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 Modelling Layer the APIs for preparing the deployment, for platform discovery, 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 by the underlying technologies with particular emphasis on (but not limited to) the ones shown in the figure. The architecture is service based: each component exposes a REST interface to the others. Integration is achieved through the offered interfaces.

A more detailed view of the architecture is provided in Figure 2. This figure shows the three layers with their main sub-components, and how data flows from one component to another. The interaction of the sub-components is expanded upon in the following subsections.

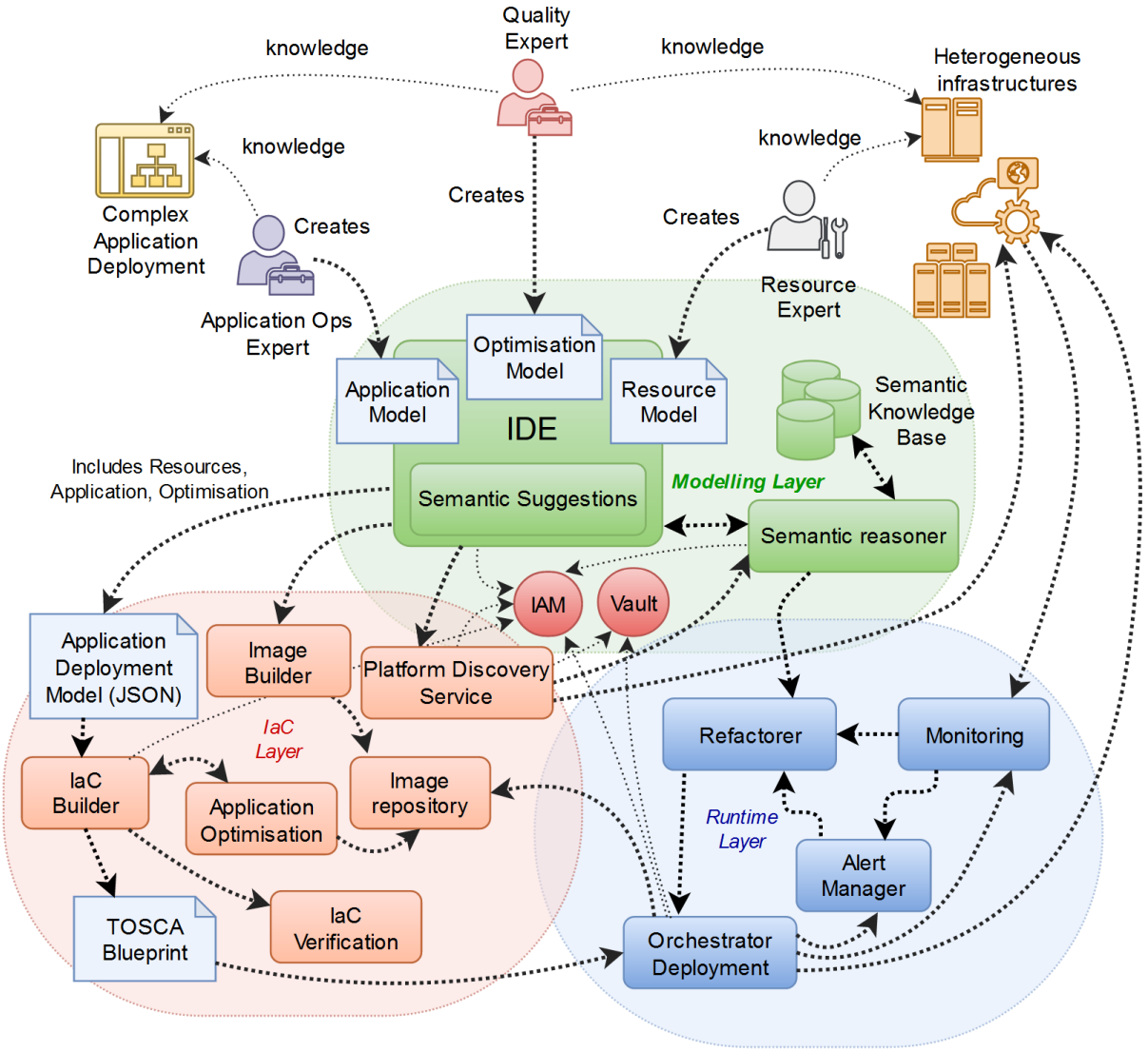



Figure 2 - Main components of the SODALITE architecture.

## 3.2 Security Pillar

SODALITE provides tools and methods to authenticate and authorize actions on API endpoints using open-source Identity Management and Secure Secret handling tools. While authorization is required - a single SODALITE endpoint can manage different infrastructures belonging to different domains. Apart from proper authentication and authorization of user actions, safe secret management across the whole deployment pipeline is also required and ensured by SODALITE.

The main changes in the security related part, compared with D2.2 are the following:

- Security scenarios are generalized for all Use Cases and explained in detail in the Security Pillar section of the document.
- Security schemes and interactions with security related components are removed from Use Case diagrams and references to the scenarios are added (for simplification and readability)

### 3.2.1 Security Pillar Toolkit

As a basis for authorization the OAuth 2.0 protocol was chosen, which is the de-facto industry standard for authorization. As for IAM provider, SODALITE uses *Keycloak*[1] - a popular and widely used open source tool which simplifies the creation of secure services with minimal coding for authentication and authorization. It allows wide customization of options exceeding the needs of SODALITE. Along with the basic authentication mechanism provided by Keycloak, SODALITE can also support such features as 2-factor authentication and seamless integration with third party identity providers like Google or GitHub. As a part of SODALITE stack, Keycloak is responsible for:

- Security configuration,
- Issuing a JSON Web Token (JWT),
- JSON Web Token validation.

Keycloak supports 2 standard mechanisms of token validation:

- Introspection endpoint,
- JSON Web Key Sets.

An Introspection mechanism (described in 4.5.4) provides a more secure way for validating tokens as the token can be revoked before expiration. It is up to component developers to choose the exact mechanism, but token introspection is encouraged to be used as the default one.

Apart from properly authorising user's actions, other concerns are also addressed by the Security Pillar - properly handling infrastructure secrets, like RSA keys, tokens, passwords. This involves 2 points to be addressed:

- Security of data in use,
- Security of data at rest.

The first point is mitigated by properly handling the secrets across the whole pipeline: not storing unencrypted information, no logging for security critical parts, proper user management on virtual containers that host SODALITE components. While SODALITE allows not storing any secrets at all and providing them in inputs, storing secrets in a vault allows to automate workflow and additionally ensure its safety. For addressing the second point e.g. security of data at rest *Hashicorp Vault*[2] was chosen, which is probably the most widely used open source tool for secret management.

Both Keycloak and Hashicorp Vault are deployed as a part of the SODALITE stack. Configuration of the components is done on the fly, so that basically these two components are ready to use after deployment. Admin credentials, roles, groups, clients, policies are created automatically, and additional configuration can be done via API calls or component Web UIs.

### 3.2.2 IAM and Secrets Vault configuration for new Project Domain

One of the problems Security Pillar is designed to solve is proper authorization of access to protected resources like models, blueprints, artifacts, secrets etc. A Project Domain entity was introduced for that purpose, serving as a relationship entity between users and protected resources. In IAM, Project Domain is designed as a set of roles, distributed across a set of groups, each group representing a user type (Application Ops Experts, Resource Experts and Quality Experts). Each role provides access to certain types of resources in a Project Domain, meaning that only a certain group of users should be granted access to these resources. For example, a Resource Expert would have the right to read and modify Resource Models in a certain Project Domain but not Abstract Application Models, whereas an Application Operation Expert would have a right to read Resource Models and modify Abstract Application Models. One user can belong to many

[1] https://www.keycloak.org/ Open Source Identity and Access Management for Modern Applications and Services

[2] https://www.vaultproject.io/ Secure, store and tightly control access to tokens, passwords, certificates, encryption keys for protecting secrets and other sensitive data using a UI, CLI, or HTTP API.

Project Domains and thus have any number of roles assigned to him. IAM roles are presented as private claims contained within JWT payload, signed by the IAM client private key.

Secrets Vault configuration of Project Domain requires creation of a Secret Store that contains secrets used in this Project Domain, Vault Policies regulating access to the Secret Store and Vault Roles that map Vault Policies to IAM roles.

For instance, Resource Expert role can be mapped into Secret write Policy and Application Operation Expert role into Secret read Policy. Then Resource Expert adds secrets to the storage and provides Application Operation Expert with a secret address that is later used in inputs for Application Model Deployment. With this configuration an Application Operation Expert need not deal directly with secret contents and a Resource Expert is able to seamlessly update secret values when needed.

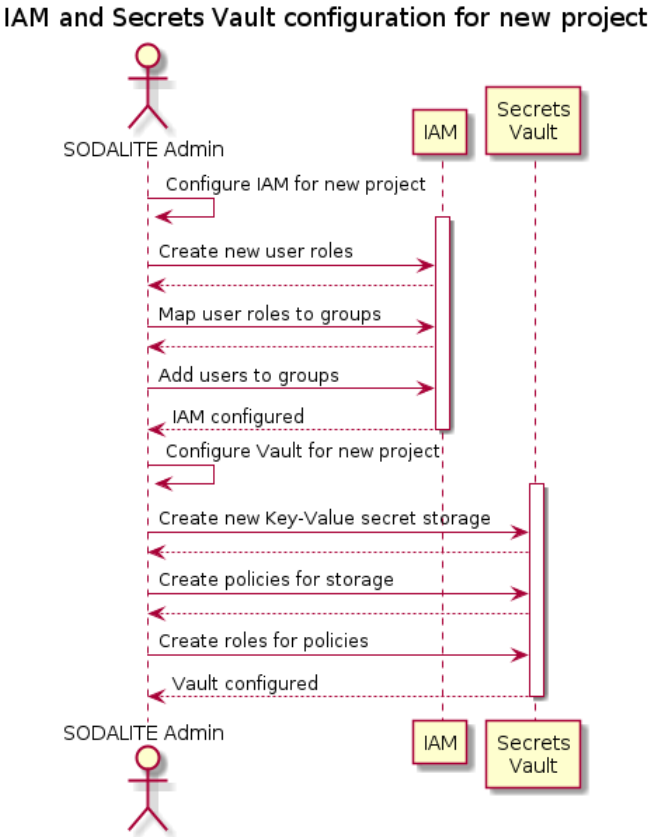



Figure 3 - Configuration of IAM and Secrets Vault.

Figure 3 describes configuration of IAM and Secrets Vault.

- A set of roles is created in IAM for a Project Domain (e.g. read Resource Models in Project Domain, write Resource Models in Project Domain, read Abstract Application Models in Project Domain, write Abstract Application Models in Project Domain).
- Roles are combined in Groups (e.g. Resource Expert in Project Domain and Application Expert in Project Domain).
- Users are added to groups according to their roles.
- In Secrets Vault a separate Secret Storage is created for the project.
- Policies defining access rights for Secret Storage are created (e.g. read, write Secrets and read Secrets).
- Roles representing a mapping entity between policies and claims in JWT token are defined (e.g. Resource Expert - read, write Secrets, Application Expert - read Secrets).
- Project Domain is configured.

### 3.2.3 Login

For issuing a JSON Web Token (JWT), Access Token OAuth 2.0 Resource Owner Password Credentials Grant flow is used. Since the whole flow is confined inside SODALITE there is no insecure exposure of credentials to the client. A JWT Access Token, once issued by IAM, is then used across the whole SODALITE workflow. The primary client in the Login scenario is SODALITE IDE, but IAM also allows a configuration for different clients, i.e. one set of rules can be applied to a user's logins from IDE and another set of rules for Automation components like Deployment Refactorer. In order to enhance security and forbidding unauthorized calls to IAM Token Endpoint, each client has a client secret assigned that is sent to IAM upon token creation to validate. User credentials and client secret are provided to the IDE during configuration.

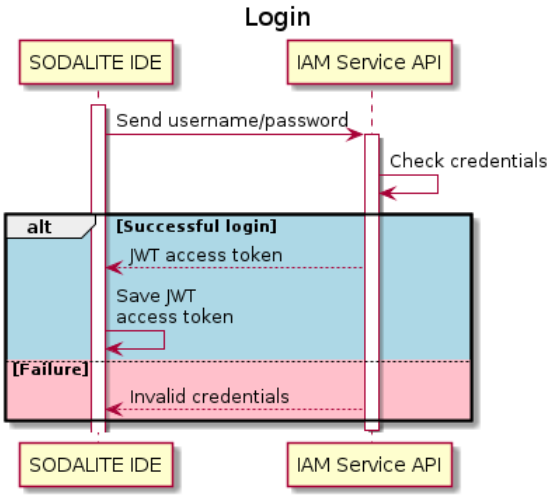



Figure 4 - Interaction between IDE and IAM.

Figure 4 describes interaction between IDE and IAM.

- IDE sends a HTTPS GET request to IAM Token Endpoint containing these credentials together with IDE client id and client secret.
- IAM verifies validity of client id and client secret and validity of user login/password.
- If all credentials are valid a JSON Web Token is returned. This token contains user security claims that grant access to different namespaces and Vault secrets.
- That JWT stored in IDE is used for authorising HTTPS requests to SODALITE services.
- If credentials are invalid a HTTP 401 error is returned.

### 3.2.4 Check JWT token

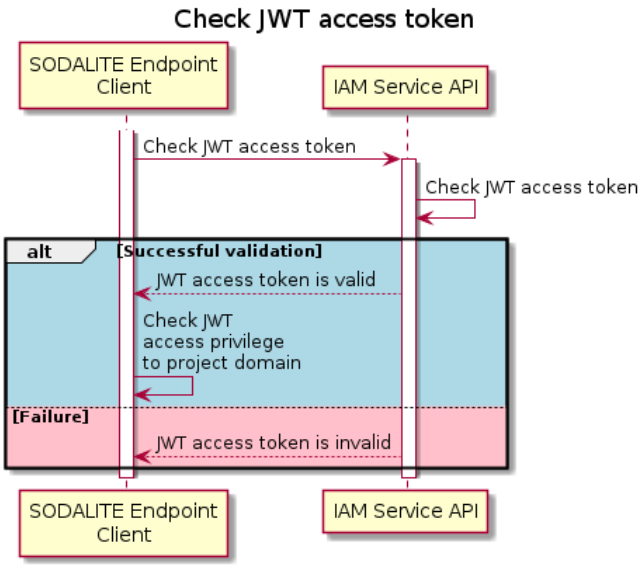



Figure 5 - interaction between SODALITE endpoint and IAM.

Figure 5 describes interaction between SODALITE endpoint and IAM.

- If an API call requests access to some protected resource (model, blueprint, secret etc) and has to be authorized it must be accompanied with JSON Web Token in the Authorization header.
- API Endpoint ensures that the token provided is valid by sending an HTTPS GET request to IAM Introspection Endpoint.
- IAM verifies that the signature is valid and the Token has not expired and returns a decoded token as a result.
- SODALITE endpoint checks that the claims in the Token correspond to ones required for accessing the protected resource.
- If the Token is invalid HTTP 401 error is returned.

### 3.2.5 Save Secret in the Vault

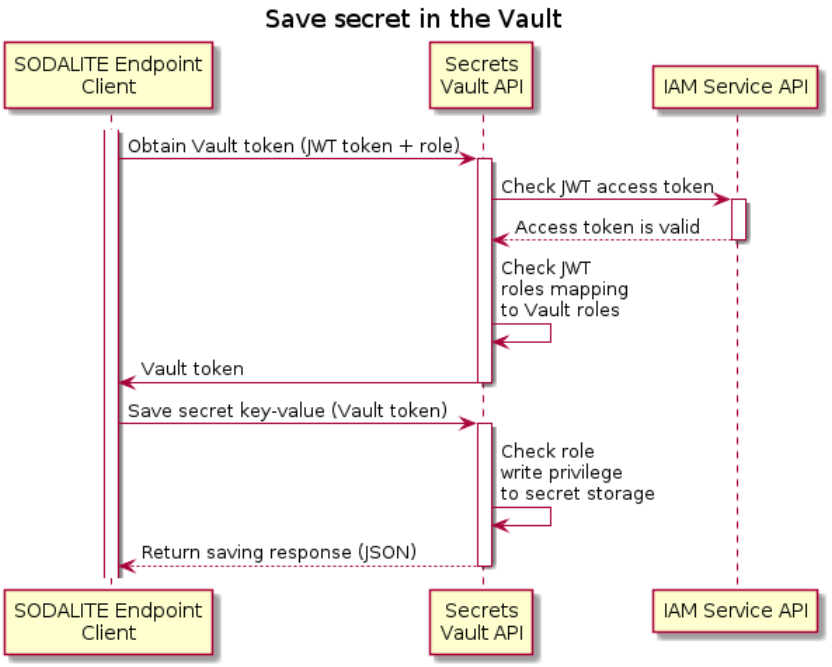



Figure 6 - interaction between a SODALITE service and Secrets Vault.

Figure 6 describes interaction between a SODALITE service and Secrets Vault.

- In order to save a secret in Secret Vault an Endpoint Client must obtain a short-lived Secret Vault Token first. In order to do this a Client must provide a JWT token and a name of the Vault role that grants access to update the Secret Storage.
- Secret Vault ensures that the JWT provided is valid by sending an HTTPS GET request to IAM Introspection Endpoint.
- If the JWT is valid, Secret Vault verifies that it contains claims that are required for the Vault role requested.
- A short-lived Vault token is returned.
- Using this token Endpoint Client sends a HTTPS POST request containing secret address, secret key-value authorized by a Vault token obtained earlier.
- Secret Vault ensures that the role associated with the Vault token has write privileges for the Secret Storage where the secret is saved.
- A successful response is returned.

### 3.2.6 Read Secret from the Vault

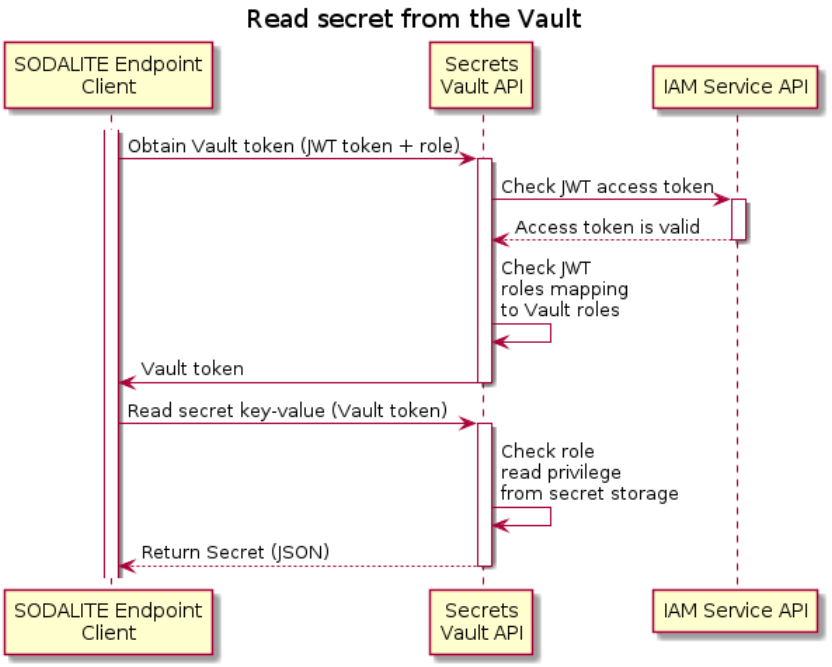



Figure 7 - interaction between a SODALITE service and Secret Vault.

Figure 7 describes interaction between a SODALITE service and Secret Vault.

- In order to read a secret from Secret Vault an Endpoint Client must obtain a short-lived Secret Vault Token first. In order to do this a Client must provide a JWT token and a name of the Vault role that grants access to read the Secret Storage.
- Secret Vault ensures that the JWT provided is valid by sending an HTTPS GET request to IAM Introspection Endpoint.
- If the JWT is valid, Secret Vault verifies that it contains claims that are required for the Vault role requested.
- A short-lived Vault token is returned.
- Using this token Endpoint Client sends a HTTPS GET request for the secret address authorized by a Vault token obtained earlier.
- Secret Vault ensures that the role associated with the Vault token has read privileges for the Secret Storage where the secret is stored.
- A response containing secret key-value is returned.

## 3.3 Modelling Layer

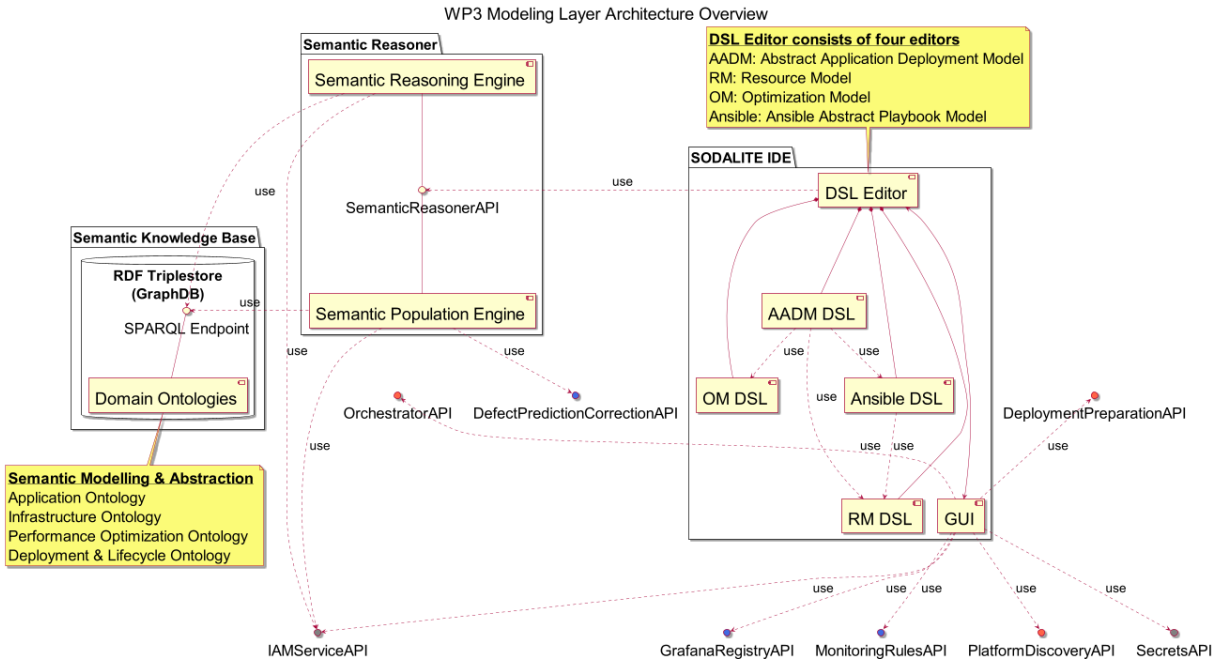



Figure 8 - SODALITE Modelling Layer.

Figure 8 shows the internal architecture of the SODALITE Modelling Layer. The interfaces offered by other components are also highlighted. A set of SODALITE domain ontologies, resulting from the abstract modelling of the related domains (applications, infrastructure, performance optimisation and deployment), are hosted in a SPARQL-served RDF Triplestore (GraphDB), constituting SODALITE's Semantic Knowledge Base. A dedicated middleware (Semantic Reasoner) enables 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 provides a user interface with a DSL editor, for the design of deployment models using knowledge retrieved from the Semantic Reasoner. The IDE also communicates with other system APIs for the monitoring of the deployment lifecycle.

The main changes introduced in the Modelling architecture compared with D2.2 report are the following:

- IDE block: Monitoring API is composed of two lower level APIs, the GrafanaRegistry API and MonitoringRules API. The IDE interacts with the GrafanaRegistry API, upon the deployment of an AADM, to register the required monitoring dashboards associated with the deployed application. The MonitoringRules API is required to register new monitoring rules authored by the AoE in the IDE editor. IaCVerificationAPI has been removed from the interactions of this layer, as it is used internally to the IaC Layer. Additionally, IDE interacts with the Refactorer API for notifying to the Refactoring layer about new application deployments to be taken care of.

### 3.3.1 Component descriptions

The descriptions in this subsection are essentially those from previous versions of the Architecture, and are included here for completeness.

#### 3.3.1.1 SODALITE IDE

*Functional Description:*

The SODALITE IDE provides complete support for the authoring lifecycle of abstract application deployment models (AADM in the following), Resource Models (RM), Optimization Models (OM) and Ansible Models (AM).

The IDE enables Application Ops Experts (AOE in the following) to create AADMs for their applications. The IDE also permits Resource Experts (RE) to create RMs that defines types of infrastructure reusable resources, Quality Experts (QE) to define OMs that improves the runtime performance of application components in target computing infrastructures, and AOEs to create AMs that defines implementations for the operations of the interfaces adopted by infrastructure resources and applications.

The IDE assists AOEs, REs and QEs in the textual authoring of the AADM (for these models graphical authoring is also supported), RMs, OMs, and AMs, thanks to features such as: a) syntax highlighting, b) autoformatting, c) autocompletion and quick fixes, d) syntactic and semantic validation/error checking, e) scoping (cross-references), f) outlining, g) context-aware smart content-assistance, 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 (adopting the QE role) and Ansible implementations for interface operations. RE can describe in the RM reusable types for infrastructure resources, their properties and attributes, the capabilities they offer, the requirements they need, or the policies they adhere to.

The IDE checks the authored models 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 user in the IDE for further inspection. Eventually, the user can refine/amend the model based on them. Additionally, the IDE can request the Semantic Reasoner for existing infrastructure resources that may fulfil requirements expressed in application components or in other resources. Matching resources are presented to the user in the IDE.

Models can be stored into the Semantic KB. Complete CRUD operations on stored models are supported from the IDE. Entities (e.g. application components, infrastructure resources) stored in the KB can be shared with other users.

The IDE also supports the deployment of AADMs into the SODALITE Runtime Layer, the governance of deployed applications, the creation of VM images from descriptors and the discovery of target infrastructures as RMs, by using the IaC Layer.

*Input:*

- AADM: AOE knowledge, other reusable resources taken from the KB, references to OMs to optimize concrete application components, references to AMs for implementations of operations in interfaces,
- AADM inputs,
- RM: RM knowledge, other reusable resources taken from the KB,
- OM: OM knowledge,
- AM: AOE knowledge, Ansible modules,
- Image descriptors,

*Output:*

1. An AADM stored in the Semantic KB
2. An AADM deployed in the Runtime layer.
3. An AADM stored in the Semantic KB
4. A RM stored in the Semantic KB
5. A OM bound to AADM components for optimization
6. An AM to be bound to interface operations for AADM components or RM types

7. An image created and registered in the registry.
8. Discovered RMs for target infrastructure.

*Programming languages/tools:*

- SODALITE DSL: XText, EMF
- SODALITE IDE: Eclipse
- SODALITE IDE DSL Editor: XText, Sirius, Java

*Dependencies:*

1. Semantic Reasoner REST API
2. Semantic Reasoner query language and OWL notation (Turtle)
3. Semantic Reasoner response schema (JSON)
4. IaC Builder REST API
5. xOpera Orchestrator
6. AAI Keycloak REST API

*Critical factors:*

The latency accessing the SODALITE KB (and retrieving request responses) from the IDE may prevent the IDE Editor from presenting real time recommendations, node targets, etc in the code assistance without some delay. Similar delay could be present when saving models into the KB, or when deploying AADM into the SODALITE Runtime Layer.

Models (AADM, RM) need to be serialized in the selected OWL Turtle 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.

#### 3.3.1.2 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.
5. Requests from the Platform Discovery Service for inserting of the discovered infrastructure resources into KB.
6. Requests from the Refactoring Option Discoverer for discovering new nodes and resources.

*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:*

- *Semantic Reasoner API:* Java, JAX-RS REST API
- *Semantic Population Engine:* Java, SPARQL query language
- *Semantic Reasoning Engine:* Java, SPARQL query language

*Dependencies:*

- Alignment with SODALITE IDE and its DSL
- Bug Predictor REST API
- AuthN/AuthZ REST API

*Critical factors:* KB Schema and SODALITE DSL must be semantically compatible.

#### 3.3.1.3 Semantic Knowledge Base (KB)

*Functional Description:*

The KB is SODALITE's semantic repository that hosts the models (ontologies) created in WP3. The ontologies are populated with domain knowledge, i.e., abstract and target resource types, resource patterns, deployment patterns, dependencies, inconsistencies, etc. This component interacts with the KBS and offers capabilities for knowledge storage and manipulation.

*Input:*

Queries from the KBS for the insertion, update, deletion and retrieval of knowledge. More complex queries 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:* N/A

*Critical factors:* N/A

### 3.3.2 Use Case Sequence diagrams

The core activity associated with the modelling layer is the one associated with UC1 (Define Application Deployment Model). However, it depends on the fact that the resources to be used for deploying an application have been specified. For this reason, we focus first on UC13 – Model Resources. For this document, only UC13 has been updated. The other UCs have been copied from D2.2 for completeness. The interaction with IAM and Vault is abstracted in all sequence diagrams and is occurring in accordance with the diagrams presented in Section 3.2.

### 3.3.2.1 UC13: Model Resources

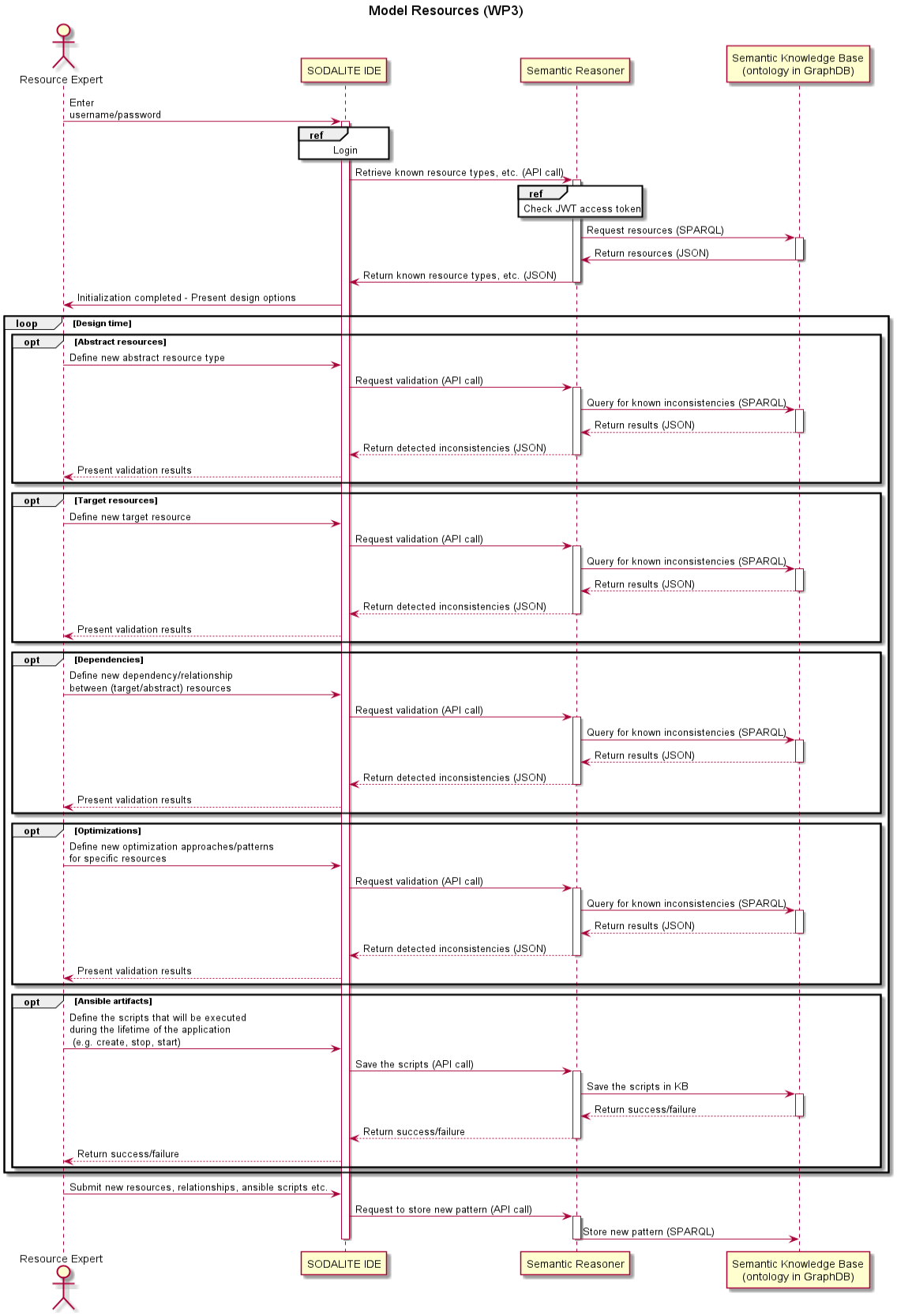



Figure 9 - Sequence Diagram for UC13.

Figure 9 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.

As part of D2.3, one new interaction has been added to the UC with respect to the authoring of Ansible scripts. Through the interfaces of a resource, scripts can be executed through the lifecycle of the application (create, stop, start etc.) such as the initialization of a database. Within the application deployment modelling process, the operations associated with application component lifecycles (interfaces) can be modeled using the Ansible DSL editor of the IDE.

### 3.3.2.2 UC1: Define Application Deployment Model

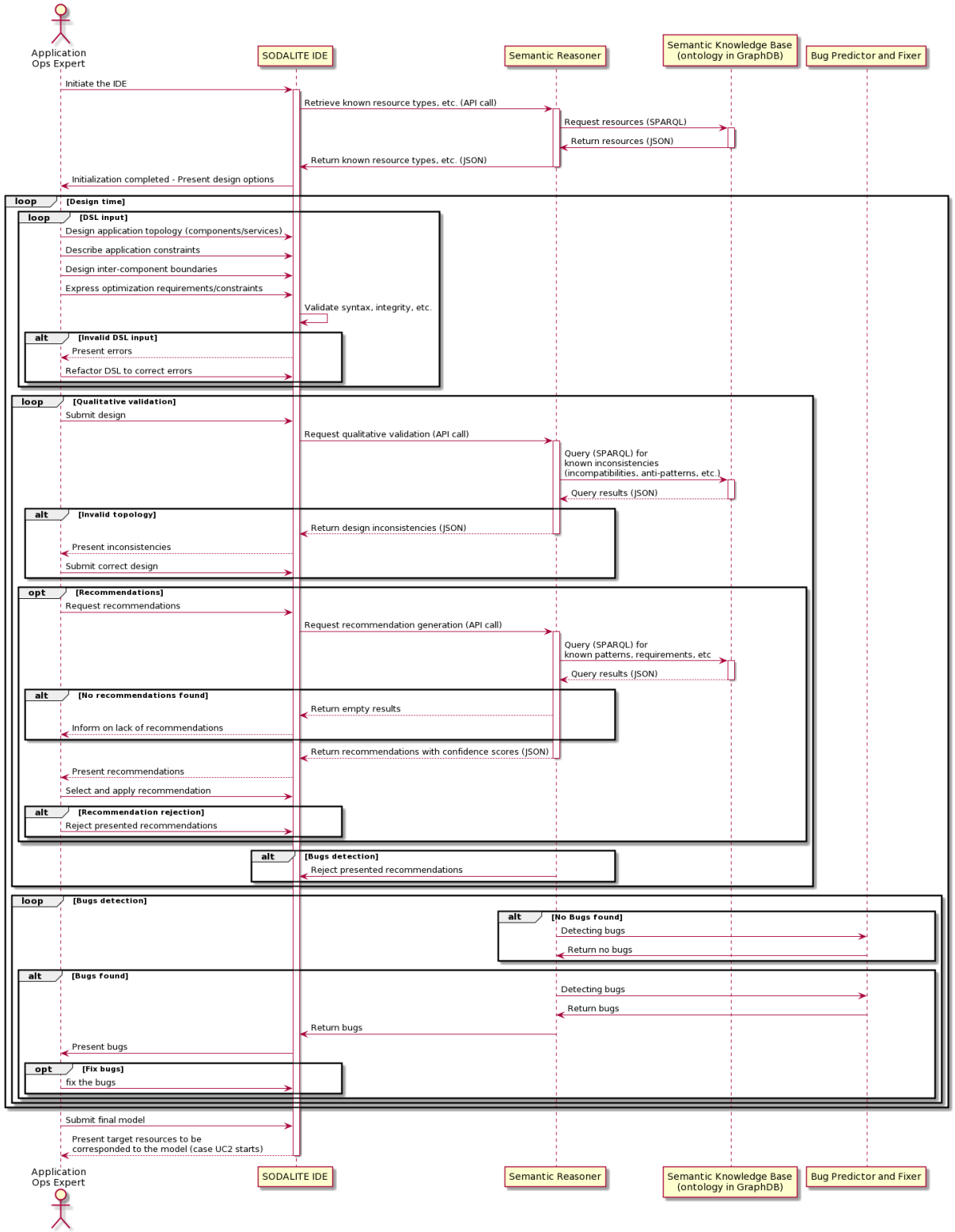



Figure 10 - Sequence diagram for UC1.

Figure 10 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. Finally, bugs and software smells are detected by Bug Predictor. The use case output is a valid ADM.

#### 3.3.2.3 UC2: Select Resources

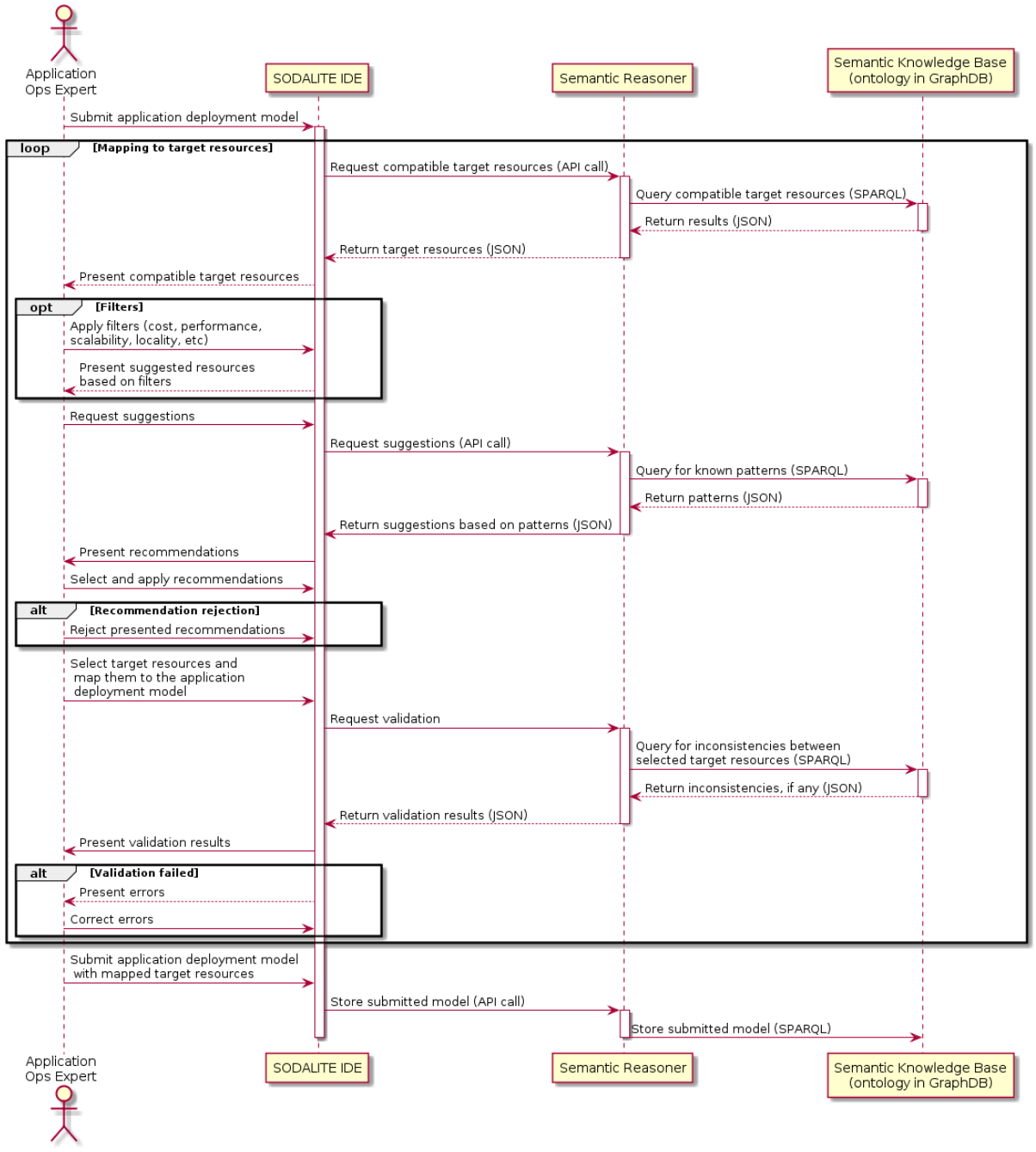



Figure 11 - Sequence diagram for UC2.

Figure 11 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 is 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.

### 3.3.2.4 UC12: Map Resources and Optimisations

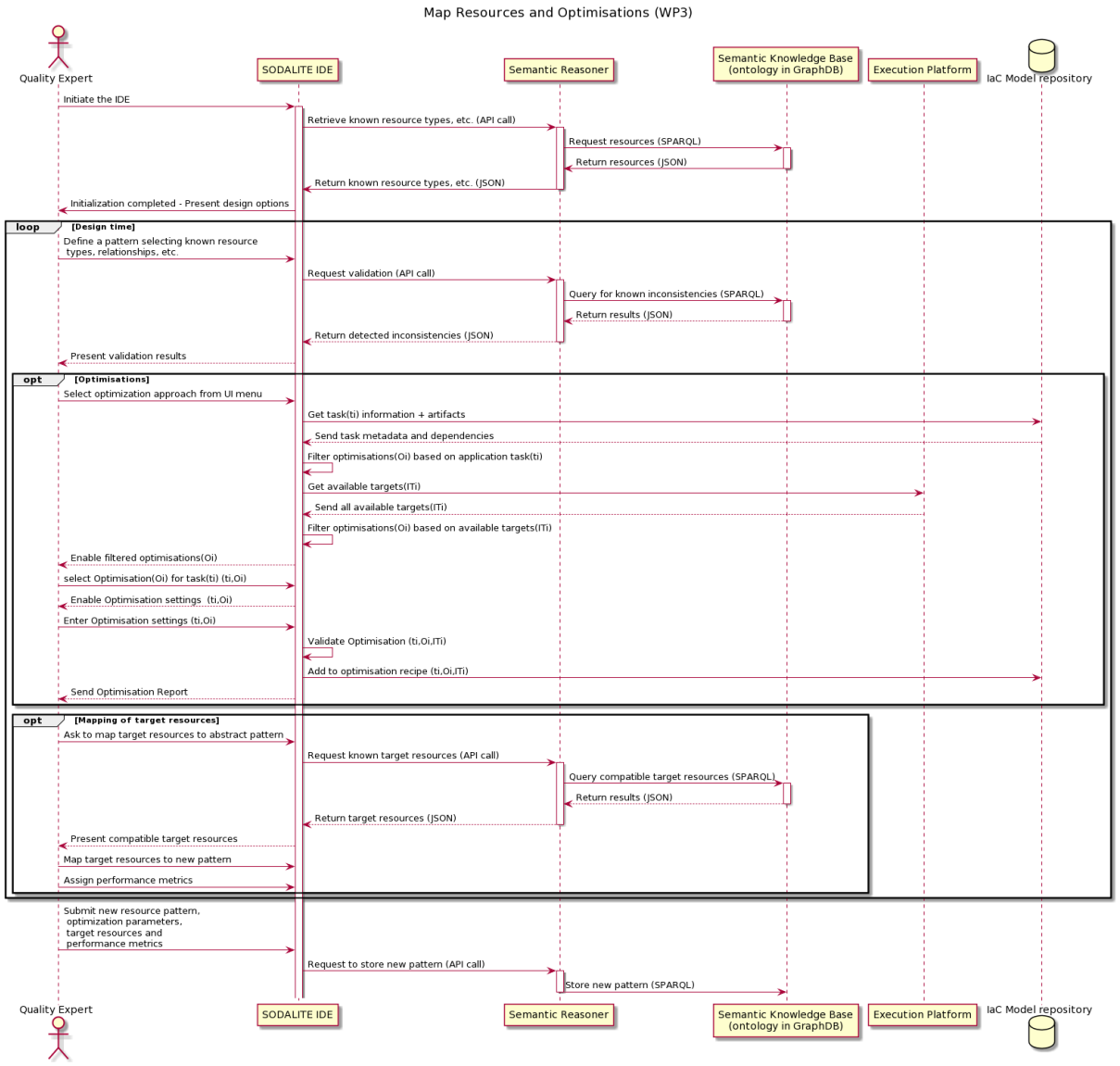



Figure 12 - Sequence diagram for UC12.

Figure 12 describes the interaction between the SODALITE components while implementing UC12 - Map Resources and Optimisations. This use case describes the process of defining abstract resource patterns by a Quality Expert (QE). 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 optimisations are enabled for the QE to select from. The QE also has to enter the settings for any selected optimisation. This is stored in the IaC Model Repository.

#### 3.3.2.5 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.

## 3.4 Infrastructure as Code Layer

The Infrastructure as Code Layer (IaC Layer) is the layer that connects the SODALITE Modelling Layer functionalities to Runtime blueprint execution of the models in the SODALITE Runtime Layer. It offers APIs and data to support the optimization, verification and validation process of both Resource Models (RM) and Abstract Application Deployment Models (AADM). However, one of the most important tasks of the IaC Layer is preparing a valid and deployable TOSCA blueprint.

During the last and third year of the project most of the components were released and several were refactored and significantly improved. During the second year of the project Platform Discovery Service had been added to the layer's architecture, to expose a REST API which helps to automate the tasks of the Resource Expert by creating a valid TOSCA platform resource model to be stored into the SODALITE's Knowledge Base. These RMs can then be used during the design of the application deployment models (AADM).

In this period the Application Optimizer component exposing a REST API (MODAK) was updated and further integrated into the pipeline enabling the SODALITE users to statically optimize the application for a given target execution platform.

Both Automation of Application Optimisation on HPC and cloud systems requiring models used for performance prediction have been improved. SODALITE prepares and uses these models for both pre-deployment (static) performance optimization and runtime (dynamic) performance optimization.

During the third year of the project IAM (Identity and Access Management) API and Secret Vault API were fully integrated into the IaC Layer and used by the components that have to protect secrets stored by the user, such as Platform Discovery Service and IaC Blueprint Builder.

During development in the third year of the project a part of the architecture was redesigned and is shown here in Figure 13.

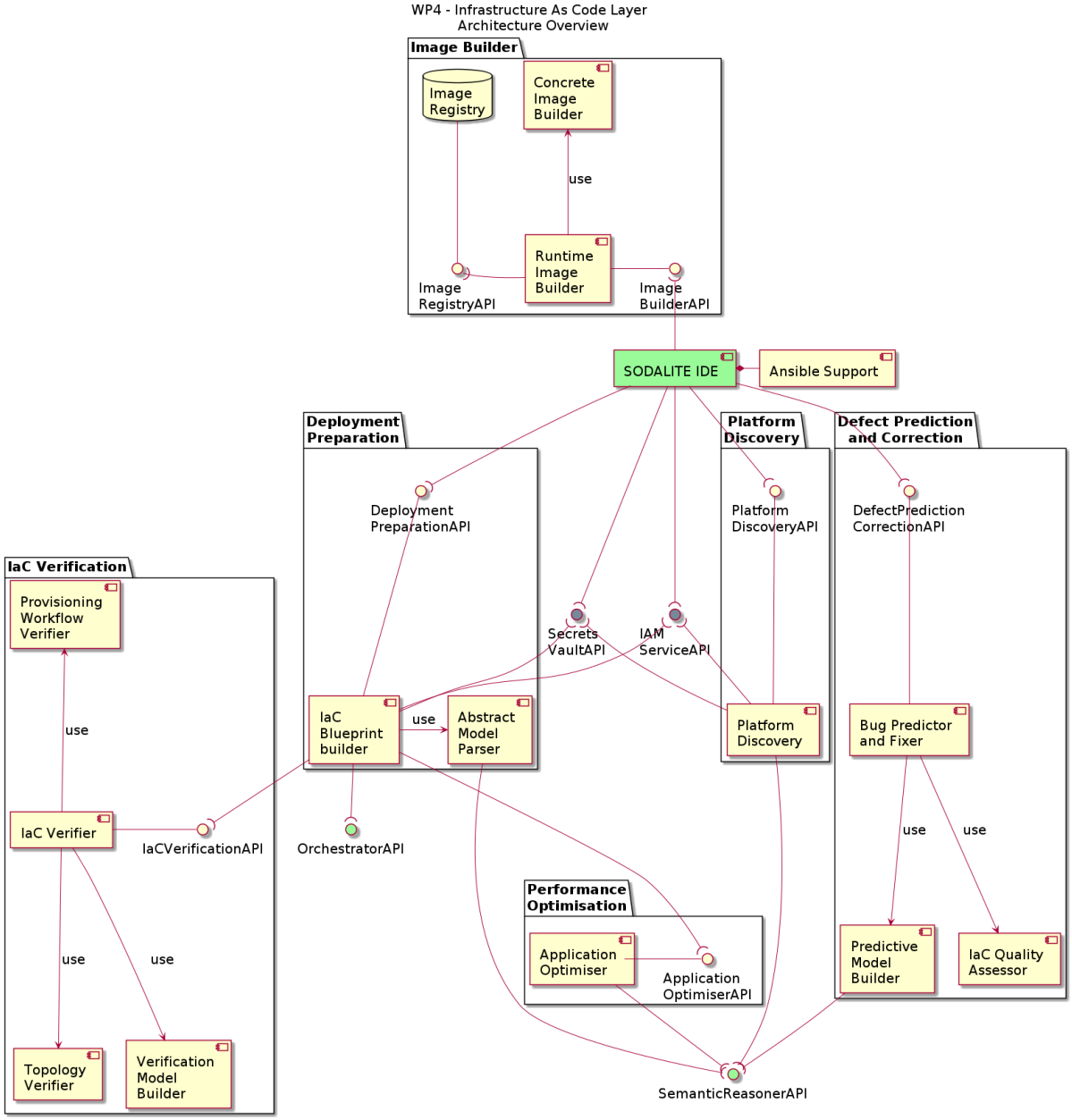



Figure 13 - Infrastructure as Code Layer.

### 3.4.1 Component Descriptions

The descriptions in this subsection are essentially those from previous versions of the Architecture, and are included here for completeness.

#### 3.4.1.1 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 (AADM).

*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.

#### 3.4.1.2 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. This component integrates with MODAK which updates the optimized images that are deployed.

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

*Input*: Abstract application deployment model, IaC Resources Model

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

*Programming languages/tools*: Python

*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.

#### 3.4.1.3 IaC Model Repository

*Functional Description*: IaC Model Repository is a part of the MODAK component and contains:

- Performance Model of an infrastructure based on benchmarks.
- Performance Model of an application based on scaling runs done in the past.

- Mapping of optimisations and applications and their suitability for a particular infrastructure.
- Optimisation recipe for a particular deployment. This contains selected optimisations by the user for an application and infrastructure target.

*Input*: Application type, node type

*Output*: Performance model and optimization recipe

*Programming languages/tools*: Python/Mysql

*Dependencies*: IaC Model Repository interacts with the SODALITE IDE and contains the Performance Model of infrastructure and application (offline analysis).

*Critical factors*: N/A

#### 3.4.1.4 Runtime Image Builder

*Functional Description*: Runtime Image Builder builds the runtime images used by the orchestrator at application deployment

*Input*: Target architecture and artifact definition

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

*Programming languages/tools*: Python

*Dependencies*: Concrete Image Builder

*Critical factors*: N/A

#### 3.4.1.5 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 could also include 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

#### 3.4.1.6 Application Optimiser - MODAK

*Functional Description*: The MODAK package, a software-defined optimisation framework for containerised AI and MPI-parallel applications considered within the SODALITE use cases, is the SODALITE component responsible for enabling the static optimisation of applications before deployment.

*Input*: MODAK requires the following inputs:

1. Job submission options for batch schedulers such as SLURM and TORQUE;
2. Application configuration such as application name, run and build commands;
3. Optimisation DSL with the specification of the target hardware, software libraries, and optimisations to encode. Also contains inputs for auto-tuning and auto-scaling.

An Image Registry contains MODAK optimised containers while performance models, optimisation rules and constraints are stored and retrieved from the IaC Model Repository. Singularity container

technology was chosen to provide a portable and reproducible runtime for the application deployment, due to better performance and native support for HPC.

*Output*: MODAK produces a job script (for batch submission) and an optimised container that can be used for application deployment.

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

*Dependencies*: IaC Model Repository, Runtime Image Builder, Execution Platform, Platform Discovery Service

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

#### 3.4.1.7 IaC Verifier

Functional Description: This component coordinates the processes of verification of the syntax and semantic of the deployment model artfacts, namely TOSCA models and Ansible playbooks.

*Input*:

- IaC models
- Correctness criteria such as well-structuredness and soundness

*Output*:

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

*Programming languages/tools*: Java and Python

*Dependencies*:

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

*Critical factors*: N/A

#### 3.4.1.8 Verification Model Builder

*Functional Description*: This component builds the models required to verify the IaC models, for example, a petri net representation for the workflow in Ansible playbooks.

*Input*: IaC models

*Output*: Verification Models

*Programming languages/tools*: Java and Python

*Dependencies*:

- Topology Verifier

*Critical factors*: N/A

#### 3.4.1.9 Topology Verifier

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

*Input*:

- Representation of TOSCA Models
- Correctness criteria

*Output*:

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

*Programming languages/tools*: Java and Python

*Dependencies*:

- IaC Verifier

*Critical factors*: N/A

#### 3.4.1.10 Provisioning Workflow Verifier

*Functional Description*: This component verifies the provisioning workflow in a given Ansible playbook 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 and Python

*Dependencies*:

- IaC Verifier

*Critical factors*: N/A

#### 3.4.1.11 Bug Predictor and Fixer

*Functional Description*: This component is responsible for predicting bugs/smells in IaC models, suggesting corrections or fixes for the detected bugs/smells, and correcting the bugs/smells applying the fix selected by the Application Ops Expert.

*Input*: Abstract IaC models

*Output*: Bugs/Smells, Fixes

*Programming languages/tools*: Java and Python

*Dependencies*:

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

*Critical factors*: N/A

#### 3.4.1.12 Predictive Model Builder

*Functional Description*: This component builds the models that can be used to detect bugs/smells in IaC models and suggest corrections. The models can include rule-based models, semantic models, and data-driven (machine learning and deep learning).

*Input*:

- IaC artifacts
- Bug/Smell and resolution knowledge (ontology and rules)
- IaC datasets
- IaC metrics

*Output*:

- Ontological Predictive Models
- Data-Driven Predictive Model
- Rule-based Models

*Programming languages/tools*: Java and Python

*Dependencies*:

- Bug Predictor and Fixer
- Semantic Knowledge Base

*Critical factors*: N/A

#### 3.4.1.13 IaC Quality Assessor

*Functional Description*: This component can calculate different quality metrics for IaC artifacts.
*Input*: IaC artifacts
*Output*: IaC quality metrics
*Programming languages/tools*: Java and Python
*Dependencies*:

- Bug Predictor and Fixer

*Critical factors*: N/A

#### 3.4.1.14 Platform Discovery Service

*Functional Description*: Platform Discovery Service takes the data needed as input platform such as platform namespace, project and credentials to access the platform to create a usable TOSCA Resource Definition from a target. This model can be stored in the SODALITE KB and reused by the AOE at Application Deployment design time.
*Input*: Target Namespace, Project, Platform Access Credentials
*Output*: TOSCA resource definition template
*Programming languages/tools*: Python, TOSCA, Ansible
*Dependencies*:

- Target platforms
- IAM Service API
- Secrets Vault API
- Semantic Knowledge Base

*Critical factors*: N/A

### 3.4.2 Use Case Sequence diagrams

For this document, only UC17 has been updated. The other UCs have been copied from D2.2 for completeness. The interaction with IAM and Vault is abstracted in all sequence diagrams and is occurring in accordance with the diagrams presented in Section 3.2.

### 3.4.2.1 UC3: Generate IaC

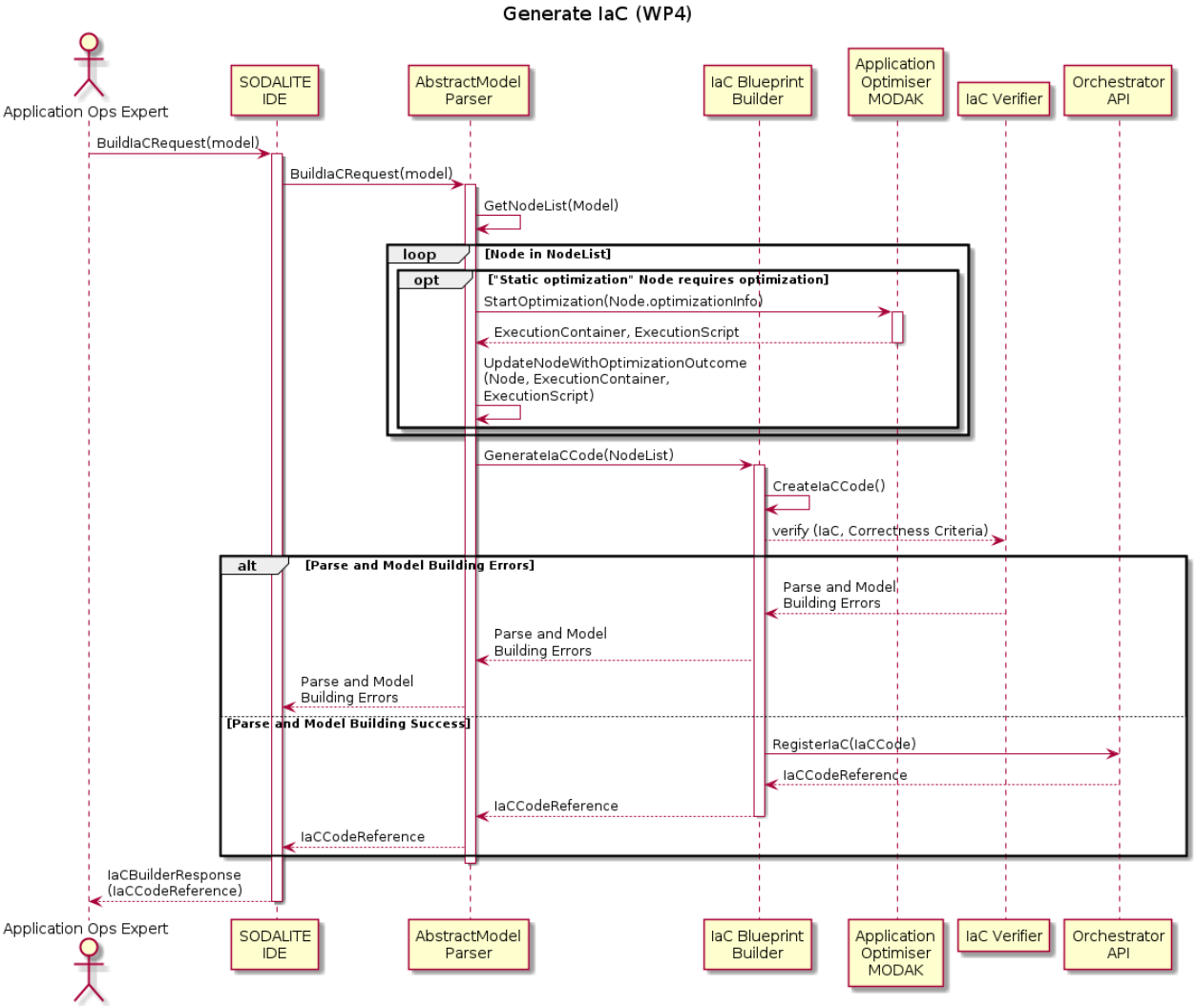



Figure 14 - Sequence diagram for UC3 Generate IaC.

Figure 14 describes the updated Sequence Diagram showing interaction between the SODALITE components while in the process of generating 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 Expert 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 with IaC node definition from the IaC Resource Model which is built into 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 and optimized images targeting a specific infrastructure platform. IaC Blueprint Builder represents a central point of the AADM to IaC transformation. After providing optimal artifacts from the Application Optimiser it creates and registers the TOSCA blueprint with the Orchestrator returning a blueprint registration token. The build-time results are then returned back to the Application Ops Expert through a registered blueprint token provided.

### 3.4.2.2 UC4: Verify IaC

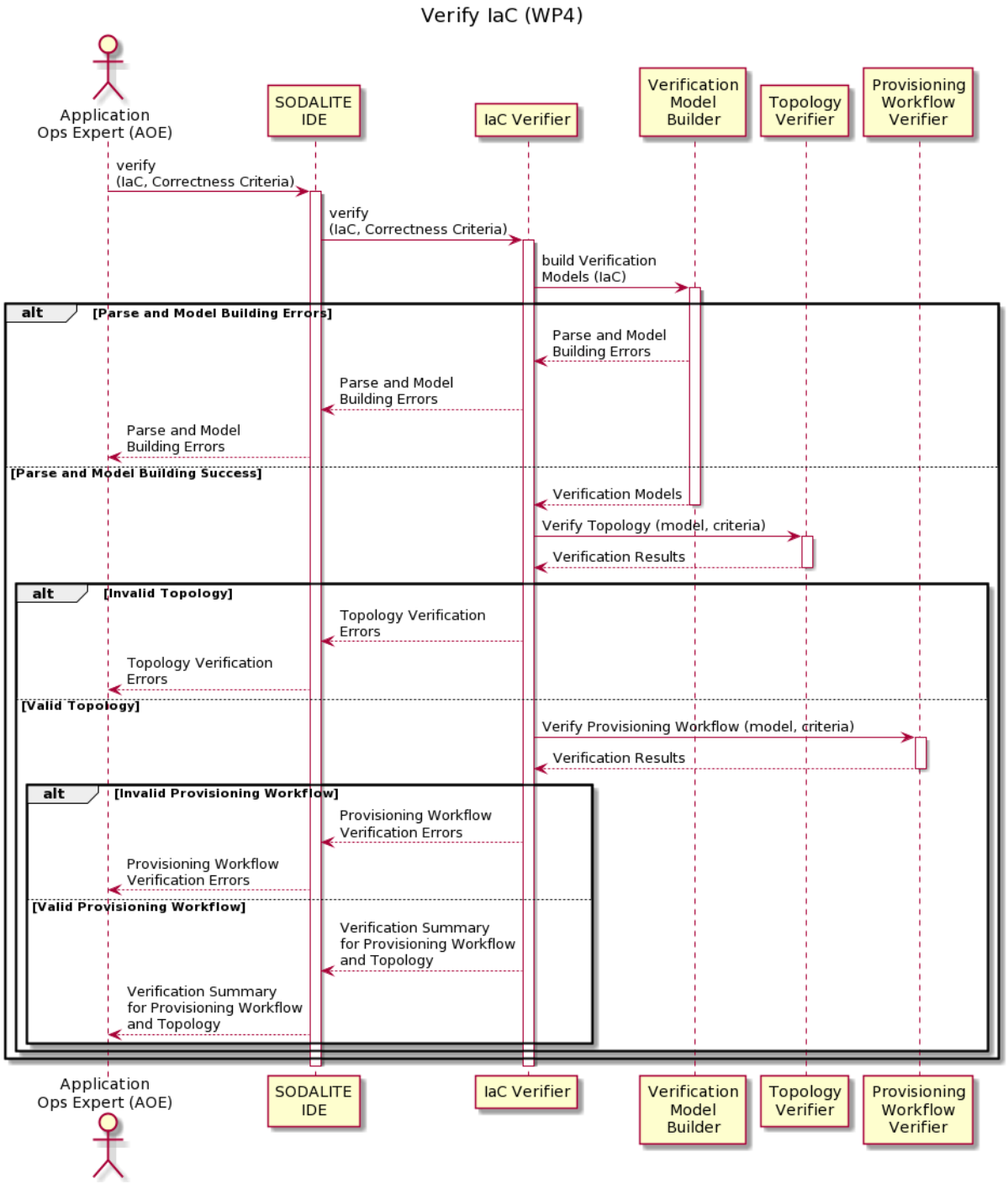
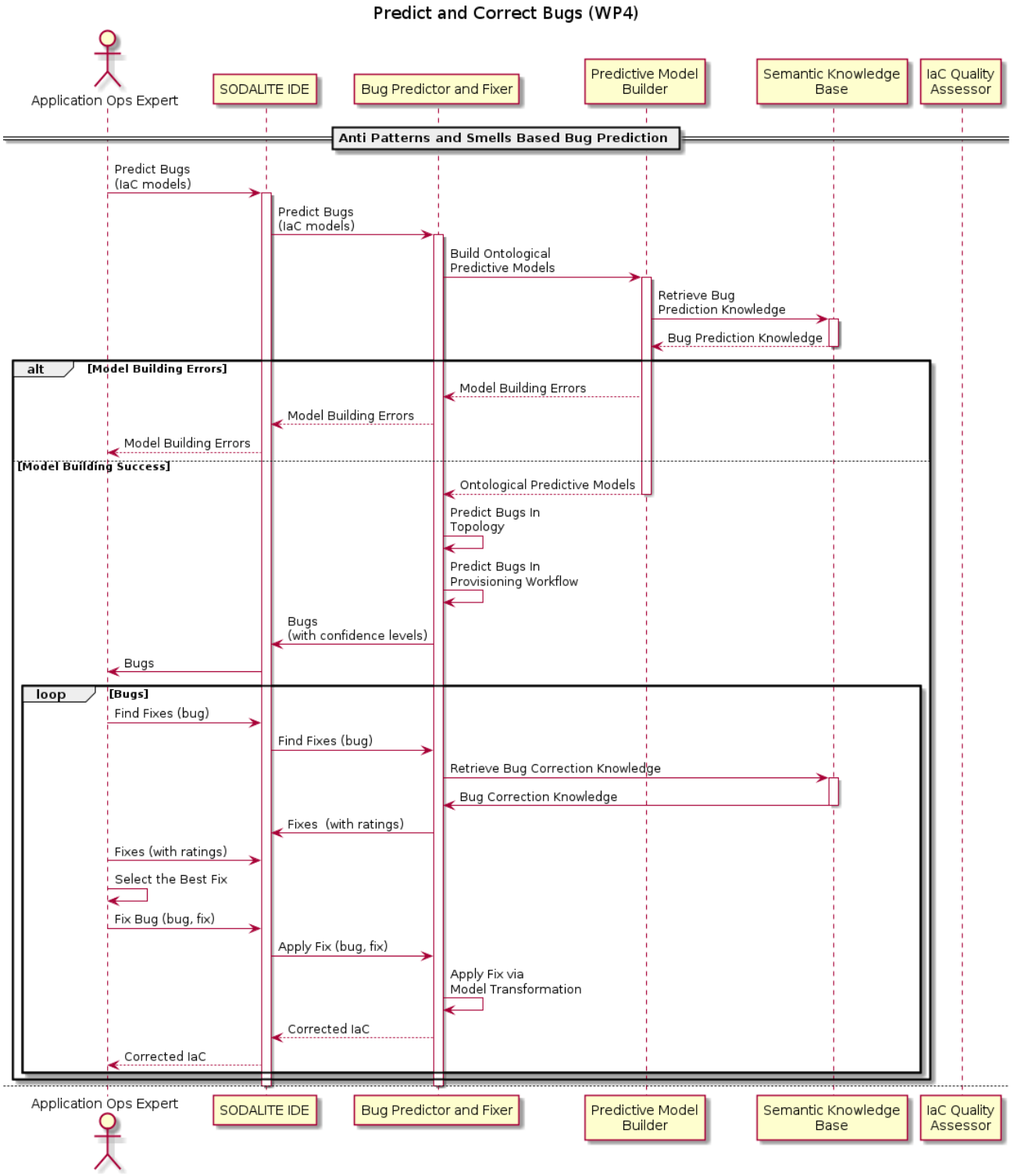



Figure 15 - Sequence diagram for UC4 Verify IaC.

Figure 15 describes the interaction between the SODALITE components while implementing UC4 - Verify IaC. Application Ops Expert provides the abstract IaC modelling 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 need 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 (e.g., Petri net

models). 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.

### 3.4.2.3 UC5: Predict and Correct Bugs



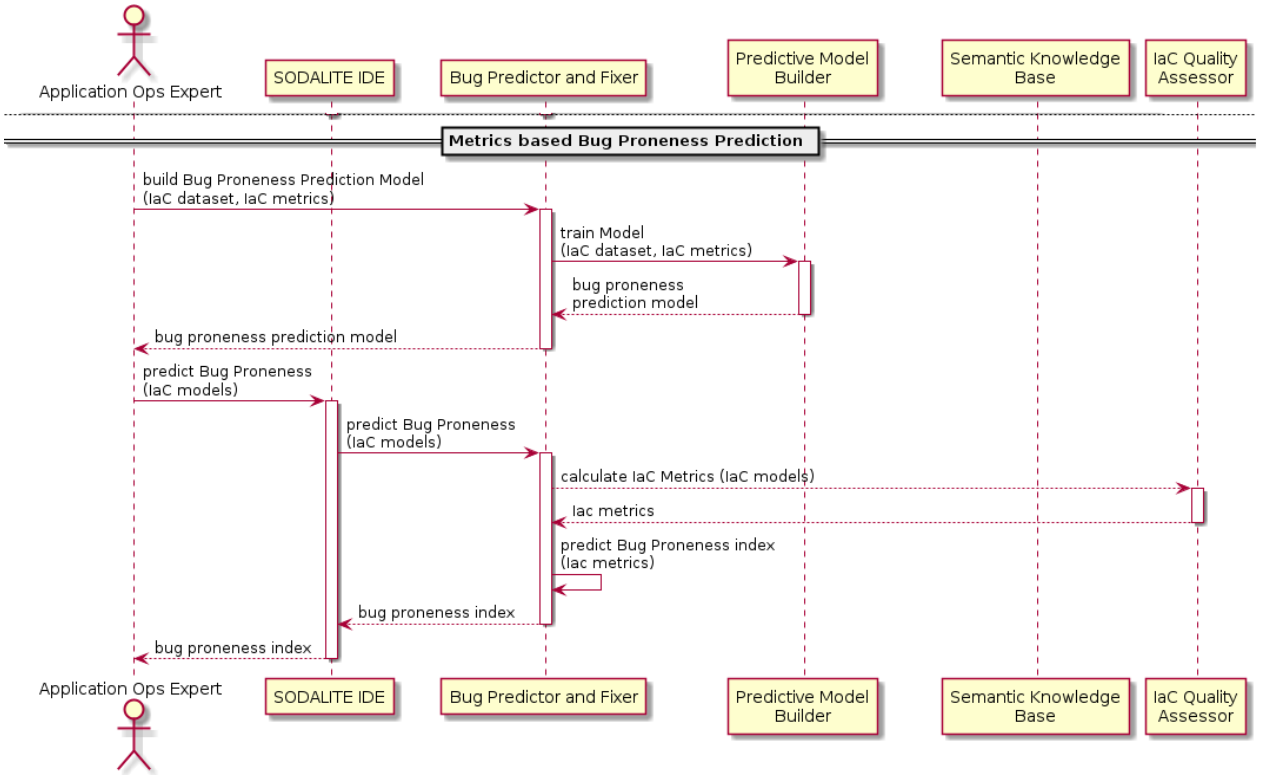



Figure 16 - Sequence diagram for UC5 Predict and Correct Bugs.

Figure 16 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.

#### 3.4.2.4 UC11: Define IaC Bugs Taxonomy

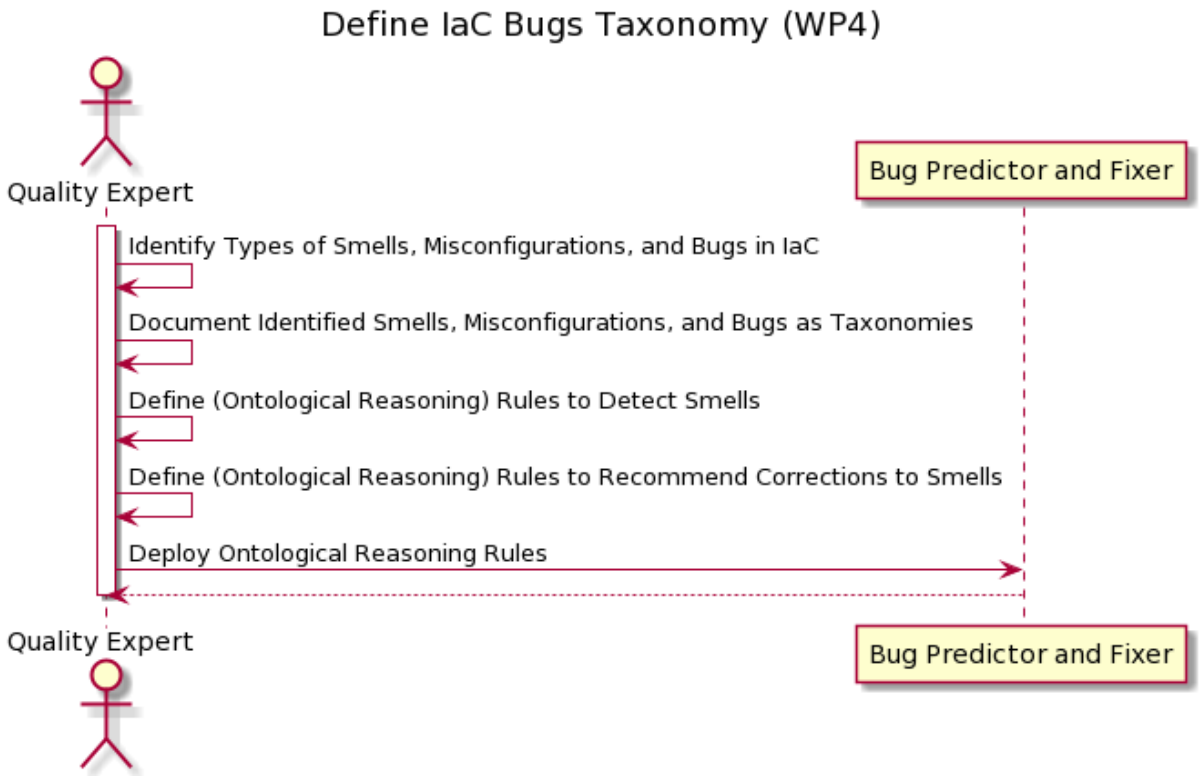



Figure 17 - Sequence diagram for UC11 Define IaC Bugs Taxonomy.

Figure 17 describes the interaction between the SODALITE components while implementing UC11 - Define IaC Bugs Taxonomy. Quality experts identify the types of smells, misconfigurations, and bugs in IaC, and build the taxonomies. The ontological reasoning rules required for detecting smells and suggesting fixes for smells are also defined. Finally, the ontological reasoning rules are deployed in Bug Predictor and Fixer.

### 3.4.2.5 UC15: Statically Optimise Application and Deployment

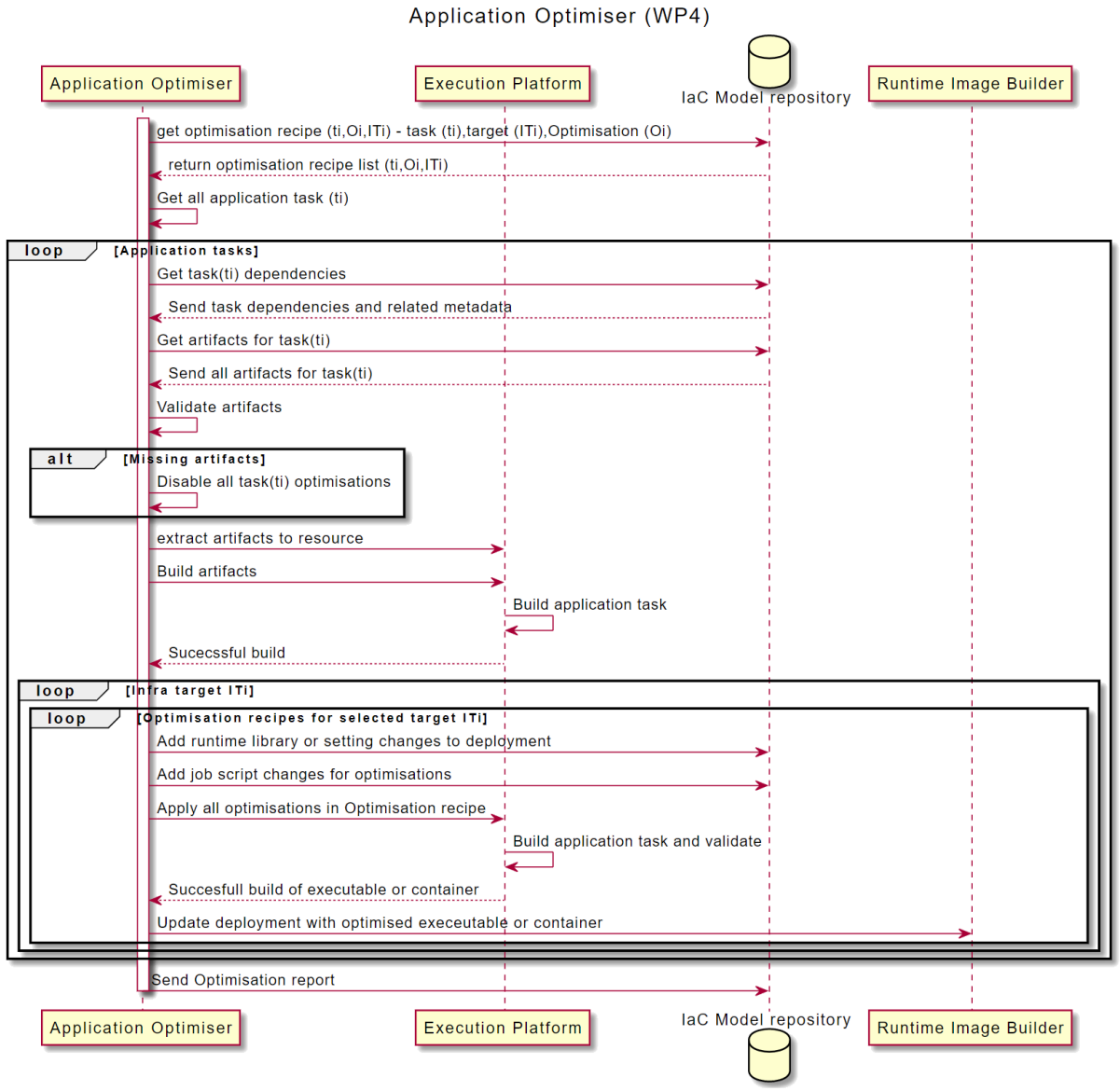



Figure 18 - Sequence diagram for UC15.

Figure 18 describes the interaction between the SODALITE components while implementing UC15 - Statically Optimise 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 is an optimised application executable or a container. The optimisation 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 is made at the end of this process.

### 3.4.2.6 UC16: Build Runtime images

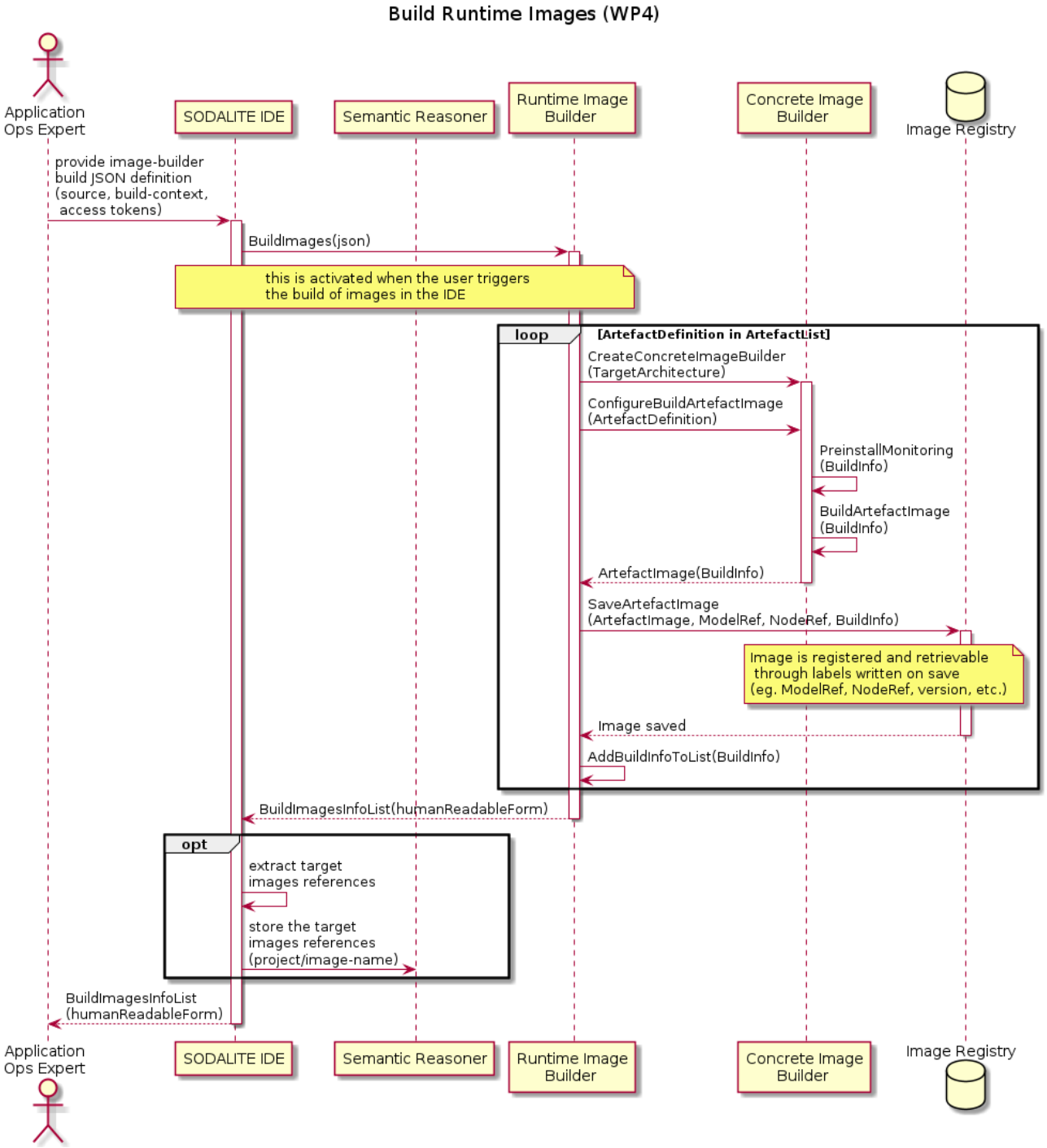



Figure 19 - Sequence diagram for UC16.

Figure 19 - 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.

### 3.4.2.7 UC17: Platform Discovery Service

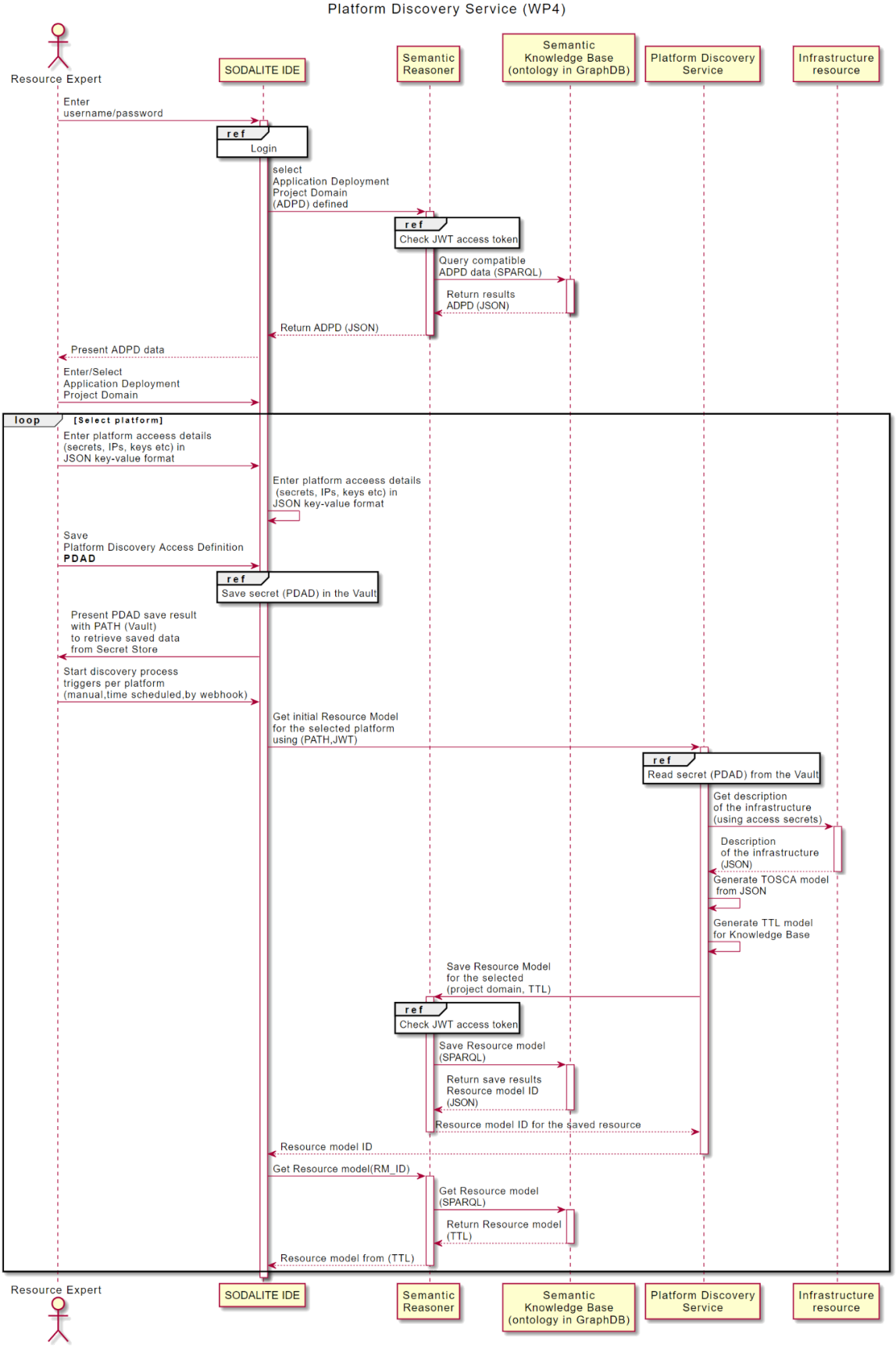



Figure 20 - Sequence diagram for UC17 Platform Discovery Service.

Figure 20 describes the interaction between the SODALITE components while implementing UC17 - Platform Discovery Service. The process is initiated by Resource Expert (RE) by selecting the project domain to which a specific set of platform resources will be added. The data about defined project domains is retrieved by Semantic Reasoner API which, based on user privileges defined in the access token, checks the validity and the list of project domains the user is authorized to approach (described in detail in Section 3.2.4). For every platform added to the project domain the RE adds a specific set of values needed by the Platform Discovery Service to execute the platform discovery (described in detail in Section 3.2.5). The set of values includes a namespace definition for the created resource, project domain, and platform access keys. The data is stored in a central Secret Vault and accessible by the Platform Discovery Service providing a valid access token for retrieving data from the Secrets Vault (described in detail in Section 3.2.6). The user can manually initiate the discovery of the infrastructure resources through IDE. Once the Platform Discovery Service has access to the platform it executes the discovery using platform specific tools that return a JSON description of the resources. In the next step Platform Discovery Service executes a JSON to TOSCA service template transformation and returns the results to the caller as TOSCA service template definition of the platform or stored into the Knowledge Base, depending on the call parameters.

## 3.5 Runtime Layer

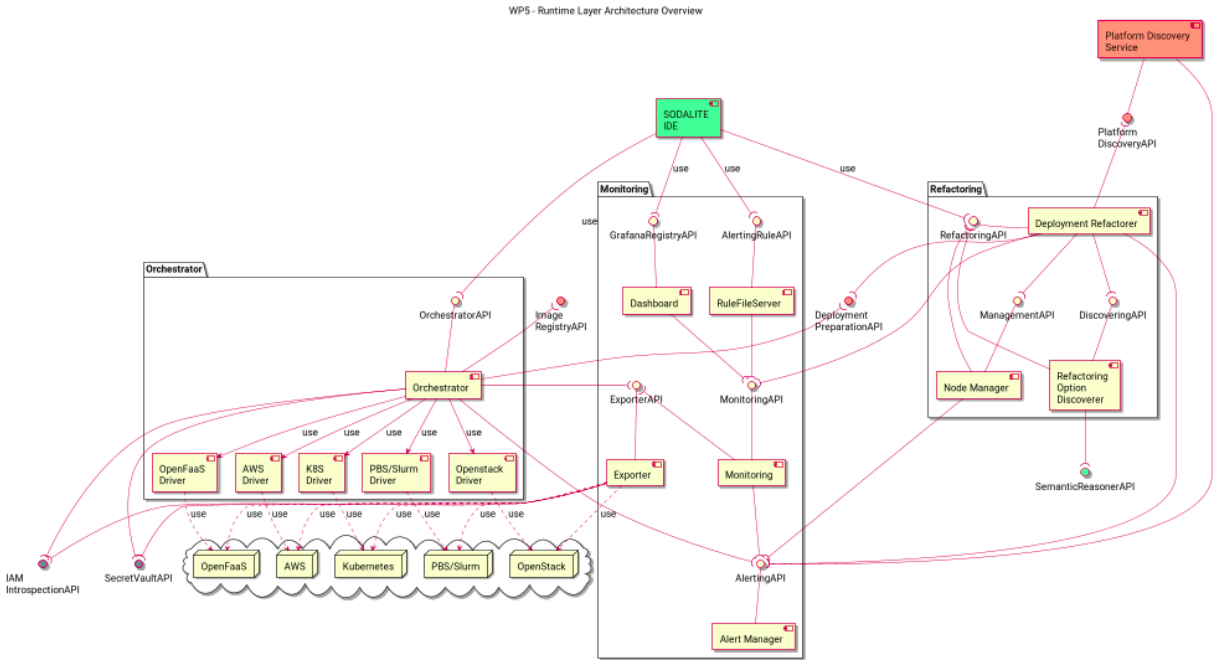



Figure 21 - Runtime Layer.

The Runtime Layer of SODALITE (see Figure 21) 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 - receives the topology description of the application to be deployed or re-deployed, as a blueprint expressed in TOSCA, and deploys the application components on the target infrastructure.
- Monitoring - monitors the application components and the infrastructure where they are deployed to be used by Refactoring and other interested SODALITE actors.
- Refactoring - proposes a new application deployment topology to fulfil the application goals. When it modifies the model in the Semantic Reasoner, it calls the Deployment

Preparation API, which triggers the generation of a new blueprint that arrives to the Orchestrator to initiate the redeployment.

The main changes introduced in the runtime architecture w.r.t. the version reported in D2.2 are the following:

- Orchestration block: different supported HPC schedulers are explicitly labeled within the PBS/Slurm driver and target infrastructure boxes. The orchestrator also uses the ExporterAPI to set up and configure monitoring exporters delivered within the application components and target infrastructures.
- Monitoring block: two new APIs are included, namely the GrafanaRegistryAPI, provided by the dashboard and consumed by the IDE, and the AlertingRuleAPI, provided by the new component RuleFileServer and consumed by the IDE. Upon an application deployment, the IDE registers it for new monitoring dashboard generation using the GrafanaRegistryAPI. AoEs can also register newly created monitoring rules by using the AlertingRuleAPI. Other internal connections among monitoring components have been fixed. Some monitoring exporters (e.g. HPC Exporter) must now use IAM Introspection API and SecretVaultAPI to obtain the user's secrets required to get access to target infrastructures to monitor. WP4 Platform Discovery Service (PDS) also plays a role in the Runtime Layer: Refactoring registers itself in PDS to receive notifications on infrastructure updates. Edge node label monitoring (a specific implementation for Monitoring/Alerting) notifies PDS on detected changes in the Edge infrastructure.
- Refactoring block: Deployment Refactoring consumes monitoring metrics through the Monitoring API.

The interaction between the IDE and the runtime components have been updated with the usage of the RefactoringAPI, GrafanaRegistryAPI and AlertingRuleAPI. See Section 3.5.1.3 for a description of those IDE-Runtime interactions.

### 3.5.1 Component Descriptions

There is one additional monitoring component (Rule File Server) not included in the Runtime architecture presented in D2.2, which is included in this version. The remaining component descriptions are copied from D2.2 for completeness.

#### 3.5.1.1 xOpera REST API

*Functional Description*: The xOpera Orchestrator manages the lifecycle of an application deployed in heterogeneous infrastructures. xOpera REST API is deployed as a dockerized component that encapsulates xOpera orchestrator and provides additional functionalities to the API clients, such as TOSCA blueprint registration, deployment session handling, blueprint and session persistence with the possibility to share registered TOSCA blueprints among different users and additional deployment governance endpoints.

*Input*: TOSCA/Ansible Blueprint Deployment plan

*Output*: Configuration of target infrastructures and applications

*Programming languages/tools*: Python, Ansible

*Dependencies*:

- Target infrastructures: HPC (PBS/SLURM), OpenStack, AWS, Kubernetes, OpenFaaS
- IAM, Secrets Vault
- Exporters

*Critical factors*: Each orchestrator has its own limitations. This results in limitations concerning the possibility to apply certain actions on the managed application.

#### 3.5.1.2 Monitoring + Exporters

*Functional Description*: Gathers metrics from the heterogeneous infrastructure and application execution, provided by standard and specialized exporters, allowing query and aggregation on them.

*Input*: Heterogeneous infrastructure

*Output*: Metrics

*Programming languages/tools*: Prometheus, Prometheus query language and API; Go for exporters

*Dependencies*: Probes or exporters that monitor the target infrastructure components and report back metrics .

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

#### 3.5.1.3 Monitoring Dashboard

*Functional Description*: Offers visual, customizable, specialized views that render different monitoring facets of target infrastructures or applications.

*Input*: Monitoring metrics

*Output*: Monitoring views

*Programming languages/tools*: Grafana

*Dependencies*: Monitoring metrics collected from querying the monitoring component.

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

#### 3.5.1.4 Alert Manager

*Functional Description*: This component is notified by the monitoring component upon the detection of monitoring violations specified in registered alert rules. The manager conducts an analysis of the accompanying metrics and reports associated alerts to the interested registered subscribers.

*Input*: Monitoring alert + associated metrics

*Output*: Alert notification sent to subscribers

*Programming languages/tools*: Python, Flask, Gunicorn

*Dependencies*: Registration in monitoring as Alert Manager, SODALITE subscribers registered.

*Critical factors*: N/A.

#### 3.5.1.5 Deployment Refactorer

*Functional Description*: This component refactors the deployment model of an application in response to violations in the application goals. The goals are monitored at runtime by collecting the necessary metrics. A machine learning based predictive model is used to predict the performance of multiple alternative deployment model variants, and to select a suitable deployment model variant for the application (the new deployment model) if the current deployment model leads to performance violations. The new deployment model is deployed through Orchestrator. The new refactoring options can also be discovered at runtime, enabling deriving new deployment model variants. The Refactorer can also detect various anomalies of a given application deployment at runtime using machine learning based models, and generate the alerts, enabling executing the corrective actions. Moreover, the Refactorer can react to the alerts and events generated by the monitoring layer by adapting the current application deployment as necessary. The adaptation policy can be specified as an ECA (Event-Condition-Action) policy.

*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 API (through IaC Blueprint Builder)
- Orchestrator
- Semantic Reasoner
- Monitoring Agent

*Critical factors*: N/A
This component addresses the following application goals:

- satisfy performance (latency and throughput),
- minimize cost/price,
- minimize resource usage;

And uses these data retrieved from the monitoring infrastructure:

- application workload, latency, and throughput,
- cost/price,
- infrastructure resource usage metrics such as CPU, Memory, and Network,
- other metrics such as energy metrics, and HPC-specific metrics.

If data is not sufficient or of good quality, the accuracy and effectiveness of the refactoring/adaptation decisions may decrease. Thus, as necessary, the Refactorer uses the existing techniques[3,4] for handling uncertainty and variable quality of the monitored data.

#### 3.5.1.6 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 Manager oversees multiple concurrent applications. The Node Manager schedules incoming requests for execution on GPUs and CPUs exploiting custom heuristics and continuously scales CPU cores using control-theory according to applications' needs.
Input:

- TOSCA blueprint with applications description and goals
- QoS metrics
- Available resources

*Output*:

- Load balancing on heterogeneous resources
- Resource Allocation

*Programming languages/tools*: Python, Kubernetes
*Dependencies*:

- Deployment Refactorer
- Orchestrator

[3] Arcelli, Davide, Vittorio Cortellessa, and Catia Trubiani. "Performance-based software model refactoring in fuzzy contexts." International Conference on Fundamental Approaches to Software Engineering. Springer, Berlin, Heidelberg, 2015.
[4] Esfahani, Naeem, Ehsan Kouroshfar, and Sam Malek. "Taming uncertainty in self-adaptive software." Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011.

- Semantic Reasoner
- Monitoring Agent

*Critical factors*: N/A

#### 3.5.1.7 Refactoring Option Discoverer

*Functional Description*: This component is responsible for discovering new refactoring options and changes to existing refactoring options. To select refactoring options, it uses various matchmaking criteria based on the properties, capabilities, requirements, usage policies of resources. For example, a new node that may offer a specific security policy (e.g., the node is placed only on a data center in a given set of regions) or scaling policy (e.g., the node can be autoscaled up to 5 instances).

*Input*: Search (matchmaking) criteria

*Output*: Refactoring options

*Programming languages/tools*: Java

*Dependencies*:

- Deployment Refactorer
- Monitoring Agent
- Semantic Reasoner

*Critical factors*: N/A

#### 3.5.1.8 Rule File Server

*Functional Description*: This component receives requests for registering new alerting rules (or deregistering them) from external clients (e.g. the IDE). Upon reception, rules are registered/unregistered within the main monitoring engine.

*Input*: A monitoring rule description, compliant with the monitoring rule language.

*Output*: A registered monitoring rule (within the main monitoring engine).

*Programming languages/tools*: REST API

*Dependencies*: Rules File Server needs the main monitoring engine, through its MonitoringAPI.

*Critical factors*: Certain anomalous runtime behavior of deployed application components could not be detected if the corresponding monitoring rules are not registered.

### 3.5.2 Sequence Diagrams

For this document, UC6, UC7, UC8 and UC18 have been updated. The other UCs have been copied from D2.2 for completeness. The interaction with IAM and Vault is abstracted in all sequence diagrams and is occurring in accordance with the diagrams presented in Section 3.2.

### 3.5.2.1 UC6: Execute Provisioning, Deployment and Configuration

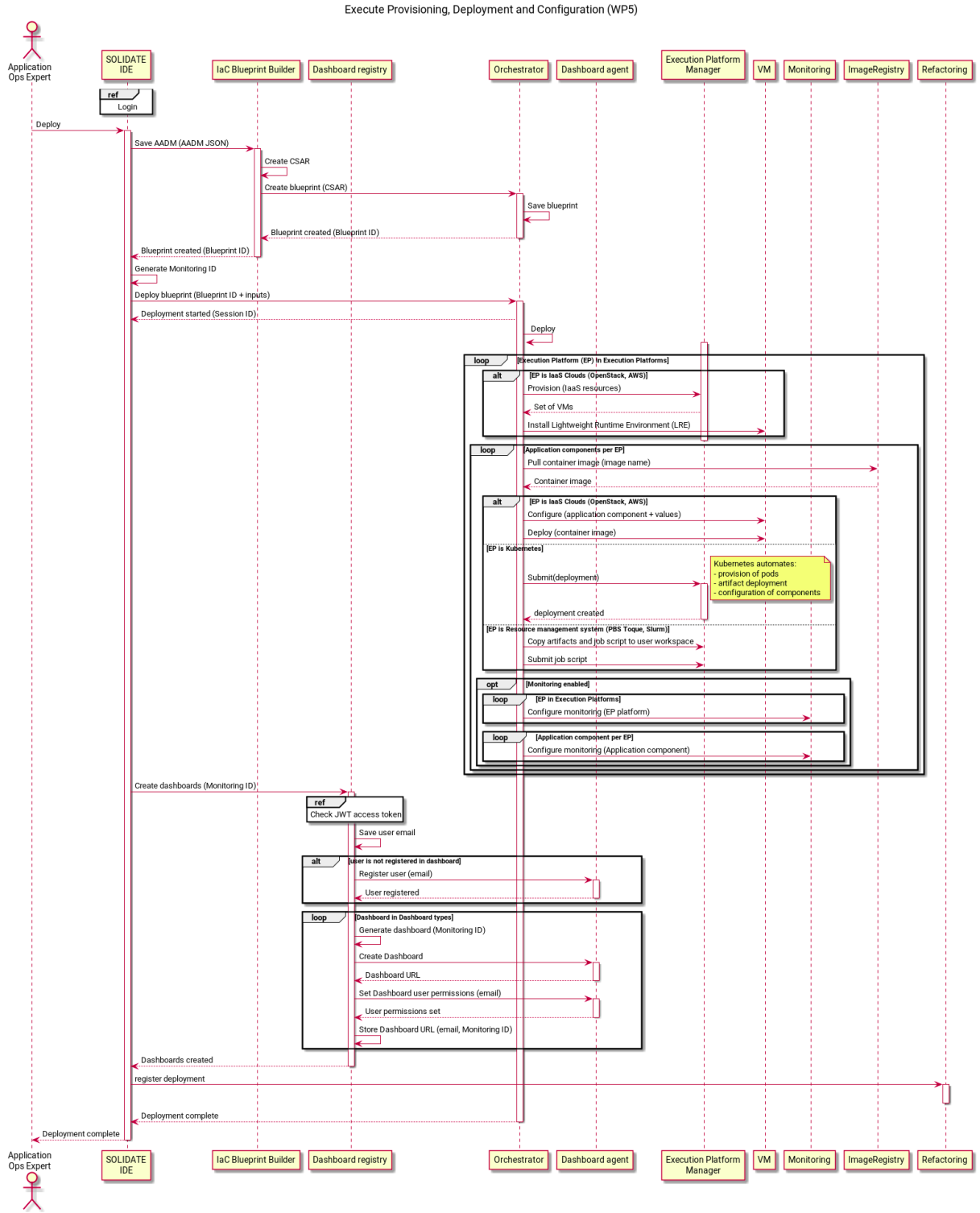



Figure 22 - Sequence diagram for UC6.

Figure 22 describes the interaction between the SODALITE components while implementing the UC6 - Execute Provisioning, Deployment and Configuration. This sequence diagram has been extended from the D2.2 version to include post-deployment interactions, between the IDE and the Refactorer and Monitoring Dashboard Registry, respectively.

Once an AADM (abstract application deployment model) has been defined, an Application Ops Expert initiates the deployment via the SODALITE IDE. The SODALITE IDE provides the AADM to the IaC Blueprint Builder, which in turn creates a CSAR (Cloud Service Archive - an archive that contains TOSCA blueprints and metadata, and the deployment and implementation artifacts) and passes it to the Orchestrator. The Orchestrator saves the CSAR and returns a Blueprint ID to the IaC Blueprint Builder, which is further passed back to SODALITE IDE. At this point, the deployment is registered in the Orchestrator system and can be later referred through its Blueprint ID, e.g. for the deployment execution or deployment updates.

Once the Blueprint ID is received by the SODALITE IDE, it directly requests the Orchestrator to start the deployment and receives a Session ID to monitor the deployment progress. Optionally, a set of inputs can be passed along the request to parameterise the deployment. Then, the Orchestrator starts the deployment by executing the deployment workflow specified in the blueprint.

For each execution platform specified in the blueprint, its resources are instantiated, and the application components are deployed on top of them. As such:

- In IaaS clouds, before the deployment of the application components, the virtual resources must be first provisioned. For that, the Orchestrator issues provision requests to the IaaS resource manager (e.g. OpenStack, AWS EC2) to create a set of virtual machines (VMs) and other resources that the application demands, e.g. security group, network and storage. The Lightweight Runtime Environment (LRE) is then installed on VMs as a runtime for the execution of the application components. Upon the installation, the Orchestrator configures and deploys application components, pulling them from specified image registries.
- Kubernetes automates the resource provisioning and the application deployment by exposing an endpoint, through which the deployment and configuration descriptions are passed. Hence, the Orchestrator submits these descriptions to Kubernetes, which configures and deploys the application components, pulled from specified image registries.
- When resource management systems are selected (e.g. Torque or Slurm), the Orchestrator pre-uploads the artifacts (e.g. pulling required container images) and the job description script to the user workspace (e.g. home directory of the user) on login (front-end) nodes and then submits the job to the batch system. UC7 describes the start of applications, deployed using one of the resource management systems.

At this point, the deployment of the heterogeneous application components is performed on different resources, and the application is started. Optionally, the configuration of a monitoring platform can be additionally performed if such mechanism is requested to the Orchestrator. First, for each allocated Execution Platform (EP), one or more different Monitoring Exporters (i.e. monitoring probes) are registered and configured within the monitoring agent. Then, a similar exporter registration and configuration process is conducted for each application component that requires a specialized exporter.

After a successful deployment, the IDE requests the monitoring dashboard registry to generate the dashboards associated with the new deployed application. The AOE user (taken from JWT token) is registered to the dashboard and user's permissions set to restrict the access to her dashboards. Then, the IDE notifies Refactoring about the new deployment, so that it gets registered for further refactoring surveillance.

### 3.5.2.2 UC7: Start Batch Application

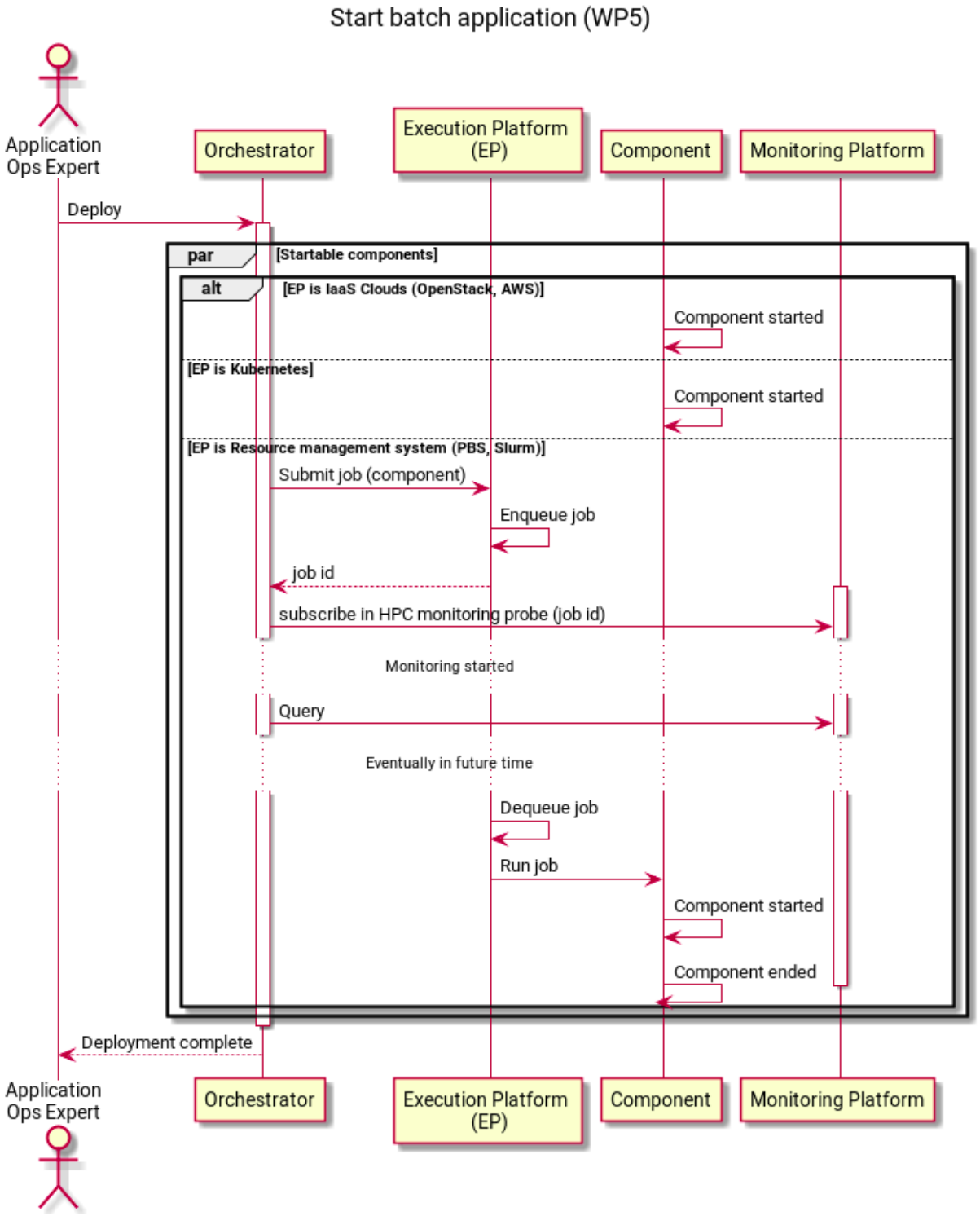



Figure 23 - Sequence diagram for UC7.

Figure 23 describes the interaction between the SODALITE components while implementing the UC7 - Start a batch application. UC7 accompanies UC6 to describe in detail the start of batch applications in HPC infrastructures, i.e. applications that take an input, process it and give the results, unlike the services (e.g. web servers, REST APIs), which start right after deployment and run continuously. Furthermore, these applications can be deployed once and executed several times

with different inputs. This sequence diagram has been simplified from D2.2, removing the IDE as a participant in the process and focusing on the deployment process that takes place in the Runtime Layer.

When a job is submitted to the batch system, it does not start immediately, but it is firstly enqueued to a specified or default queue. It starts once the job's turn comes in the queue after compute resources become available, and it leaves the queue for its execution. It may take some time depending on the resources' availability in the queue.

During the deployment (or redeployment), when the resource management system is selected as an execution platform, the Orchestrator submits a job and monitors its state, whether it is running or finished. When the job is finished, the Orchestrator determines whether the execution was successful or failed. The deployment terminates when the execution is failed, otherwise the deployment continues with the next application component. In the case of HPC scheduling, the precise start time for job execution (after being dequeued) is unpredictable, as it depends on the eventual availability of the requested resources. Therefore, job monitoring must start at queue time, as it is requested by the orchestrator to the Monitoring System, which, at this point, starts collecting statistics describing the application job status.

From then on, the Orchestrator can initiate the collection of metrics for its purposes (e.g., to check application job health).

### 3.5.2.3 UC8: Monitor Runtime

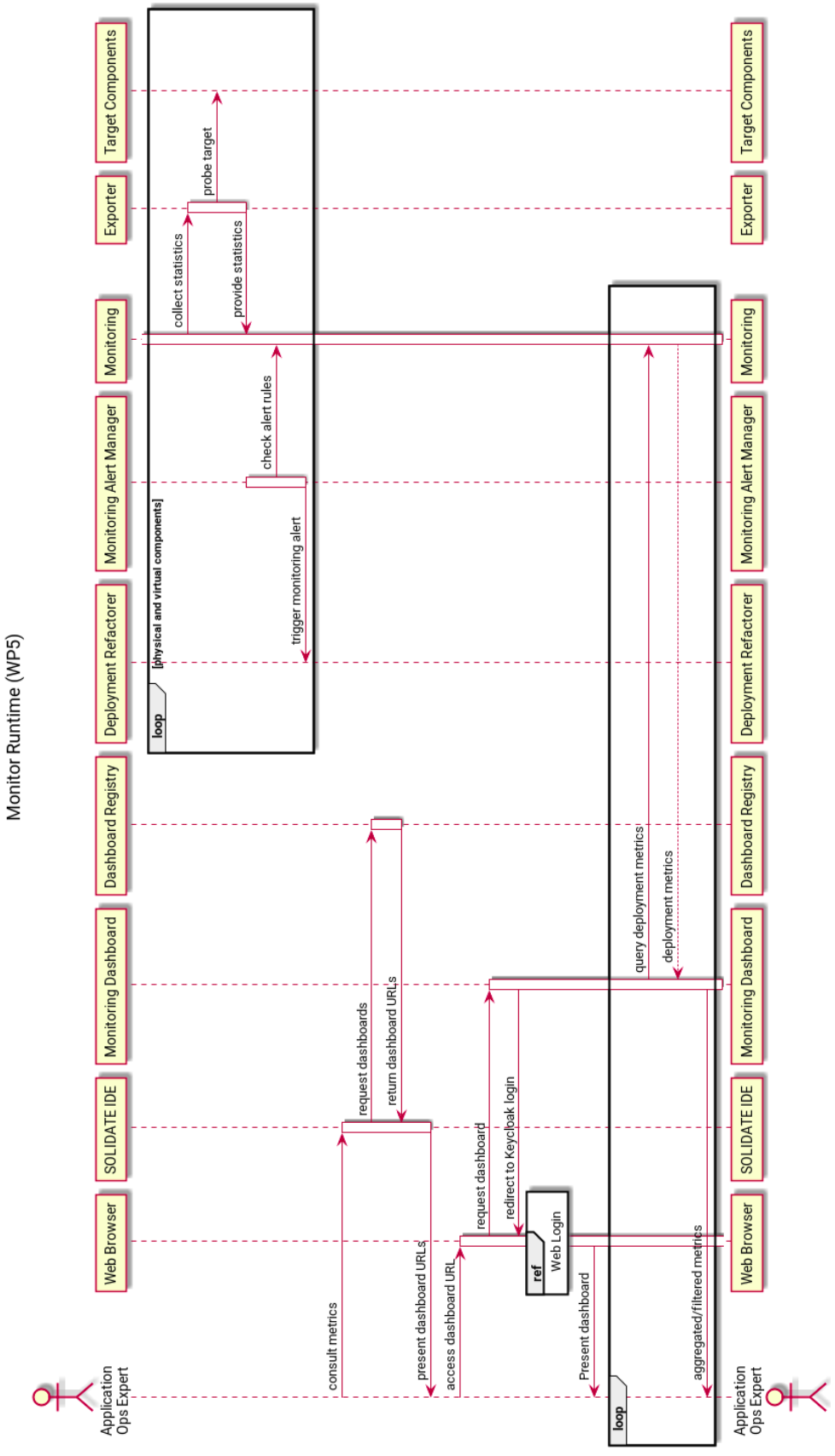



Figure 24 - Updated sequence diagram for UC8

Figure 24 describes the interaction between the SODALITE components while implementing the UC8 - Monitor Runtime. This sequence diagram has been updated from D2.2 to emphasize the restricted (IAM-meditated) access to the monitoring dashboards only associated with AOE's deployed applications.

The Monitor component collects system statistics on an ongoing basis. On each host (whether physical or virtual) there are one or more exporters (i.e. probes) that interact with the Monitoring component and report back to it some standard (but also specialized) statistics about their target, which could be either the execution platform or the application component. Statistics are usually collected on each target by reading various counters and registers that hold updated system statistics.

Periodically, Monitoring collects the statistics from all the registered exporters and stores them into its internal database for further inspection, upon the reception of queries requested from external clients. High level reports on standard statistics are available to be consulted by AOEs in the Monitoring Dashboard that is accessible from the SODALITE IDE. AOEs can only access the dashboard associated with their application deployments.

Some monitoring alerts could also be triggered by the Monitoring Alert Manager to the Deployment Refactor, for those situations where a condition (defined within a dynamic alerting rule) holds on concrete monitoring stats.

In such cases, the Deployment Refactor component could make placement decisions based on the resource usage. In other cases it may be desirable to inspect some specific non-standard monitoring figures in order to isolate the cause of some observed anomaly. In this case, the AOE can request such a report by using the Monitoring Dashboard component, which, in turn, creates and issues to the Monitoring component all the queries required to create the report. Using the results to those queries returned by Monitoring, the dashboard presents the aggregated report to the AOE.

### 3.5.2.4 UC9: Identify Refactoring Options

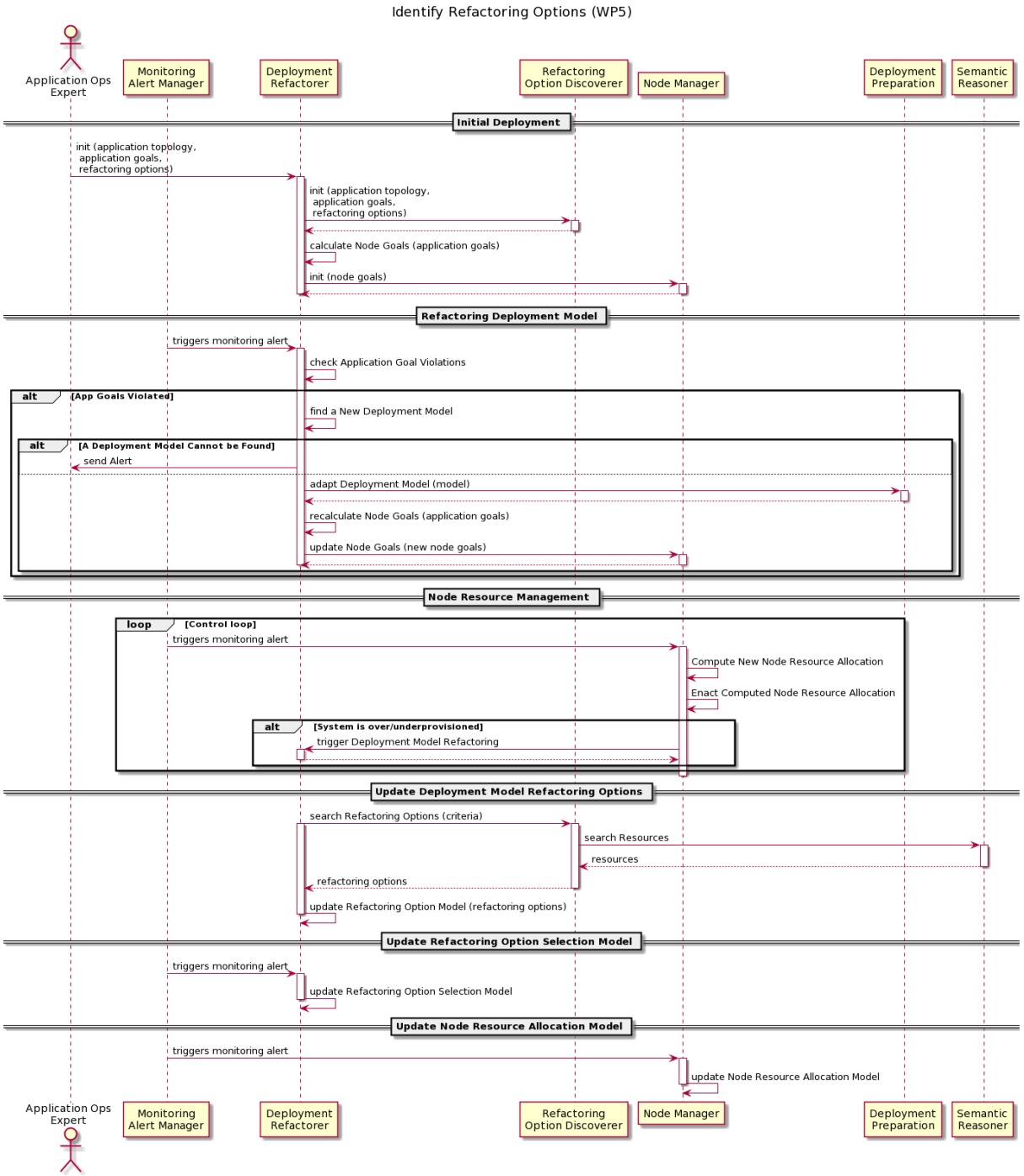



Figure 25 - Sequence diagram for UC9.

Figure 25 describes the interaction between the SODALITE components while implementing UC9 - Identify Refactoring Options. The 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 the Monitoring Alert Manager, the Deployment Refactorer is eventually notified when any of the defined application goals are violated. Upon such circumstances, the 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, the Application Ops Expert is alerted. The new deployment is enacted via the Deployment Preparation API. The Deployment Refactorer also may reassign node-level goals as necessary. The Refactoring Option Discoverer can find the new refactoring options as well as the changes to the existing refactoring options. It uses the Semantic Reasoner for this purpose. Both the Deployment Refactorer and the 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.

#### 3.5.2.5 UC10: Execute Partial Redeployment

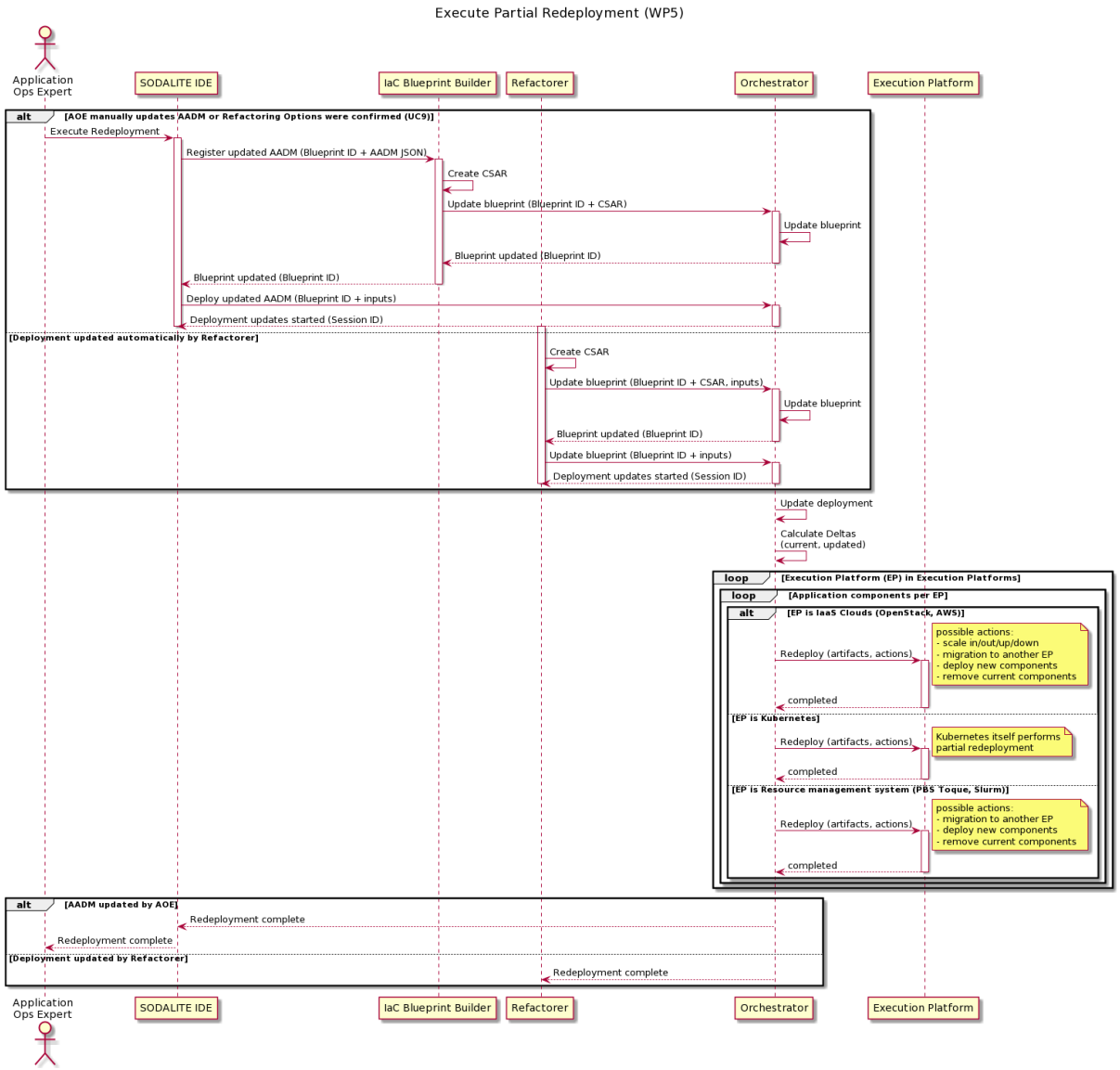



Figure 26 - Sequence diagram for UC10.

Figure 26 describes the interaction between the SODALITE components while implementing UC10 - Execute Partial Redeployment. A redeployment (or deployment updates) is an act of modifying application topology or application parameters at runtime. The partial redeployment refers to

updates of only affected components of an application, i.e. those components that require modifications during the runtime of the application as opposed to the modification of the whole application topology. As an example, a certain application component may need the updates of the container image or it may require scaling to improve the performance. Another example might be the changes in application topologies, when new components are introduced or old components need to be removed.

In SODALITE, partial redeployment can be triggered via the SODALITE IDE when the Application Ops Expert manually changes an AADM or he/she confirms one of the Refactoring Options suggested by Refactorer as described in UC9. Refactorer may also be configured to automatically request a redeployment if it detects any Refactoring Options at runtime. In both cases, a reference to a particular Blueprint ID, which is obtained after execution of UC6, should be used.

When a redeployment is triggered manually by the Application Ops Expert, the SODALITE IDE requests IaC Blueprint Builder to register a new CSAR from the updated AADM and to redeploy the blueprint in the Orchestrator. After that, SODALITE IDE directly requests the Orchestrator to start the deployment updates and it receives a Session ID to monitor the redeployment progress.

Alternatively, when a redeployment is triggered automatically by Refactorer, it creates a CSAR, which reflects the updated deployment, and updates the blueprint in the Orchestrator. Similarly, it then directly requests the Orchestrator to start the deployment updates and receives a Session ID to monitor the redeployment progress.

At this point, the deployment updates are executed by the Orchestrator. 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 IaaS Cloud or Kubernetes, 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.

It should be noted that Kubernetes, being itself an orchestration platform, is enforcing partial redeployment.

For resource management systems (common in HPC), these Execution Platforms lack flexibility in scaling at runtime, hence the scaling actions are not present as possible adaptation actions; however, all the other actions can be executed as well (migration, deployment and removal of components).

### 3.5.2.6 UC18: Deployment Governance

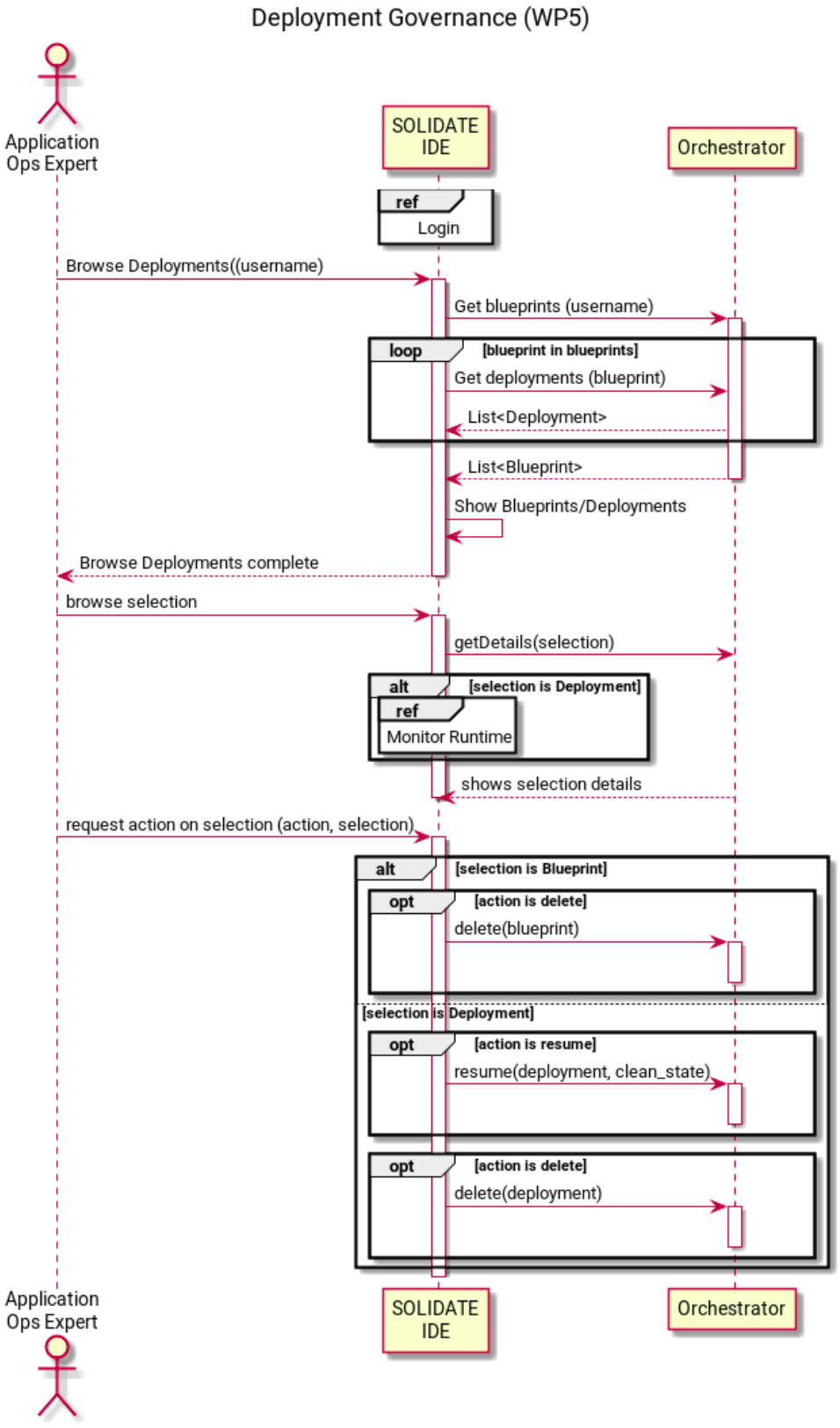



Figure 27 - Sequence diagram for UC18.

Figure 27 describes the interaction between the SODALITE components while implementing the UC18 - Deployment Governance. This new UC describes the management, mediated through the IDE, of AoE's application deployments. A logged AoE can browse her deployed applications retrieved from the runtime Orchestrator. For every AoE's application, or blueprint in Orchestrator terminology, there could be one or more deployments. AoEs can select an application (blueprint) or one of its deployments and browse its details. In the case of a deployment, AoEs can additionally access the associated monitoring dashboards to browse runtime monitoring metrics (see UC8). Several actions are supported depending on the selection (e.g. either a blueprint or a deployment). In case of selecting a failed deployment, it can be resumed (from the first failing node or from an initial state, depending on the clean_state boolean parameter). Any selection can also be deleted. In case of blueprints, deletion is only possible when it does not own any deployment.

# 4 Evaluation plan and KPI accomplishment

The SODALITE evaluation plan is articulated around the following actions:

- The operationalization and measurement of the technical KPIs defined in the Grant Agreement, in order to demonstrate the incremental accomplishment of such KPIs and take recovery actions where needed.
- The evaluation of the SODALITE platform carried out by the case study owners through the adoption of the SODALITE approach. This evaluation is continuous and has led, through the project execution, to the identification of new requirements and use cases that have extended the SODALITE functionality.
- The quality control processes put in place during the project to achieve higher quality code.

In the following sections we elaborate on the above aspects.

## 4.1 Operationalization and measurement of KPIs

The technical KPIs defined in the Grant Agreement had the objective of providing a tool to assess the level of accomplishment of the project goals at key milestones in the project. In Deliverable D2.2, we have defined the operationalization of such KPIs, that is, the metrics and the measurement process through which such KPIs can be measured. The first measures have been taken at M24 as planned and concerned both those metrics that were planned to be assessed at that time and also those that were planned for a following project milestone. The results of this measurement process have been reported in Deliverable D6.3 and are summarized in Table 1, taken from D6.3.

Table 1 - Summary of technical KPI status at M24.

| KPI | Description/Target | Due by | M24 status |
|---|---|---|---|
| **KPI 1.1** | This KPI refers to the capability of the modelling layer to support the so far defined use cases in terms of abstract application and infrastructure structures. **Target: 25% coverage** | M24 | 66% coverage |
| **KPI 1.2** | The KPI refers to performance patterns found in the demonstrating use case. We use the use case requirements to map this and calculate the lower bound. **Target: 80% coverage** | M33 | Clinical UC: 80% coverage, Snow: 92% coverage, Vehicle IoT: Not relevant |
| **KPI 1.3** | The KPI refers to execution constraints on compute, memory, network and storage and possibilities found in the demonstrating use case. We use the use case requirements to map this and calculate the lower bound **Target: 80% coverage** | M33 | Clinical UC: 66% coverage, Snow: 83% coverage, Vehicle IoT: TBM |
| **KPI 2.1** | When AOE exploit abstractions in their application code, we expect an increase in performance of 15% | M30 | Clinical UC: 3% speedup increase, Snow: 10% speedup increase |

| | | | |
|---|---|---|---|
| | **Target: 15% speedup increase** | | |
| **KPI 2.2** | The KPI refers to when the deployment of an application is dynamically optimized with respect to changing workloads to improve resource usage, and to reduce SLA violations and component failures. **Target: 20% improvement over the baseline** | M30 | Node Manager: 96% reduction in SLA violations; 15% reduction in allocated resources<br>Deployment refactoring: 96-99% accuracy and efficiency of performance prediction for deployment alternatives |
| **KPI 3.1** | The focus will be on development of deployment descriptions, not application code. **Target: 10% improvement over the baseline** | M24 | 28% improvement for a TOSCA expert |
| **KPI 3.2** | This reduction is specifically concerning resource management. **Target: 30% improvement over the baseline** | M24 | 19% improvement for a TOSCA expert |
| **KPI 4.1** | Integration of the SODALITE system allows for combined use of all its components. **Target: 95% component compatibility** | M33 | 100% component compatibility, specific features to be improved integration-wise |
| **KPI 5.1** | LOC released as open source / LOC produced by SODALITE to build the platform. **Target: 80% open source code** | M36 | 100% open source code |
| **KPI 5.2** | The meaning is the following: given the subset of the SODALITE code that is built extending an existing open source project, 60% of this code is donated back to the existing project. To check the fulfillment of this KPI we need to identify the reference projects we extend. **Target: 60% of upstreamed code** | M36 | 96% of code submitted to upstream projects has been merged, a further 3% has been submitted but not yet merged. |

As can be seen from the table, almost all KPI targets were met already at M24. In Deliverable D6.4 we will present the results of the new evaluation campaign that is being conducted in the last part of the project, in order to assess the KPIs at M33 and M36. Given KPIs KPI 2.1 and 2.2 are the only KPIs planned to be evaluated at month 30, while deliverable D6.4 will come later, we want to anticipate some values here. As for KPI 2.1, we have measured a speedup of 18% for Clinical UC and a speedup of 20% with Snow UC. We can also confirm the values already presented for KPI 2.2,

that is, 96% reduction in SLA violations for Node Manager and 15% reduction in allocated resources. As for deployment refactoring: 96-99% accuracy and efficiency of performance prediction for deployment alternatives.

As part of the development of this deliverable, we have reconsidered the evaluation process and the metrics defined at M24 in D2.2 and we have concluded that they are still fully applicable and adequate for the M33 and M36 evaluation.

Moreover, considering the stronger integration of the design time and runtime elements of the SODALITE platform, we have decided to extend the scope of the controlled experiments used for assessing KPI 3.1 (Reduction in software and/or application development time and cost) and KPI 3.2 (Reduction in software management (redeployment, reconfiguration) time and cost). While in the initial experiments involving external users we limited the usage of the SODALITE framework only to the IDE, leaving the usage of the entire platform only to the case study owners, this time we plan to involve also a limited number of external users in the exploitation of the whole framework. More specifically:

For what concerns KPI 3.1, external users and TOSCA experts will be asked to model the AADM of a simple application, to generate the TOSCA code themselves by running the IaC blueprint builder and to execute this code through the orchestrator to test its correctness (in the initial experiment, they were only focusing on the modeling activity and the SODALITE team was checking the correctness of their work offline).

For what concerns KPI 3.2, we will ask our subjects to generate a new version of an AADM, to experiment with the versioning features and to perform the generation and execution steps as described above.

## 4.2 Evaluation of the platform by the case study owners

An important element of the SODALITE evaluation approach is based on the usage of the platform by the case study owners. Table 2 identifies the features (UML UCs) exploited by the case studies in the first project year (in gray) and those adopted in the second project year (in green). Moreover, it draws the plan for the last year of the project. In this new year, a new UML UC has been defined (UC18) and this will be exploited by all case studies. Moreover, all other UCs will be experimented again, given that they have been extended and consolidated in the last period.

Table 2 - Coverage of the SODALITE UML use cases by the demonstrating use cases by M36

| Use Case | Virtual clinical trial | SNOW | Vehicle IoT | Testbed Providers |
|---|---|---|---|---|
| UC1 Define Application Deployment Model (WP3) | | | | |
| UC2 Select Resources (WP3) | | | | |
| UC3 Generate IaC code (WP4) | | | | |
| UC4 Verify IaC (WP4) | | | | |
| UC5 Predict and Correct Bugs (WP4) | | | | |
| UC6 Execute Provisioning, Deployment and Configuration (WP5) | | | | |
| UC7 Start Application (WP5) | | | | |
| UC8 Monitor Runtime (WP5) | | | | |
| UC9 Identify Refactoring Options (WP5) | | | | |
| UC10 Execute Partial Redeployment (WP5) | | | | |
| UC11 Define IaC Bugs Taxonomy (WP4) | | | | |

| | | | | |
|---|---|---|---|---|
| UC12 Map Resources and Optimisations (WP3) | | | | |
| UC13 Model Resources (WP3) | | | | |
| UC14 Estimate Quality Characteristics of Applications and Workload (WP3) | | | | |
| UC15 Statically Optimize Application and Deployment (WP4) | | | | |
| UC16 Build Runtime images (WP4) | | | | |
| UC17 Platform Resource Discovery (WP4) | | | | |
| UC18 Deployment Governance (WP5) | | | | |

| Y1 | Y2 | Y3 |
|---|---|---|

## 4.3 Code quality control processes

As discussed in Deliverables D2.4 and D2.2, the SODALITE consortium has put in place a quite rigorous process to control the release of new code, the execution of code analysis aimed at discovering major problems in the source code, the automatic execution of tests, the generation of the Docker images associated to each component and their upload on Docker Hub. We have additionally strengthened the quality control process by paying specific attention to security, which is clearly a key element of any piece of modern, interconnected software. Therefore, the consortium has adopted security as one of its core quality metrics.

### 4.3.1 Tools

The consortium has adopted a number of tools to ensure the production of secure and high quality software, in addition to those which are detailed in deliverable D2.2 Section 5.2. Previously, the consortium adopted Sonar and sonarcloud.io for the continuous assessment of code quality. Sonar includes a number of tools, only some of which are relevant to security. These include:

- “Security hotspots”, which examines code for common security issues and identifies code that must be checked manually to determine if there is a security issue.
- “Code smells”, which examines code for common anti-patterns and identifies code which needs further analysis and/or specific corrections to improve the security, organization or quality of the code.

However, Sonar does not cover the full security lifecycle. It is focused solely on the quality of the code.

The consortium has chosen to package all of the components in Docker containers. The Docker ecosystem introduces additional security considerations, such as:

- Software installed in the container to support the Consortium's component may be out of date. When the consortium builds a Docker container, certain supporting software must be added to the container. This dependency software may have security issues and subsequently fix the issues. If the consortium does not update the containers it builds, they will continue to use the out of date dependency software and therefore have security vulnerabilities.
- Software in the base container may be out of date. If this happens, the Consortium's container may have a security vulnerability that is not part of the Consortium's code. The consortium is not responsible for fixing the issues in the base container. However, the consortium is responsible for selecting an appropriate base container. The security of the base container is an important decision in this selection.

The consortium has sought software to detect these issues as well as other security issues. After a search process, Snyk (snyk.io) has been selected as a monitoring and alerting tool for security issues. The consortium has been using three primary features of Snyk:

- "Open Source Security", which scans the dependencies of the components and alerts the consortium when one is known to have security issues.
- "Container security", which scans Docker containers and determines if any of the base containers has security vulnerabilities.
- "Static analysis", which examines component code and looks for common security issues.

When Snyk finds such an issue, it alerts the consortium. Then the consortium can then apply corrective measures.

### 4.3.2 Possible Metrics

Snyk's results come in the form of a number of issues and, for each issue, a severity rating. For example, a particular component might have 1 Critical Severity issue, 3 High Severity issues, 10 Medium Severity Issues, and 2 Low Severity issues. In industry, it is common to follow "Application Security Policies", which define the policies and procedures used for checking for and remediating security issues. These will define when security checks must be run, what delay is acceptable in remediating them and when new releases of software must be made in order to update dependencies.

For software written internally, Application Security Policies often consider the type and severity of the issue. Additionally, when dependency software is involved, policies will consider if an update to the dependency is available.

The consortium has not yet selected metrics to use with Snyk. The consortium intends to select metrics similar to the following examples. However, these are examples and the final selected metrics may be different:

- Zero critical and high severity issues in consortium-written code at the time when a component is released, unless a written explanation is provided explaining why the issue cannot be resolved.
- Zero critical and high severity issues in dependencies when a component is released, unless no updated version of the dependency is available or a written explanation is provided explaining why the issue cannot be resolved.
- Re-release of released containers within 7 days if a critical security issue is found and within 14 days if a high severity issue is found in dependencies included in the container.
- 50% reduction in issues of all severities.
- Use of base containers with the fewest possible critical and high severity issues.

### 4.3.3 Open Issues

The consortium has not completed the integration of Snyk. The following issues are still "open" and must be resolved before Snyk is fully integrated:

- Integration of Snyk with Jenkins. The consortium wishes to integrate Snyk with the existing Jenkins pipeline used in the project to support the continuous integration and deployment activities and to favor automation. However, on one hand Snyk produces a list of security issues, some of which the component authors would not be able to resolve if the problem is detected on an updated software dependency. On the other hand, Jenkins can only produce a binary result (i.e., component accepted or not accepted). Thus, the consortium must:
  - Determine if it is possible for Snyk to return only issues the component authors are able to correct.
  - Determine if Snyk can ignore known issues that are not currently resolved.
- Set metrics for which test results from Snyk are and are not acceptable.
- Selection of metrics. The consortium must finalize the metrics to be used with Snyk.

- Creation of an Application Security Policy.

# 5 Conclusions

This document presented the last update of the work done on requirements, architecture, and KPIs. This is the final version of the series and collects all the work done, with special emphasis on what has been done over the last (third) year of the project. The document collects all requirements and summarizes their status (those fully implemented, those cancelled, and those for which we managed some exceptions). It also provides an up-to-date version of the architecture of the SODALITE environment, with special emphasis on the work done on authentication and authorization and on the changes implemented in the last year of the project. The architecture described here complies with Milestone MS7 (*Final Architecture*). Finally, the section on KPIs recalls all the KPIs we promised and checked, updates the evaluation plan, and discusses the work done in this last year on the automated assessment of the security of produced Docker images.