MULTICLOUD APPLICATIONS TOWARDS THE DIGITAL SINGLE MARKET

Deliverable D2.4

Integrated architecture v1

Editor(s):	Juncal Alonso
Responsible Partner:	TECNALIA
Status-Version:	Final - v1.0
Date:	30/11/2017
Distribution level (CO, PU):	PU

Project Number:	GA 731533
Project Title:	DECIDE

Title of Deliverable:	D2.4 – Detailed architecture v1
Due Date of Delivery to the EC:	30/11/2017

Workpackage responsible for the Deliverable:	WP2 – DECIDE requirements and DECIDE solution integration
Editor(s):	TECNALIA
Contributor(s):	Juncal Alonso, Gorka Benguria, Marisa Escalante, Maria Jose Lopez, Iñaki Etxaniz (TECNALIA), Luis Miguel, Javier Gavilanes, Gema Maestro (Experis), Lorenzo Blasi, Paolo Barone (HPE), Lena Farid, Majid Salehi,Simon Dutkowski (Fraunhofer), Anna Mikhailova, Andrey Sereda(CB)
Reviewer(s):	TECNALIA
Approved by:	All Partners
Recommended/mandatory readers:	WP3, WP4, WP5

Abstract:	This deliverable will contain the first version of the detailed design of DECIDE framework: its components, modules, interfaces.
Keyword List:	Architecture, components, technical design, interfaces, interoperability, deployment, DevOps.
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and the Commission is not responsible for any use that may be made of the information contained therein

Document Description

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	21/07/2017	First draft version including TOC	TECNALIA
v0.2	07/09/2017	Sections assignments agreement.	TECNALIA
V0.3	21/09/2017	Included content in sections: 2.4, 3.1.1, 3.3.1, 3.3.3.	TECNALIA
V0.4	09/10/2017	Included content in sections: 2.1, 2.2, 2.4, 2.5, 3.1.2, 3.3.1, 3.3.2, 3.4.2, 4 (draft text with the main ideas). Included comments in section 3.3.3.	Fraunhofer, TECNALIA, CB.
V0.5	16/10/2017	Included content in sections: 2.5, 3.4.1.	HPE.
V0.6	16/10/2017	Included content in sections: 2.3, 3.2.1.	Experis
V0.7	18/10/2017	Included content in sections: 1, 2.1. Included new annex	TECNALIA
V0.8	23/10/2017	Included content in sections: 2 (picture), 4, 5.	TECNALIA
V0.9	24/10/2017	Included content in section 4. Included references (section 6). Included terms and abbreviations Adapted content so that all the sections cover equivalents ideas.	TECNALIA
V0.10	24/10/2017	Updated content in Annex 1: App description.	HPE
V0.11	30/10/2017	Added sub-section in section 2 (2.6- Sample multi-cloud application in DECIDE: Sock shop application) Updated content in section 3.3.3 Updated content in section 4 Updated content in Annex 1	TECNALIA
V0.12	16/11/2017	Updated content in Annex 1 Updated content in section 2.5, 3.3.2.	TECNALIA, HPE, Fraunhofer.
V0.13	17/11/2017	Updated content in Annex 1	TECNALIA, Experis.
V1.0	21/11/2017	Internal review comments addressed	TECNALIA, Fraunhofer.

Table of Contents

List of Figures

List of Tables

Terms and abbreviations

API	Application Programming Interface
App	Application
APP	Application
CSP	Cloud Service Provider
DB	Data Base
DevOps	Development and Operation
DoA	Description of Action
EC	European Commission
IDE	Integrated Development Environment
KR	Key Result
M12	Month twelve
M23	Month twenty-three
MCSLA	Multi Cloud Service Level Agreement
NFR	Non-Functional Requirement
RCP	Rich Client Platform
SDK	Software Development Kit
SLA	Service Level Agreement
Sw	Software
UI	User Interface
WP	Work Package

Executive Summary

This document provides the specification of the DECIDE integrated architecture at month 12, a first specification[1] of the entire DECIDE integrated architecture, aiming to ensure a smooth integrated specification at conceptual, functional and technical level of the different building blocks, i.e. Key Results, that constitute the DECIDE tool-suite. DECIDE tool-suite enables users (developers and operators) of multi-cloud applications to implement the DECIDE extended DevOps approach, by providing a comprehensive set of tools that assists users to complete the DECIDE lifecycle [1].

DECIDE comprises diverse technical and scientific activities that contribute altogether to the jointly materialization of the DECIDE outcomes. However, a successful instantiation of these techniques and tools requires an integration task force, materialized in this document, which aims to converge these different conceptual and technical approaches and the earlier detection and fixing of potential misalignments that may occur during the initial design and development phases.

In this scope, the global and integrated architecture described in this document aims to: a) provide an overall and comprehensive conceptual and functional description of the DECIDE tool-suite in their current state, b) ensure a smooth conceptual and technical interoperability among DECIDE tools that guarantees a correct instantiation of the DECIDE extended DevOps approach, c) describe the tools interoperability needs in terms of messages consumed and provided by each tool, d) provide a detailed structural and behavioural view of the different tools, grouped on packages of related functionality (with respect to the DECIDE extended DevOps approach, and e) expose and discuss the available possibilities for the deployment of DECIDE tool-suite. The current document is describing at general level the different components inside the DECIDE tool suite, the details of the components are described in the deliverables to be generated in the different technical WPs (WP3, WP4 and WP5).

To conclude, this document will be continuously checked, used, aligned and updated as a result of other DECIDE activities during the design and development of their components and tools (a second version of the deliverable will be delivered in M23), to ensure that they are conceptually and technically aligned and compatible with others as these may need to interoperate with.

[1] Final specification will be released on M23

1 Introduction

1.1 About this deliverable

This document provides the M12 specification of the DECIDE integrated architecture. This architecture collects and describes the main functional key results, tools and components that constitute the DECIDE tool-suite, that is, the comprehensive set of tools created by DECIDE, for developers and operators of multi-cloud applications to apply the DECIDE DevOps extended approach.

DECIDE Key Results (and related tools and components) are described as functional blocks, including structural and behavioural aspects. The descriptions of the components included in this document aim to provide a general overview of the functionalities of the key results and the interactions between them. The internal representation of the tool (i.e. its internal technical specification) is not addressed in the document but left to further dedicated technical reports (for each tool) that will describe them, along with the actual implementation in the different releases [2].

This overall description aims to prove a complete and correct coverage of the DECIDE extended DevOps approach, providing means (in form of tools and components) to support each phase of the approach.

Special attention is also given to address interoperability between tools, that is, the dependencies between tools and the technical ways in which those dependencies are managed. Interoperability implies taking care of different dimensions and which allows us to identify and define:

- Messages exchange, including compatibility at data content (semantic alignment), data format, serialization format, etc.
- User-driven interaction model, as we foreseen most of interoperable situations driven by end-users of the tools.

This analysis enables to obtain an earlier detection of possible conceptual and technical misalignments (among tool providers), either at conceptual level (i.e. semantics), or at functional and a technical level. Based on this analysis, the document proposes a harmonized conceptual, functional and technical common view that removes these misalignments and enables the specification (by each tool provider) of an interoperable tool design. In particular, the dependencies amongst tools are identified by detecting the products they consume (as inputs) and produce (as outputs), which would be in turn produced and consumed by other tools.

The document also presents and discusses the different possibilities for the deployment of the tools to be implemented in DECIDE and for the whole DECIDE Framework.

1.2 Document structure

This document is structured as follows.

Section 2 provides an overall conceptual and functional introduction to the entire DECIDE tool-suite, an introduction to the DECIDE multi-cloud concept and to the DECIDE proposed extended DevOps approach. Section 3 provides a detailed description of each DECIDE tool, individually and on the scope of the interactions with other tools in DECIDE tool-suite, structural (i.e. component dependencies, required and provided interfaces, etc.) and behavioural (i.e. temporal ordered interactions), attending mainly to interoperability concerns. Section 4 describes the deployment alternatives for the DECIDE tools, both as individual components and as an ecosystem as a whole. Section 5 shows up some conclusions of the document.

In the Appendix of the document, the current version of the App Description, which is the main mechanism for the interchange of information between tools in DECIDE, is presented.

2 Overview of the DECIDE integrated conceptual architecture

This section sketches the structural view of the DECIDE tool-suite architecture. More elaborated behavioural and structural views of the DECIDE tool-suite architecture will be presented in next sections.

This DECIDE architecture sketch (figure 1) is structured in blocks that correspond to the main DECIDE key results and are co-located extended DevOps phases for multi-cloud applications as defined in [1], i.e. . 1- Design and development, 2- continuous integration and testing, 3- pre-deployment, and 4- continuous operation, to introduce the DECIDE KRs. Some of the KRs (i.e. ACSmI) support several phases of the DECIDE DevOps extended approach. In the figure below two elements are depicted in a different way in order to stress their singularity:

- DevOps framework: DevOps framework is one of the KRs of DECIDE. It is singular because, on one hand, it includes the necessary existing tools for development and integration (i.e. software repository, software development Kit, IDEs, etc.) and on the other hand, it provides the means to integrate the rest of the KRs in a unique stable toolkit (UIs, actions handling, etc.). More information about the DevOps framework is provided in section 3.2.1.
- App Description: Application Description (App Description from now on) is not a DECIDE KR, tool or component as such. It is a file where the actual status of the application is described. This file is used for the different KRs in DECIDE to store/acquire information about relevant information with respect to the application needed to the correct operation of the different tools. More information about the Application Description is provided in section 4 and in Annex 1.

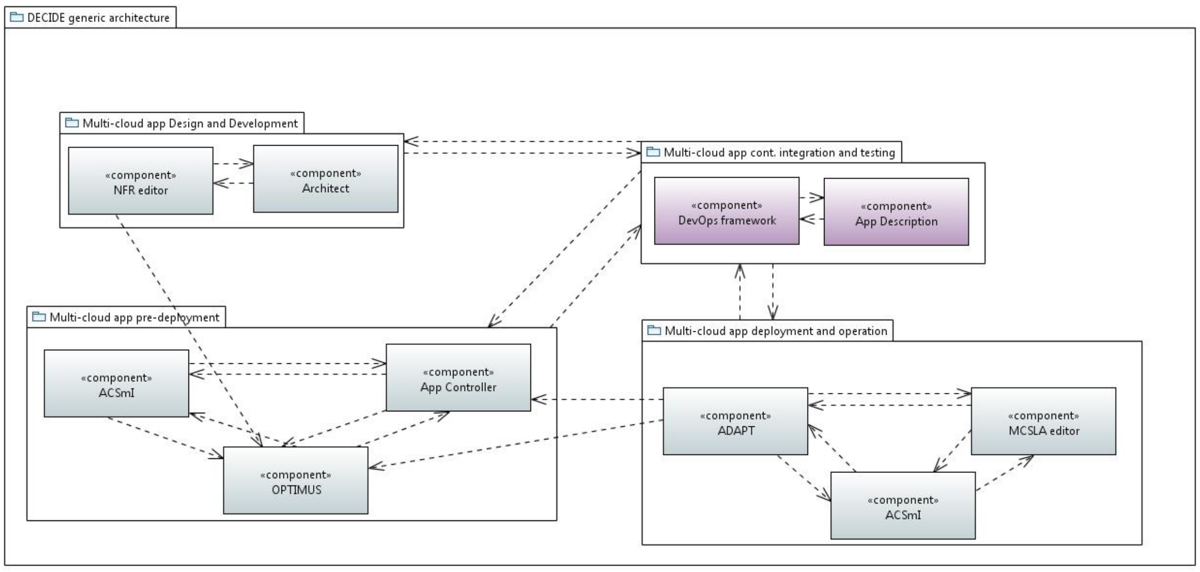
DECIDE generic architecture
Multi-cloud app Design and Development
«component» NFR editor
«component» Architect
Multi-cloud app cont. integration and testing
«component» DevOps framework
«component» App Description
Multi-cloud app pre-deployment
«component» ACSmI
«component» App Controller
«component» OPTIMUS
Multi-cloud app deployment and operation
«component» ADAPT
«component» MCSLA editor
«component» ACSmI

Figure 1. DECIDE integrated generic architecture.

2.1 Multi-Cloud classification

In the context of DECIDE a multi-cloud application is defined as a set of components distributed across heterogeneous cloud resources but that still succeed in interoperating as a single whole.

As described in D2.1 [1], multi-cloud is the use of multiple computing services for the deployment of a single application or service across different cloud technologies and/or Cloud Service providers. This may consist of PaaS, IaaS and SaaS entities in order to deliver an integrated end to end solution

This definition of multi-cloud, when referring to the resources where the different components are deployed, includes services which are in disperse cloud providers or different cloud platforms (regardless of vendor) [3]:

- Deployment of services across multiple geographically dispersed cloud service providers.
- Deployment of services across different cloud technologies within a single cloud service provider.
- Deployment of services within a single cloud service provider in one technology.

2.2 DECIDE Tools for multi-cloud applications design and development

2.2.1 NFR editor

NFR editor is the component where the developers can state the NFR they want to consider during the development and operation of the multi-cloud application. The NFRs can be qualitative, quantitative or both. In the context of DECIDE action the NFRs are closed to the following characteristics: Availability, cost, location, security (legal), performance, scalability but they can be extended with new ones when needed (NFR editor will provide means for this).

The selected NFRs that will be considered during the whole DECIDE process:

1. During the design phase, for proposing the most appropriate architectural design for complying the selected NFRs;
2. In the pre-deployment phase for performing the simulation with the objective of fulfilling the NFRs;
3. In the continuous operation phase for monitoring and assessing that the selected NFRs are being fulfilled at run-time.

2.2.2 ARCHITECT

The ARCHITECT tool consists of a catalogue of architectural patterns. These patterns serve for the optimization, development and deployment of applications to become multi-cloud aware. The idea is to present the developers with a set of patterns to apply to their code. These patterns will be suggested to the developers based on the selected and prioritized NFRs as well as on additional data concerning the application. The ARCHITECT tool is to be used by the developers at their discretion in the design phase. The ARCHITECT tool is also closely related to the development phase. as the patterns suggested provides a description of how these patterns can and may be applied to the source code.

2.3 DECIDE Tools for multi-cloud applications continuous integration and continuous testing

2.3.1 DevOps framework

The DevOps framework is the integration point for all DECIDE tools and Key Results. It provides three main functionalities:

1. It serves as entry point to DECIDE. A user wishing to utilize the tools will do so through the DevOps framework.
2. It integrates the different tools and KRs. It provides access to them, a UI to check information about the projects (microservices data, metrics, SLAs violations, etc.) and centralizes the UIs of all tools.
3. It orchestrates the workflow. The DevOps framework will launch the appropriate tool for each phase of the application's lifecycle.

2.4 DECIDE Tools for multi-cloud applications (pre) deployment

2.4.1 OPTIMUS

OPTIMUS deployment simulation tool will be responsible for evaluating and optimizing the non-functional characteristics from the developer's perspective considering a set of provided cloud resources alternatives. OPTIMUS, working with the continuous deployment supporting tool (DECIDE ADAPT), will provide the best possible deployment application topology, based on the non-functional requirements set by the developer, automating the provisioning and selection of deployment scripts for multi-cloud applications.

2.4.2 App Controller

The functionality of the Application Controller as understood by the DECIDE consortium should reflect the status and state of the application and connect this application status and state with the DECIDE tools in the sense of enabling each tool to understand its corresponding fulfilments.

This functionality has been conceptualised and solutions for various components and parts of the Framework have been introduced. These are:

- Application Description (JSON File) (see Section 4) – is an information model specific to the DECIDE Application and is stored in a repository (e.g. Git) to be accessed by each tool. Each tool shares needed information by pushing and pulling from a dedicated repository. With this solution, an interoperable mechanism has been introduced. Furthermore, no running service is required, which limits a single point of failure and allows the tools to function individually.
- Application Manager (see **¡Error! No se encuentra el origen de la referencia.**) is a reusable component and holds the logic for the Application Description, i.e. the model. It also allows reading and writing from the code repository.
- Triggering the tools depending on the current state of the application and the order of the workflow, via e.g. a Continuous Integration (CI) tool. (This will be sketched out in more detail in year 2 of the project).

Furthermore, the Application Controller component assists in managing the knowledge regarding the currently used deployment configuration and historical ones. It keeps records whether a deployment configuration was successful and if any SLA violations had occurred in the application operation time. With this information, OPTIMUS is able to suggest new and adequate deployment configurations. This is described in more detail in Section **¡Error! No se encuentra el origen de la referencia.**. In the next s tage of the project (year 2), the Application Controller will attain additional functionalities as described in the DoA [2].

2.4.3 ACSmI

The Advanced Cloud Service (meta-) Intermediator (ACSmI) will provide means to assess continuous real-time verification of the cloud services non-functional properties fulfilment and legislation compliance enforcement. ACSmI will be solution-centric as it will be able to discover services from a range of services available in a service registry, always making sure that the best combination for the user ((i.e. OPTIMUS and ADAPT)) is met, while ensuring the integrity and security of the overall ACSmI solution. ACSmI will also be able to ensure the governance and overall quality of the service provision to the customer by continuously monitoring the fulfilment of the SLAs as well as propagating the legislation changes.

2.5 Tools for multi-cloud applications continuous deployment and operation

2.5.1 ADAPT deployment and monitoring

DECIDE ADAPT tool offers the following functionalities: deployment of multi-cloud applications, monitoring of the deployed applications to verify if the declared MCSLA is satisfied or not, and deployment adaptation to cope with identified violations.

ADAPT uses information from the Application Description to generate and apply the scripts for the deployment of the multi-cloud application. ADAPT uses ACSmI as a unified interface to create, monitor and release CSP resources. ADAPT continuously monitors the MCSLA and in case of a violation (from either the application or the underlying CSP resources), it informs the operator and triggers the redeployment process. Redeployment can be automatic for a low technology risk application or subject to operator's confirmation for a high technology risk one.

2.5.2 MCSLA editor

The MCSLA Editor module is part of the continuous operation phase and serves as the user interface (UI) through which the developer will specify the multi-cloud SLAs agreed with the client. The MCSLA editor provides the developer all the possible SLOs and SQOs, which may partly incorporate default values, aggregated values or overwritten values depending on the values resulting from the contracted CSPs. This resulting MCSLA serves as the contract between the developer and the users of the application. Additionally, the MSCLA will be used for monitoring purposes.

2.6 Sample multi-cloud application in DECIDE: Sock shop application

DECIDE integrated architecture and the included components and tools will be tested and validated through the different DECIDE use cases, which are being described in several deliverables in DECIDE [4] [5]. The final requirements of the use cases will be delivered at the same time as this deliverable. For this reason, for the first version of the prototypes to be delivered in M12, as well as for the conceptual definition of the architecture included in this report, a sample "multi-cloud compliant" application has been chosen, in order to be able to prove and execute all the DECIDE components with the same "multi-cloud" based application. The application chosen has been the Sock Shop application

The Sock Shop application is a micro-services-based application used to illustrate micro-services architectures [6]. Sock Shop application implements the front-end of a website that sells socks and each of the components is implemented as a micro-service. The application is intended to aid the demonstration and testing of micro-service and cloud native technologies.

The application is structured into 7 main components, Payment, Orders, Carts, Catalogue, Users, Shipping and Queue Master. These 7 components can be used as the components to be deployed into different cloud resources, following the DECIDE definitions for multi-cloud applications (see section 2.1).

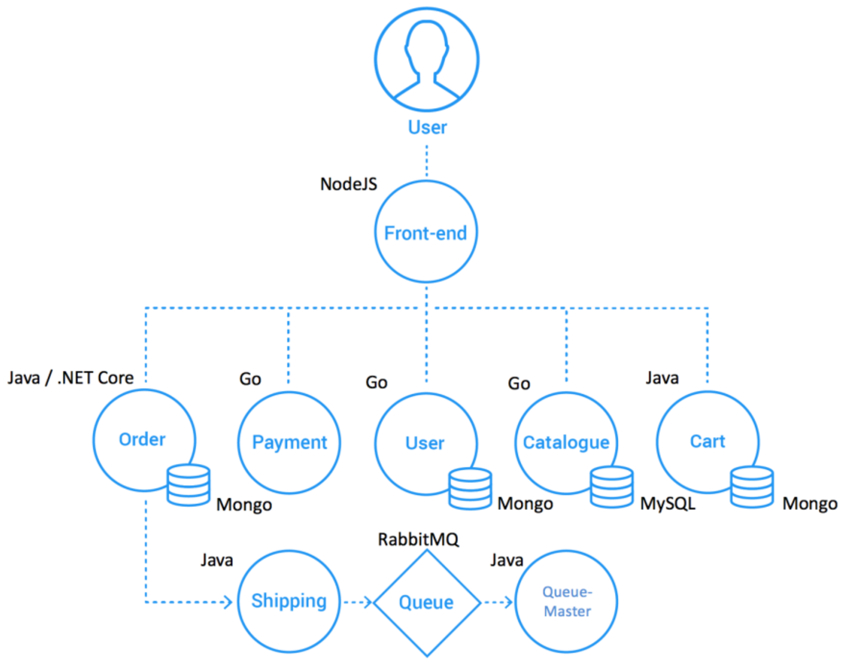

Figure 2. Sock Shop application main components [7]

This Sock Shop application can be deployed using different technologies and frameworks as described in [6]. Each of the DECIDE partners implementing the different components in the architecture has used its own deployment of the Sock Shop application (i.e. Kubernetes, Docker), focusing on the most relevant aspects for its specific needs. In the next releases of the DECIDE components, the same instance of the Sock Shop application will be used, as the first step to provide an integrated ecosystem for the use cases applications.

3 Detailed DECIDE integrated architecture

3.1 Tools for multi-cloud applications design and development

3.1.1 NFR editor

NFR editor is the component where the developers can state the NFRs that they want to consider during the development and the operation of the multi-cloud application.

3.1.1.1 Structural description

The NFR editor will support the developer in the "continuous architecting" phase, as it will be the main mean to select the relevant NFRs (qualitative) so that ARCHITECT can propose specific patterns for them. The NFR list is closed, for the context of DECIDE, to the following characteristics: availability, cost, location, security (legal), performance, and scalability but it can be extended with new ones when needed.

The NFR editor will also support the developer in the "pre-deployment" phase for detailing the previously selected NFRs. In this step, the NFR editor will provide the means for the developer for selecting the concrete values of some of the NFRs (only those that can be quantified i.e. cost, location) and applying these values to concrete components of the multi-cloud application.

The main functionalities of the NFR editor are:

- Provide the available qualitative NFRs, and the means to select them at application level.
- Provide the available quantitative NFRs, and the means to detail their values at component level.
- Provide the means to store the selected NFRs (qualitative and quantitative).

In the next figure, the sub-components of the NFR editor are showed and explained afterwards:

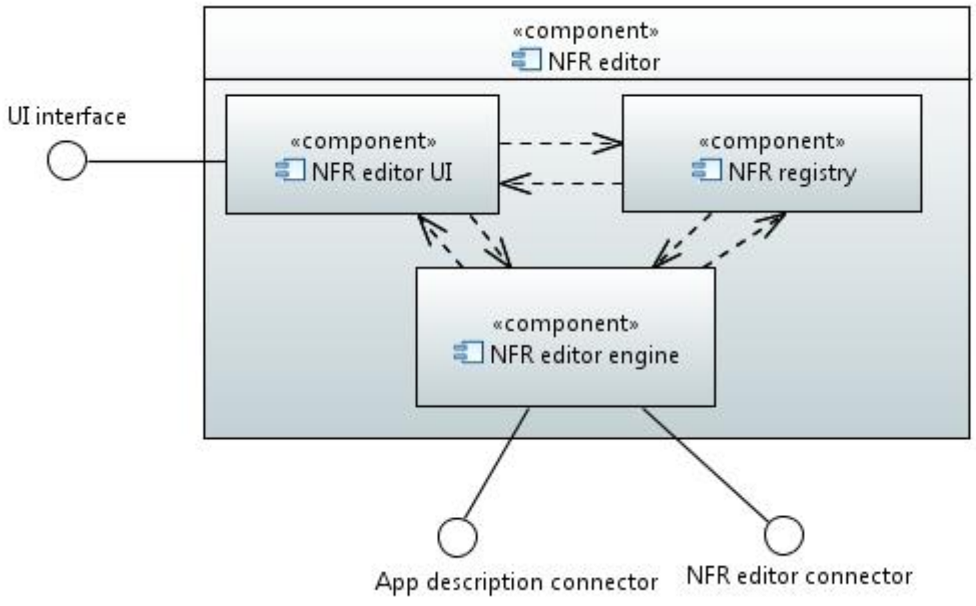

Figure 3. NFR editor component diagram

The NFR editor is composed of the following sub-components:

NFR editor UI

This subcomponent provides the graphical user interface of the NFR editor, both for selecting the qualitative NFRs at the continuous architecting phase and for detailing the quantitative values for each component. The NFR editor UI will be loaded when ARCHITECT or OPTIMUS call it, through the NFR editor engine.

NFR registry

The NFR registry stores the available NFRs to be loaded by the NFR editor UI. This registry will be static, and it will contain the possible NFRs and the possible values to be assigned.

NFR editor engine

The NFR editor engine is the component that manages the different activities to be carried out by the NFR editor. This sub-component gets the requests from ARCHITECT and OPTIMUS and triggers the different sub-components inside the NFR editor. It also stores the values of the selected NFRs in the App Description.

NFR editor will communicate with ARCHITECT and OPTIMUS for getting the requests to select the NFRs, with App Description to store the selected values for the NFRs of the different components and with the DevOps framework for providing the UI in the integrated DECIDE framework.

In the following diagram, the external interfaces of the NFR editor are shown:

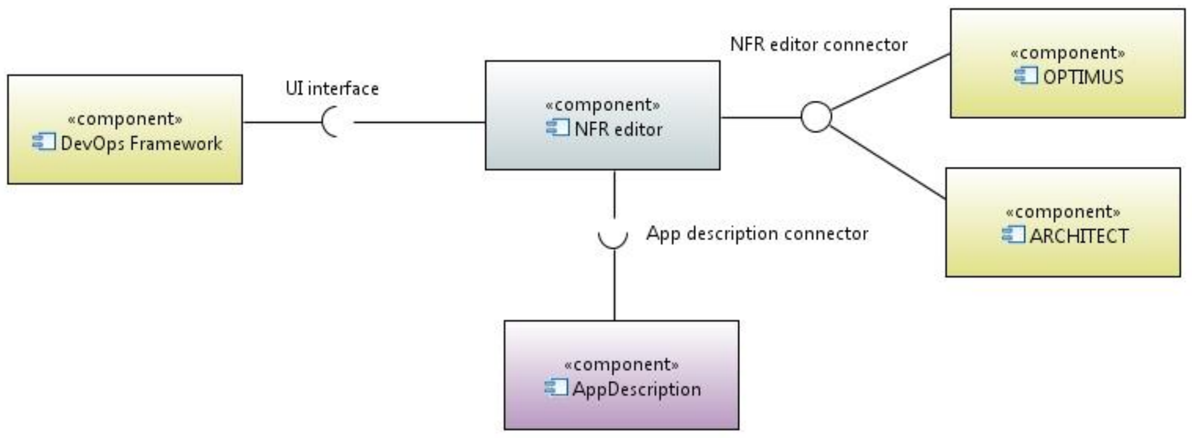

Figure 4. NFR editor external interfaces component diagram

3.1.1.2 Behavioural description

In the next picture, the interaction between the NFR editor and other components in DECIDE is shown as well as the messages they share:

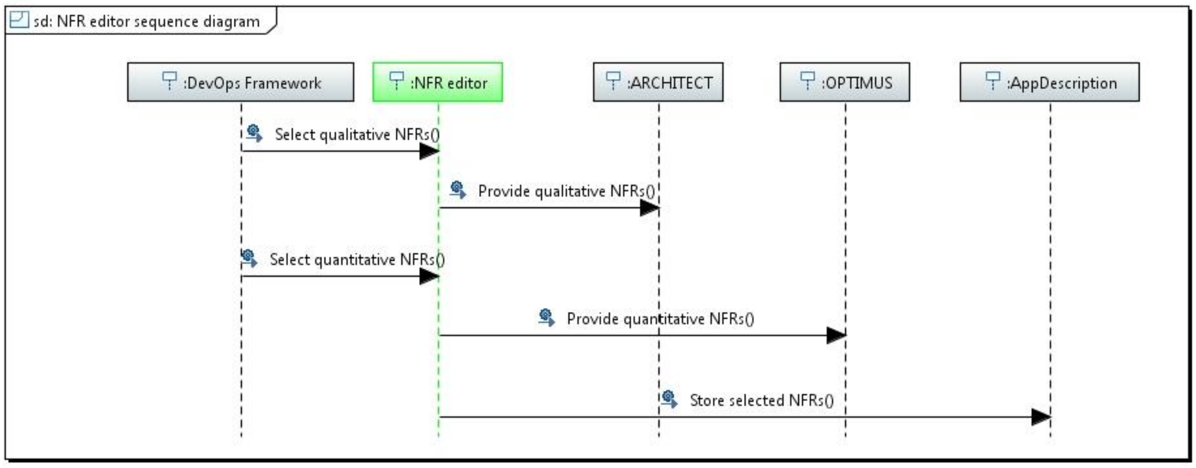
sd: NFR editor sequence diagram
:DevOps Framework
:NFR editor
:ARCHITECT
:OPTIMUS
:AppDescription
Select qualitative NFRs()
Provide qualitative NFRs()
Select quantitative NFRs()
Provide quantitative NFRs()
Store selected NFRs()

Figure 5. NFR editor sequence diagram.

1. Select qualitative NFRs: The developer through the DevOps Framework UI (or the UI of the NFR editor) selects the corresponding NFRs from the list provided by the NFR editor.
2. Provide qualitative NFRs: ARCHITECT gets the selected NFRs from the NFR editor.
3. Select quantitative NFRs: The developer through the DevOps Framework UI (or the UI of the NFR editor) selects the value for corresponding NFRs from the list provided by the NFR editor, for each of the micro-services of the multi-cloud application.
4. Provide quantitative NFRs: OPTIMUS gets the selected values for the NFRs from the NFR editor.
5. Store selected NFRs: The NFR editor stores the selected NFRs and values in the Application Description so that they can be assessed during the operation phase of the multi-cloud application.

3.1.2 ARCHITECT

ARCHITECT supports the developer with preparing the application for a multi-cloud deployment scenario by providing and suggesting a set of (multi-)cloud patterns, which must or should be applied to the application.

3.1.2.1 Structural description

By means of the functional requirements, ARCHITECT is decomposed in several functional blocks and interfaces. The ARCHITECT component has a set of functional requirements [1] that can be summed up in the following functionalities:

- Provide/ recommend to the user (i.e. the developer) architectural patterns based on his/her prioritized NFRs as well as additional information (supplied by the user), with guidelines on how to apply them, to which component this needs to be applied and in which order. This should be performed through a UI.
- Provide a repository of relevant multi-cloud patterns.

Apart from these functionalities, ARCHITECT will help to initiate the development of an application in the context of DECIDE. This includes the creation of the DECIDE project artefacts, mainly consisting of the application description contained in a code repository.

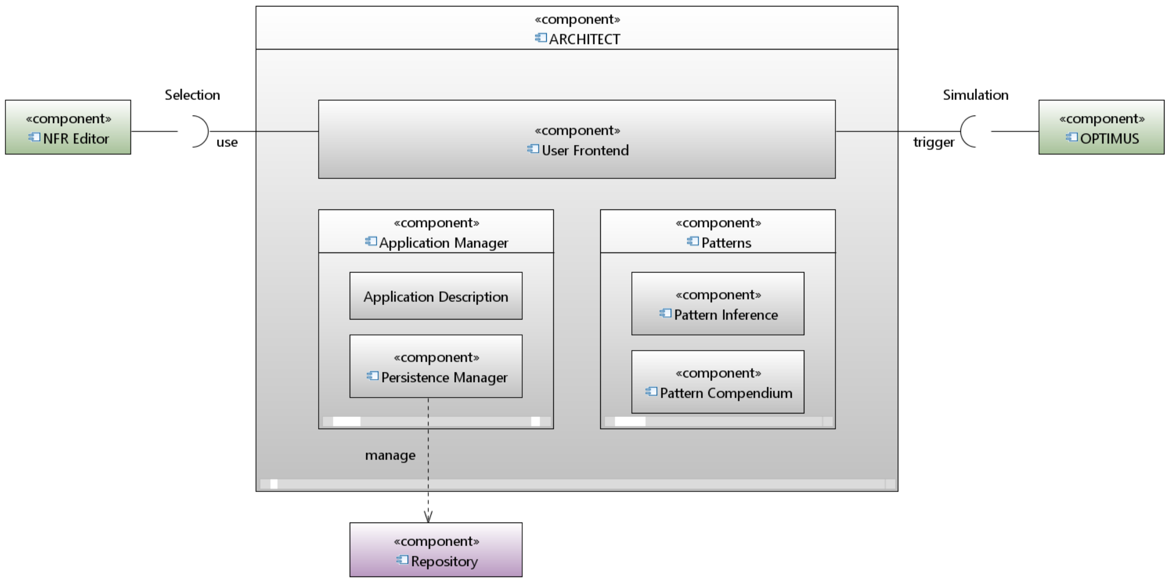

Figure 6 Components of ARCHITECT

ARCHITECT consists of three core elements, as depicted in the figure above. A frontend for user interaction, the application description manager for dealing with the DECIDE project model, and finally the patterns catalogue with the pattern inference engine.

User Frontend

This element depends on the context. For example, if ARCHITECT is integrated in an IDE, this part provides the mechanism how the ARCHITECT component is plugged in. its main task is the interaction with the developer and it provides the necessary user interfaces to collect and maintain all the application information and to enable the use cases (Figure 7). The User Frontend is the workflow-controlling component of ARCHITECT.

Application Manager

This element is responsible for a convenient abstraction level for the information model of the DECIDE application. It manages all application information in a persistent manner. That means, it encapsulates and hides the technical details, (e.g. the fact that the application description is coded and stored as a JSON structure inside a code repository).

Patterns

This element contains a catalogue of patterns, NFRs and their relationships. The contained information can be enriched to hold additional information experienced over time. The patterns catalogue provides functions that allow the inferring of patterns based on a given set of NFRs and optionally some fixed patterns.

ARCHITECT itself does not provide any external interfaces. Nevertheless, at least the *Patterns* component will be implemented as autonomous library and its functionality could be offered as micro-service being accessible for other implementations. This allows an easy integration of ARCHITECT in a polyglot environment. Nevertheless, ARCHITECT does consume two interfaces, one from the NFR Editor) and the other from the OPTIMUS component).

NFR Editor

ARCHITECT utilizes the NFR Editor for collecting the set of defined non-functional requirements from the application developer. ARCHITECT expects as return value from the editor the list of NFRs that the developer has selected.

OPTIMUS

For a manual triggering of the simulation phase, ARCHITECT should be able to call OPTIMUS. The main artefact transferred is the *Application Description*. Depending on the provided interface of OPTIMUS it can either be referenced through the code repository or be handed over as a parameter in the API method. The result will be returned using the same mechanism. The *User Frontend* and the *Application Manager* may display the result in the current environment in an appropriate way.

3.1.2.2 Behavioural description

Based on the list of the functional requirements [1], several use cases for the developer were identified (Figure 7). These are mainly the creation of a new project, a change of NFRs; a change of selected patterns of an already existing DECIDE project; Finally, the developer or the used CI tool should be able to enter the next DECIDE phase by triggering OPTIMUS for a deployment simulation.

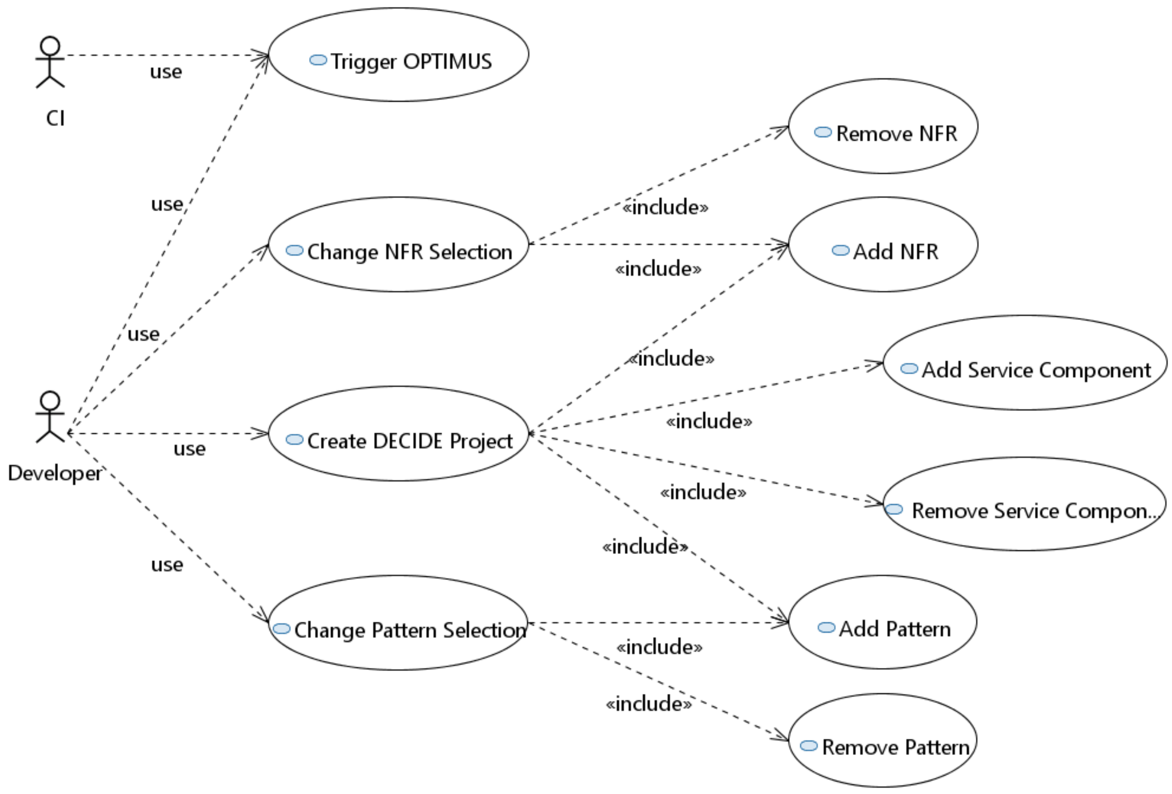

Figure 7. Use Cases of ARCHITECT

By means of a sequence diagram (**¡Error! No se encuentra el origen de la referencia.**), the *Create DECIDE Project* process will be exemplary further detailed and linked to main requirements.

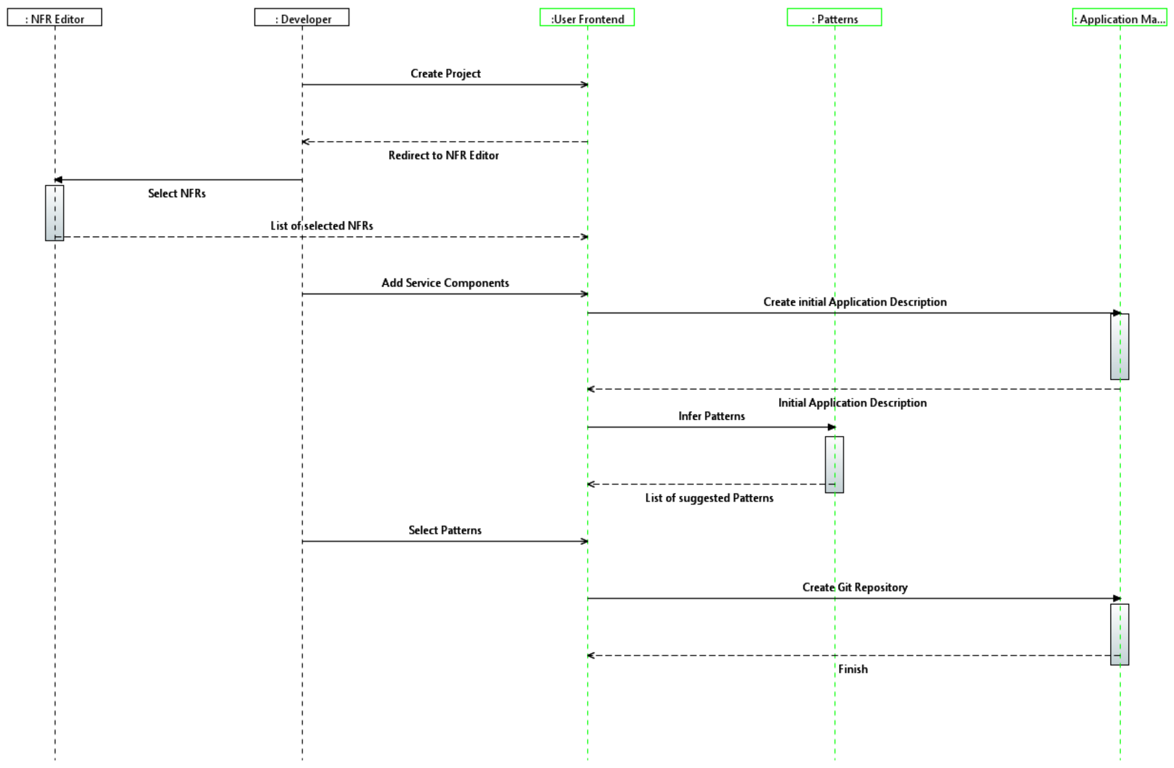

Figure 8 Create new DECIDE Project, sequence diagram.

1. The developer starts the creation of a new DECIDE project.
2. The *User Frontend* will then forward the user to the NFR Editor where she/he can select a set of prioritized NFRs. The NFR Editor returns to ARCHITECT *User Frontend* with the selected list of NFRs.
3. Next, the user is asked about general information about the application, e.g. which micro-services are contained and how they are related to each other and to the application in general. Furthermore, the developer is asked to select any patterns from the catalogue that definitely apply or should apply to the application.
4. In the next step, the *User Frontend* part requests an initial *Application Description* from the *Application Manager*, preliminary only memory.
5. Subsequently, based on the selected NFRs and patterns, a list of (additional) inferred patterns is suggested to the developer.
6. After the developer has finalized the list of applied patterns for the application, the User Frontend finishes the creation process by persisting the final *Application Description* using the *Application Manager*.

3.2 Tools for multi-cloud applications continuous integration and testing

3.2.1 DevOps framework

3.2.1.1 Structural description

The DevOps framework is the entry point to DECIDE. It allows a user to define a new project and modify its metadata[2], and centralizes de UIs of the different tools and KRs. Besides, the DevOps framework

[2] Project metadata captured form the user: i.e number of microservices, git where the project is located, users, etc.

will act as the orchestrator of the DevOps workflow, launching the corresponding tool whenever appropriate.

The following figure shows the components of the framework:

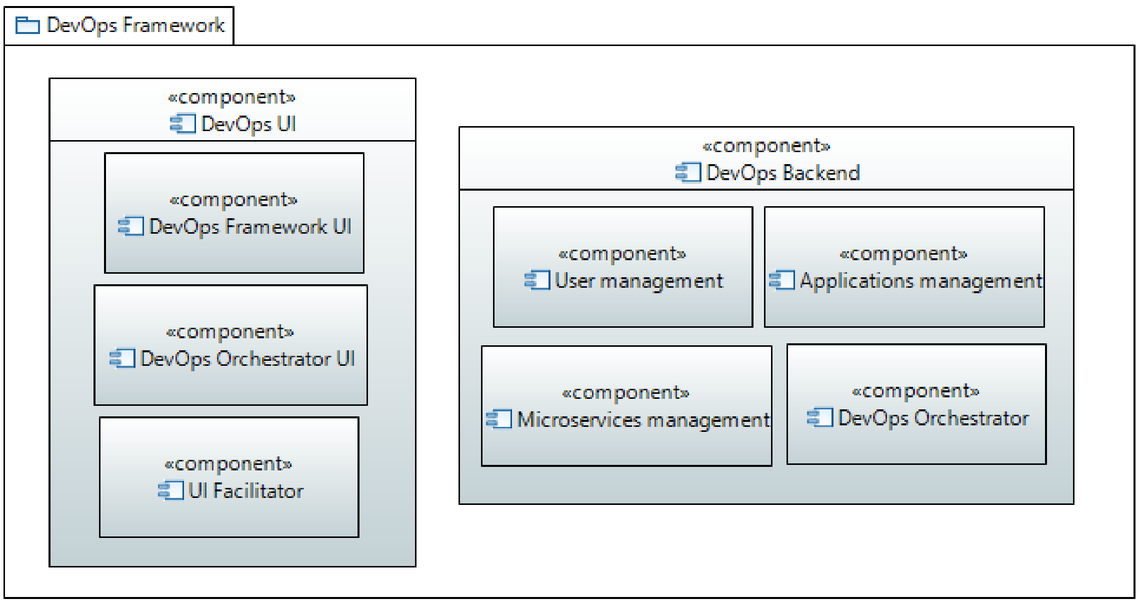
DevOps Framework
«component» DevOps UI
«component» DevOps Framework UI
«component» DevOps Orchestrator UI
«component» UI Facilitator
«component» DevOps Backend
«component» User management
«component» Applications management
«component» Microservices management
«component» DevOps Orchestrator

Figure 9. DevOps framework component diagram

User management

This module takes care of managing the different identities of DECIDE users. Different user roles will have access to different sets of functionalities.

Application management

This component creates and manages the workspaces that are assigned to each application.

Microservices management

This component handles the creation and edition of microservices and their metadata.

DevOps Orchestrator

This module is in charge of launching the appropriate tool depending on the point on the workflow, and to provide them with the necessary information.

DevOps Framework UI

This module provides a graphical interface for tasks related to applications and microservices management.

DevOps Orchestrator UI

This component provides a graphical interface for configuring and showing information about the DevOps tools (tools for continuous development, integration and testing) involved in the workflow.

UI Facilitator

This module facilitates the integration of the UIs of the different Key Results in the DevOps Framework.

The DevOps Framework communicates with the Application Description to store the data entered by the user; with ARCHITECT to obtain the list recommended patterns; with the DevOps tools to configure them and display their information; and with the UIs of the different KRs, since the DevOps Framework unifies all user interfaces. It also instructs OPTIMUS to trigger a new simulation when it corresponds. The figure below shows these communications:

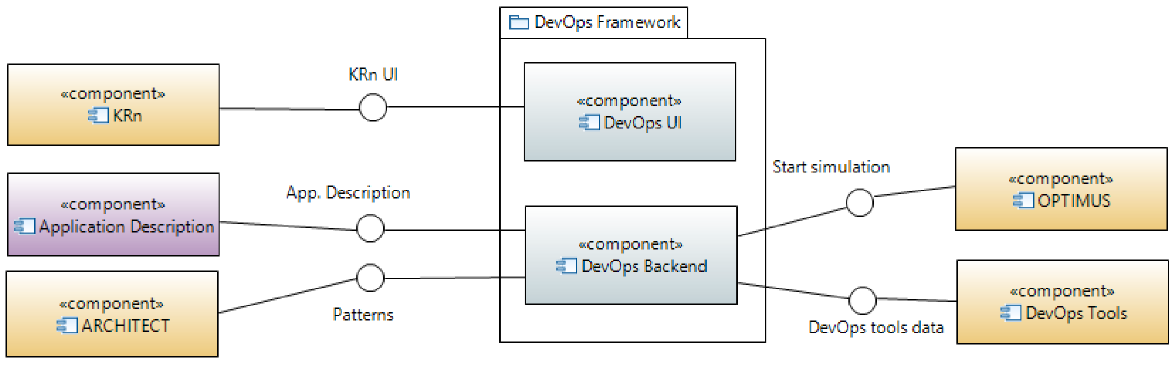

Figure 10. DevOps Framework interfaces diagram

3.2.1.2 *Behavioural description*

The DevOps Framework provides a graphical interface for a user to introduce information about the application and its microservices. When that information is complete, it updates the Application Description with it. It also configures the tools for continuous development, integration and testing according to the provided application information.

ARCHITECT, based on the Application Description, generates a list of recommended patterns, which then sends back to the DevOps Framework to be displayed.

While the development, integration and testing process is taking place, the DevOps Framework receives the information reported by the corresponding tool and displays them in the UI.

It also sends OPTIMUS an order to start a new simulation, either after a direct instruction from the user or as a result of a violation during ADAPT's monitoring process.

The following figure shows these communications:

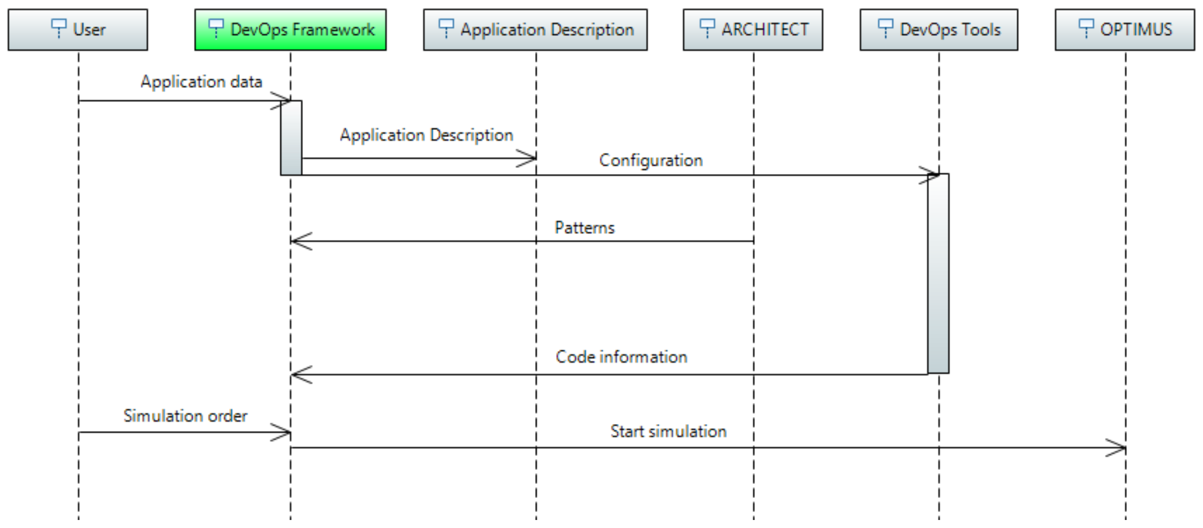

Figure 11. DevOps Framework sequence diagram

3.3 Tools for multi-cloud applications (pre) deployment

3.3.1 OPTIMUS

3.3.1.1 Structural description

The main functionalities in OPTIMUS are:

- Multi-cloud application classification**.** This functionality will include the classification of the components that from the multi-cloud application (computing, computing IP, storage persistency, storage DB) [8]. For this purpose, the profiling of the multi-cloud application has to be considered as an input. This classification will be based on the information provided by the developer and the information stored in the general applications profiling repository.
- Theoretical deployment generation**.** Once the classification is made, and the NFRs gathered, it will perform a process where it will obtain a theoretical schema for the deployment. This schema will be composed of generic types of CSPs, associated to the types set to the micro services. With these generic types of CSPs suitable for the components, a request will be made to the corresponding service of ACSmI. This functionality requires the "CSP modelling" functionality to be available.
- Simulation. The combination of the different possibilities of deployment, taking into account the theoretical deployment and the sorted list of CSPs (from ACSmI) that suit them, will be ranked in order to select the best ones. The schema with the needed information for the App controller will be built and shown to the developer to confirm it.

The figure below shows the components and sub-components that will support the listed functionalities:

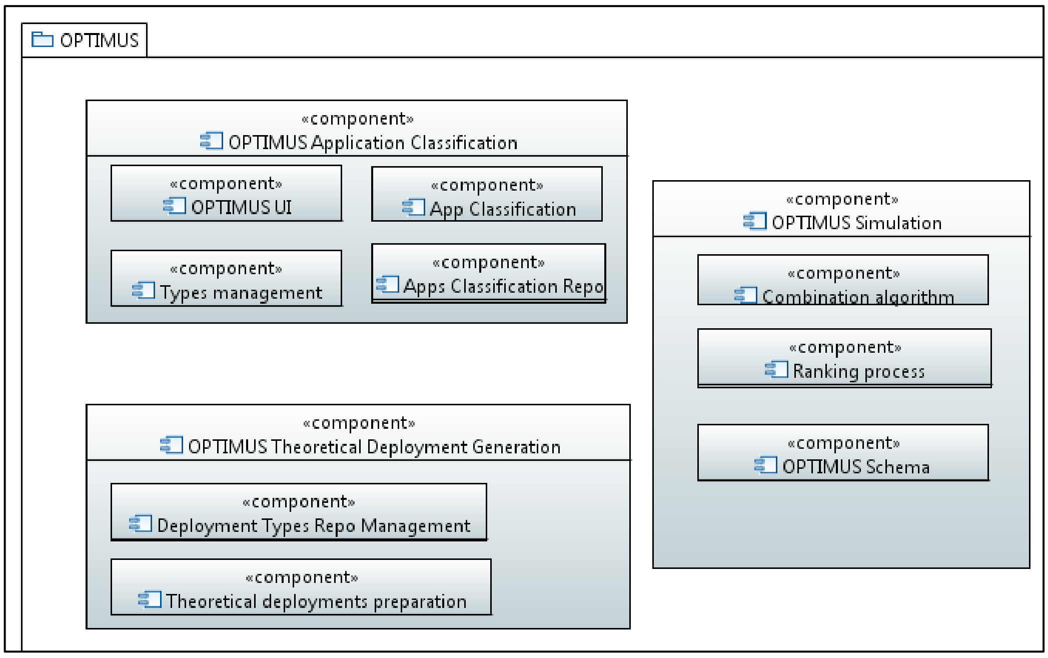
OPTIMUS
«component» OPTIMUS Application Classification
«component» OPTIMUS UI
«component» App Classification
«component» Types management
«component» Apps Classification Repo
«component» OPTIMUS Simulation
«component» Combination algorithm
«component» Ranking process
«component» OPTIMUS Schema
«component» OPTIMUS Theoretical Deployment Generation
«component» Deployment Types Repo Management
«component» Theoretical deployments preparation

Figure 12. OPTIMUS component diagram

Application classification

The input for this classification task will come from the developer (the UI will show information to him to complete and confirm), and it will be matched with the information stored about types of multi-cloud applications, and the characteristics associated to each of those types. The output will be loaded into *Apps classification* repo. This information will be part of the App Description.

The *Types management* component will manage the system knowledge about types of multi-cloud applications. This information is related to the defined CSP types (ACSmI).

Theoretical deployment generation

The *Deployment Types* repo will contain information about the micro-services types and the CSPs on which they could be theoretically deployed. The maintenance of this repo will be performed by the *Deployment types management* module. At this point of the project, it will be static information but later on it should be updated with the data provided by ACSmI.

Taking as input the multi-cloud application classification (micro-services) and the Apps NFRs, the theoretical deployment generation component will access the *Deployment Types* repo to obtaining the set of CSPs that can be used for its deployment. Once it has all the information, it will create a list of possible CSPs for each micro-service, and will assign some sort of score for each option.

Simulation

The entry for the combination process will be the information about different possibilities for the deployment. The algorithm will perform a combination of all these possibilities, using the different CSPs that can be used for each micro-service. These combinations will be sorted from the best of them until the worse. An output with the best possible deployment will be created.

The best option for the deployment could be selected and confirmed by the developer, and sent (as schema) to the controller.

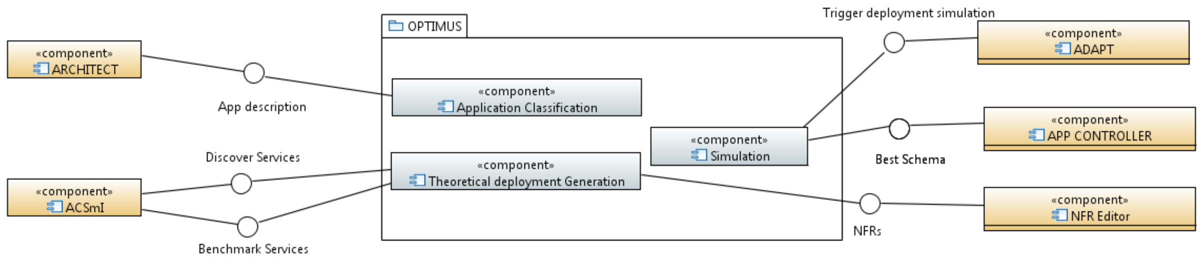

Figure 13. OPTIMUS external interfaces component diagram

3.3.1.2 *Behavioural description*

The behaviour of OPTIMUS tool, and the interchanged data among the different actors in this part of the DECIDE workflow, is shown by the picture below:

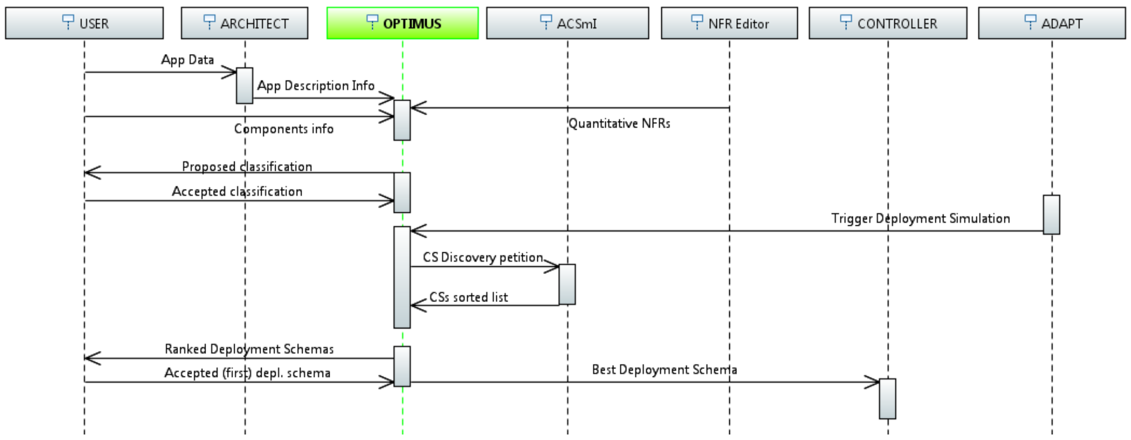

Figure 14. OPTIMUS Sequence diagram

The messages among OPTIMUS and the rest of tools are:

1. App Data: The first information about the application is provided by the user, and requested by *ARCHITECT* UI.
2. App Description Info: Once *ARCHITECT* has suggested the most suitable patterns, the related information is stored as part of the Application Description and *OPTIMUS* will have access to it.
3. Quantitative NFRs: The NFRs indicated by the user are an important information in order to obtain a proper classification and the best theoretical deployment for the application.
4. Components info: The developer or user has already developed the application or has a detailed design about it. *OPTIMUS* requires additional information about the micro-services that the application is composed of.
5. Proposed classification: *OPTIMUS* IU presents the user the results of the application classification.
6. Accepted Classification: The user, through *OPTIMUS* UI, accepts the classification made.
7. Trigger Deployment simulation: When ADAPT identifies a violation, a new redeployment simulation is triggered by requesting *OPTIMUS* and sending it the information about the violation
8. CS Discovery petition: *OPTIMUS* asks to *ACSmI* for a list of CSPs that fulfill the requirements that the user and the classification process have established.

9. CSPs sorted list: *ACSmI* sends *OPTIMUS* a list whose first element is the CSP that best fits the non-functional requirements.
10. Ranked Deployment Schemas: After *OPTIMUS* has created the schemas for the deployment, based on the information sent by *ACSmI*, it will present to the user as a ranking and he or she will accept it, and therefore, the best one will be selected.
11. Accepted (first) deployment Schema: The schema selected by the user, the best schema presented by *OPTIMUS*.
12. Best Deployment Schema: This output is the global result from *OPTIMUS*. It will be used by the *App controller* tool for building the deployment script.

3.3.2 Application controller

In year 1 of the project the Application Controller will initially assist in managing the intelligence regarding the current and historical deployment topology. The aim of this functionality is to mitigate reusing deployment topologies that were faulty or inadequate in the past.

Further functionality as stated in Section **¡Error! No se encuentra el origen de la referencia.** has been realised in other various areas.

More detailed information about the implementation of the Application Controller can be found in D3.10 [9]

3.3.2.1 Structural description

The following requirements have been elicited in the project for the Application Controller [1], these are translated below in Figure 15 as functional components. The requirements can be summed up in the following functionalities:

- Holding the intelligence of the different deployment configurations that the multi-cloud application has had in its operation time. Storing these deployment configurations will allow avoiding those configurations that resulted problematic in terms of security, performance or legal awareness.
- Maintain an interface to OPTIMUS in order to receive the chosen deployment configuration to be stored.

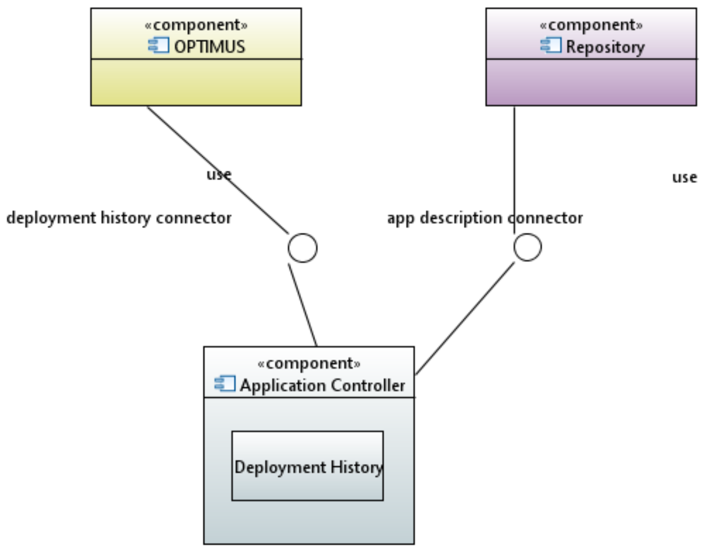

Figure 15. Component Diagram for Application Controller

The Application Controller component holds the intelligence for the different deployment configurations that the multi-cloud application has had in its operation time. Storing these deployment configurations will allow in the future avoiding those that resulted problematic in terms of fulfilling the multi-cloud application's fixed NFRs. The Application Controller component is in charge of this functionality. It provides OPTIMUS and ADAPT an interface to read and write the current and historical configurations. Furthermore, it stores the deployment history in the code repository where the Application Description is also stored.

The deployment history will include meta-data regarding the deployment configuration such as Time and date of deployment, the current status, information on the microservice, CSP data and information regarding any SLA violations.

3.3.2.2 *Behavioural description*

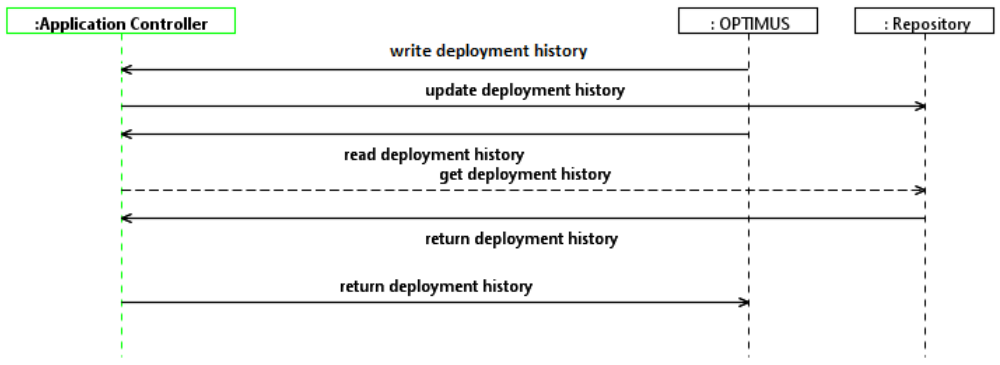

Figure 16. Sequence Diagram for writing and reading deployment configuration

The sequences performed for reading and writing to the deployment history are as follows:

1. Once a successful deployment has taken place, OPTIMUS registers the current deployment configuration with the Application Controller component. The information to be registered is as described above and is specified in the meta-data model (see annex 1 for the meta-model information).
2. The *Application Controller* component will update the existing file that holds the deployment history in the code repository. In the case that the file does not exist, the Application Controller component should create it.
3. If an SLA violation takes place, this is reported and the file is updated accordingly (same process as prior step).
4. In the case of a new simulation phase, OPTIMUS shall read from the deployment configuration history through the *Application Controller* component.
5. The *Application Controller* component supplies in turn OPTIMUS with the information needed in order to evaluate which deployment topology is adequate.

3.3.3 ACSmI

3.3.3.1 *Structural description*

The Advanced Cloud Service (meta-) intermediator (ACSmI) will provide a cloud services store where discovery, contracting, managing and monitoring different cloud services. ACSmI will provide the means to assess continuous real-time verification of the cloud services non-functional properties fulfilment and legislation compliance enforcement. ACSmI will also provide means for Cloud services endorsement from different CPS.

These are the main functionalities envisioned:

- Endorse a cloud service into the ACSmI. ACSmI will allow to register services. This can be performed by the CSP itself, by the multi-cloud application operator or by the ACSmI Administrator. The registry of each service should cover the different defined terms for modelling the CSPs and their services. This will allow the discovery of the services from the service registry.
- Discover and benchmark services. OPTIMUS will indicate the NFR of the services that shall be delivered to the ACSmI as input. ACSmI will discover, from the services stored in its registry, the most appropriate ones for that set of NFRs. Then, from the set of discovered services, ACSmI will prioritize these services in terms of NFRs fulfilment (including legal aspects) which will be passed to OPTIMUS as a short list. The list will include, additionally, the degree of fulfilment of the NFRs requested by the user.
- Contract services. This functionality will allow dealing with all the activities related to the contracts within the ACSmI. Depending on the type of services and the CSP, ACSmI will manage the contracts in two different ways: 1) ACSmI will facilitate contracting services directly by the user to the provider and 2) ACSmI will manage the contract itself with the provider and the user. In this last case, ACSmI will have mainly two types of contracts. The first one is the contract with the CSP and the other one is with the user of the services intermediated by the ACSmI.
- Manage CSPs. This functionality will allow the management of the different connectors to facilitate the contracting of the services and to monitor them. This functionality will be in charge of informing ADAPT with the required information for the deployment of the multi-cloud application through the different contracted services.
- Monitor NFR CSPs and manage the violation alerts. This functionality will monitor the SLA (NFRs) of the service offered by the CSPs to detect any violation of the SLAs. These metrics will be recorded and passed to the ADAPT monitoring during the operation of the services. If a violation is detected, an alert to the CSP will be sent.
- Monitor the use and bill the user. This functionality will allow calculating the costs generated by the user for using ACSmI recommended cloud services, and to provide with the corresponding invoice. To be able to generate the billing of the contracted services, the ACSmI shall monitor the use of the different cloud services.

The high-level architecture of the ACSmI is presented next. The Figure 17 is an updated version of the ACSmI architecture presented in D5.1 "ACSmI requirements and technical design" [3].

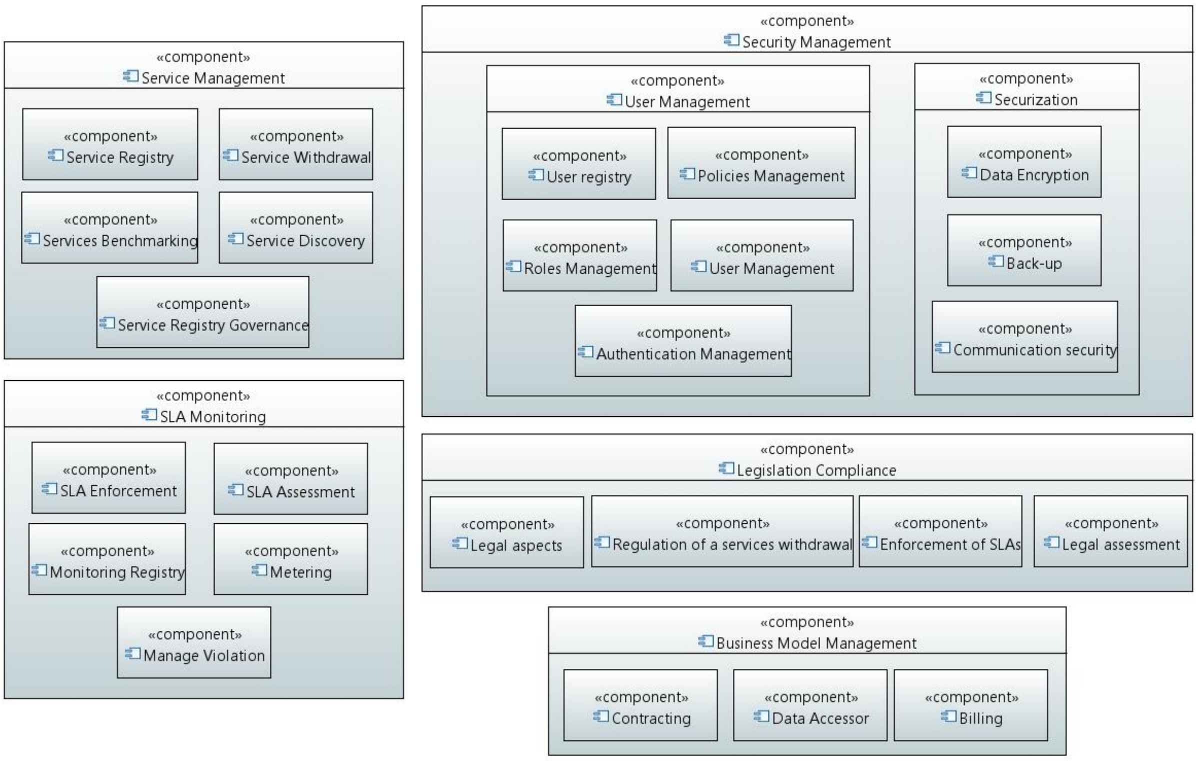
«component» Service Management
«component» Service Registry
«component» Service Withdrawal
«component» Services Benchmarking
«component» Service Discovery
«component» Service Registry Governance
«component» SLA Monitoring
«component» SLA Enforcement
«component» SLA Assessment
«component» Monitoring Registry
«component» Metering
«component» Manage Violation
«component» Security Management
«component» User Management
«component» User registry
«component» Policies Management
«component» Roles Management
«component» User Management
«component» Authentication Management
«component» Securization
«component» Data Encryption
«component» Back-up
«component» Communication security
«component» Legislation Compliance
«component» Legal aspects
«component» Regulation of a services withdrawal
«component» Enforcement of SLAs
«component» Legal assessment
«component» Business Model Management
«component» Contracting
«component» Data Accessor
«component» Billing

Figure 17. ACSmI High Level Architecture

There are five main components in charge of implementing the core functions of the ACSmI. Next, a high-level description of the main components and their corresponding sub-components is presented.

Service Management

This component is in charge of executing and managing all the operations related to the services offered by the ACSmI. Functions like cloud services endorsement, intelligent discovery, or service operation are covered by this component and the corresponding sub-components. The sub-modules included in Service Management are:

a. *Service Registry*: The service registry is in charge of registering all the information related to the services offered by ACSmI. The type of information to be registered will be related to the information about the NFRs.
b. *Service Registry Governance*: The Service Registry Governance is responsible for managing the access and update to the service registry.
c. *Service Discovery*: This sub-component is in charge of managing the requests from the users (OPTIMUS) to discover the services. It gathers and processes the NFRs from OPTIMUS when discovering services in the ACSmI.
d. *Services Benchmarking*: This sub-component will be in charge of comparing the different NFRs of the services and providing a short list of services based on the degree of the fulfilment of the NFRs.

Legislation Compliance

The legislation compliance is in charge of assessing the compliance of the services with respect to the different legislations (i.e. GDPR[3] and Code of Conduct of CSPs). The sub-modules included in this module are:

a. *Legal aspects component* is a database where all the legal-related information gathered from the CSPs when endorsing a service is stored.
b. *Legal assessment component,* is responsible of checking if the information collected from the CSPs accomplishes the requirements set by the applicable legislation, as requested by the user when eliciting the NFR. This module is also in charge of ensuring that any changes in the legislation will be propagated and all the services of the service registry will be reassessed.
c. *Enforcement of SLAs component*, is responsible of indicating what action is possible/ will be taken when one or more SLAs are not respected.
d. *Regulation of a service withdrawal* is responsible of showing how contracts are terminated as well as what terms regulate the termination of a service, e.g. data format on exit, data portability, security measures etc.

Cloud service SLA monitoring

This module is in charge of the management of the monitoring of the services in the ACSmI. This module is composed of the different sub-modules to perform the corresponding activities:

a. *Manage violation. This sub module is in charge of managing and alert that a service in the ACSmI is not fulfilling the SLA.* Metering sub-component *collects the different SLA terms that will be monitored and selects the metric/parameters associated to each of the different terms. This module will use the 'Contracting' interface to receive the SLA terms that need to be monitored in a machine-readable way.*
b. *SLA Enforcement* retrieves the values of these parameters, and stores them in the monitoring repository.
c. *SLA Assessment* assesses the compliance of the SLA of the contracted services with respect to the values retrieved for the parameters by the SLA enforcement sub-module (contracted values vs. real values).
d. *Manage Violation* is a sub-module in charge of managing and triggering alerts when a service contracted by ACSmI is not fulfilling its SLA.
e. *Monitoring repository* provides, assists, and automates the storage of metrics and values, and the detected violations in its repository.

Business Model management

This core component is in charge of the execution and management of all the operations related to Service Contracts in the ACSmI. It also performs all the activities related to the financial operations with the different users of the ACSmI. The sub-modules included in this component are:

a. *Contract Manager:* It is in charge of the management of the core functions with respect to the service contracts. It manages mainly two different types of contracts: 1) contracts between the user and the ACSmI and 2) contracts between the CSPs (service providers) and the ACSmI.
b. *Service Contract Registry:* This sub-module stores the different contracts existing in the ACSmI.
c. Accounting: It is responsible for monitoring and calculating the total values in order to bill the users for the services and to pay the CSPs for the services used.

[3] General Data Protection Regulation

d. Billing: It generates the bills for the users.

Security management

This component is in charge of designing and developing the means to guarantee the secure operation of the ACSmI. It includes functionalities such as identity propagation and federated authentication and authorization, but not only, as this module deals with all the aspects related to the management of security of the ACSmI, such as data and communication security as well as backup services. These services shall secure all data generated and stored resulting from the activities performed by the ACSmI itself. These shall be stored in the service registry, service contract registry and user registry components of the ACSmI. The sub-modules included in this component are:

a. *Roles Manager*: It manages the activities related to the roles in the ACSmI (creation, modification, assignment, deletion).
b. *Policy Manager*: It manages the activities related to the policies in the ACSmI (creation, modification, assignment, deletion).
c. *User Manager*: It manages the activities related to the users in the ACSmI (creation, modification, roles assignment, deletion).
d. *User Registry*: It stores all the information associated to the users of the ACSmI.
e. *Authentication* Manager: This sub-module performs the authentication of the users and manages the access to the different actions/functions of the ACSmI for every user.
f. *Communication Security*: It provides secure communication means using the SSL transport layer encryption between the client and the platform as well as between the platform and cloud infrastructures.
g. *Backup service*: It is responsible to carry out incremental back-ups for allowing the recovery of data of the ACSmI in case it is necessary.
h. *Data encryption*: It is responsible to encrypt the data of the ACSmI in order to maintain secured these data in case of cyber-attacks.

A detailed description and design of each component is presented in the deliverable D5.1 "ACSmI requirements and technical design" [3].

The following picture presents the interfaces that ACSmI will have with other DECIDE components.

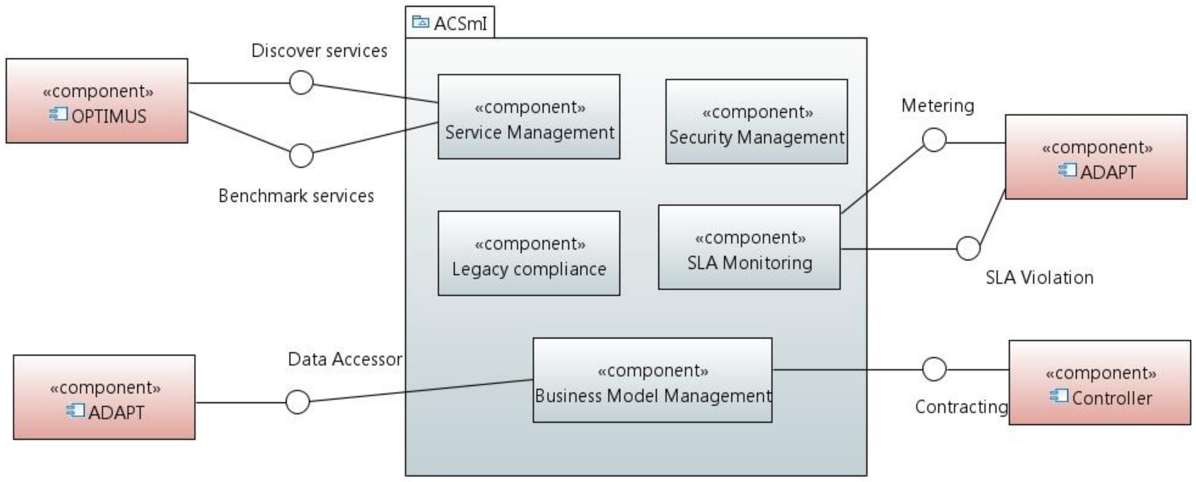

Figure 18. ACSmI external interfaces

The internal interfaces between the ACSmI components are detailed in the D5.1 "ACSmI requirements and technical design" [3].

3.3.3.2 Behavioural description

The ACSmI behaviour and the interchanged data among the different actors in this part of the DECIDE workflow, is shown in Figure 19.

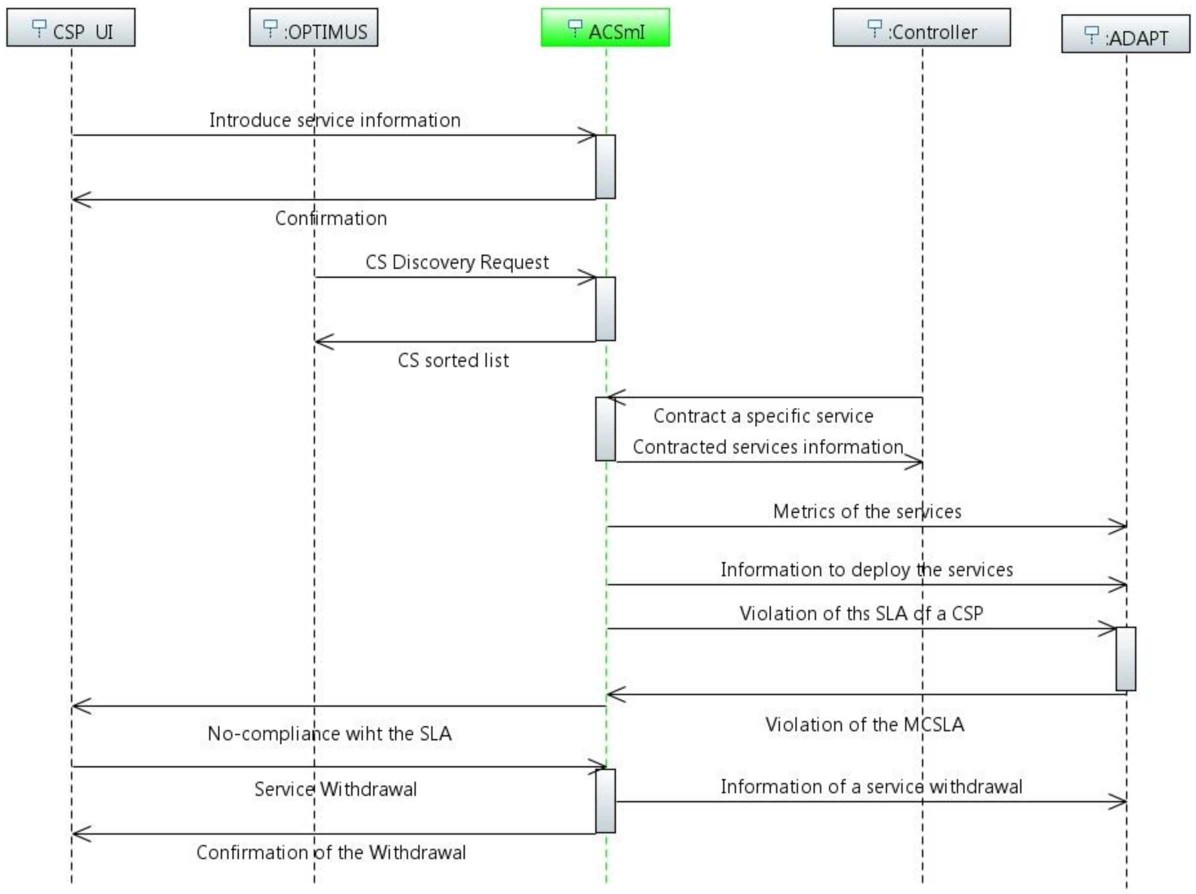
CSP UI
:OPTIMUS
ACSmI
:Controller
:ADAPT
Introduce service information
Confirmation
CS Discovery Request
CS sorted list
Contract a specific service
Contracted services information
Metrics of the services
Information to deploy the services
Violation of ths SLA of a CSP
Violation of the MCSLA
No-compliance wiht the SLA
Service Withdrawal
Information of a service withdrawal
Confirmation of the Withdrawal

Figure 19. ACSmI Sequence Diagram

Four external components interact with ACSmI:

1. *CSP UI*. Each CSP or the *ACSmI* operator should introduce the information to maintain the *service registry* updated. This component should interact with the *service management* module. This component will also be informed if a non-compliance of the SLA occurred. The CSP will have the possibility to ask for the withdrawal of a service from the service registry.
2. *OPTIMUS* will request the discovery of services that cover the requirements. Once these services are discovered by the *ACSmI*, a message with the services discovered will be sent to *OPTIMUS*. This list of services will be sorted according to the level of compliance with the requirements.
3. *Application Controller*. The *Application Controller* will request *ACSmI* to activate those services required for the application, and *ACSmI* will return the relevant information to deploy the application components (or micro-services) on these services. Instead of the *Application Controller*, the information to activate the services could be provided by the DECIDE *DevOps framework*.
4. *ADAPT*. The interaction between *ACSmI* and *ADAPT* can occur from three different ways: 1) *ACSmI* will provide *ADAPT* with all the information required to deploy the application, in addition to the one provided by the application controller, 2) *ACSmI* will provide *ADAPT* monitoring with the metrics of the contracted services to check if there is a violation of the

MCSLA and also to inform of a violation of SLA, and 3) *ACSmI* will inform *ADAPT* that a service is not available in the service registry any more due to a withdrawal process.

3.4 Tools for multi-cloud applications continuous operation

3.4.1 ADAPT

3.4.1.1 Structural description

The main functionalities of ADAPT are the following.

- Deployment of the multi-cloud application. ADAPT generates and applies the scripts to deploy the application's components (containerized microservices) on one or multiple Cloud providers, as indicated in the Application Description. The Application Description provides detailed information about the microservices and their containers, about the related CSP resources to be used, and about the MCSLA to be monitored for both the application and the underlying cloud resources. A mandatory prerequisite is that a contract for the needed CSP resources has already been signed.
- Monitoring of the application MCSLA. ADAPT also monitors the status of the deployed multi-cloud based application and verifies that the non-functional requirements and the SLOs are being fulfilled. If a violation of any of the NFRs or SLOs is detected, ADAPT monitoring components will generate the proper actions depending on each situation and context: an alert saying that the working conditions are not met will be sent to the operator, and the "adaptation" process will be launched, through the violation handlers component.
- Adaptation of the multi-cloud application. If the application is of high technology risk, the operator will have to confirm the following redeployment configuration proposed by OPTIMUS; whereas, in case of low technology risk, the redeployment will be automatic.

The main components of ADAPT are shown in the following Figure along with their external interfaces.

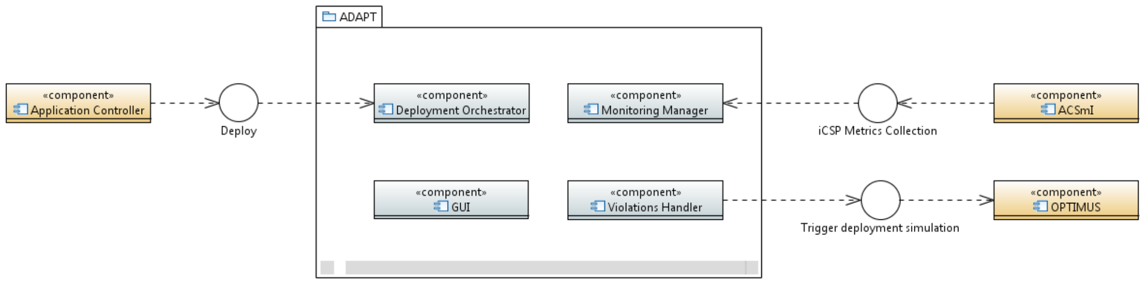
ADAPT
«component» Application Controller
Deploy
«component» Deployment Orchestrator
«component» Monitoring Manager
iCSP Metrics Collection
«component» ACSmI
«component» GUI
«component» Violations Handler
Trigger deployment simulation
«component» OPTIMUS

Figure 20. ADAPT Component Diagram and Interfaces

Deployment Orchestrator

The *Deployment Orchestrator* is responsible for orchestrating the deployment lifecycle (deployment, un-deployment, user confirmation, redeployment) for user applications and their components. This component gets all its input information from the Application Description.

Monitoring Manager

The *Monitoring Manager* controls the monitoring functionality for the application, according to its defined (Multi-Cloud) SLA. It identifies and raises any related violations, including those for the CSPs where the application is deployed on. Monitoring information for the CSPs is collected from ACSmI.

Violations Handler

The *Violations Handler* will handle any violation raised by the *Monitoring Manager*, either regarding the application MCSLA or the CSPs' NFRs. Violation handling may lead both to alerting the operator and to contacting OPTIMUS to trigger a new re-deployment simulation for the application, thus starting a re-adaptation process.

More details on ADAPT architecture can be found in DECIDE deliverable D4.1 [10] .

3.4.1.2 Behavioural description

The interactions of ADAPT with other tools of the DECIDE framework are shown in the following Figure 21.

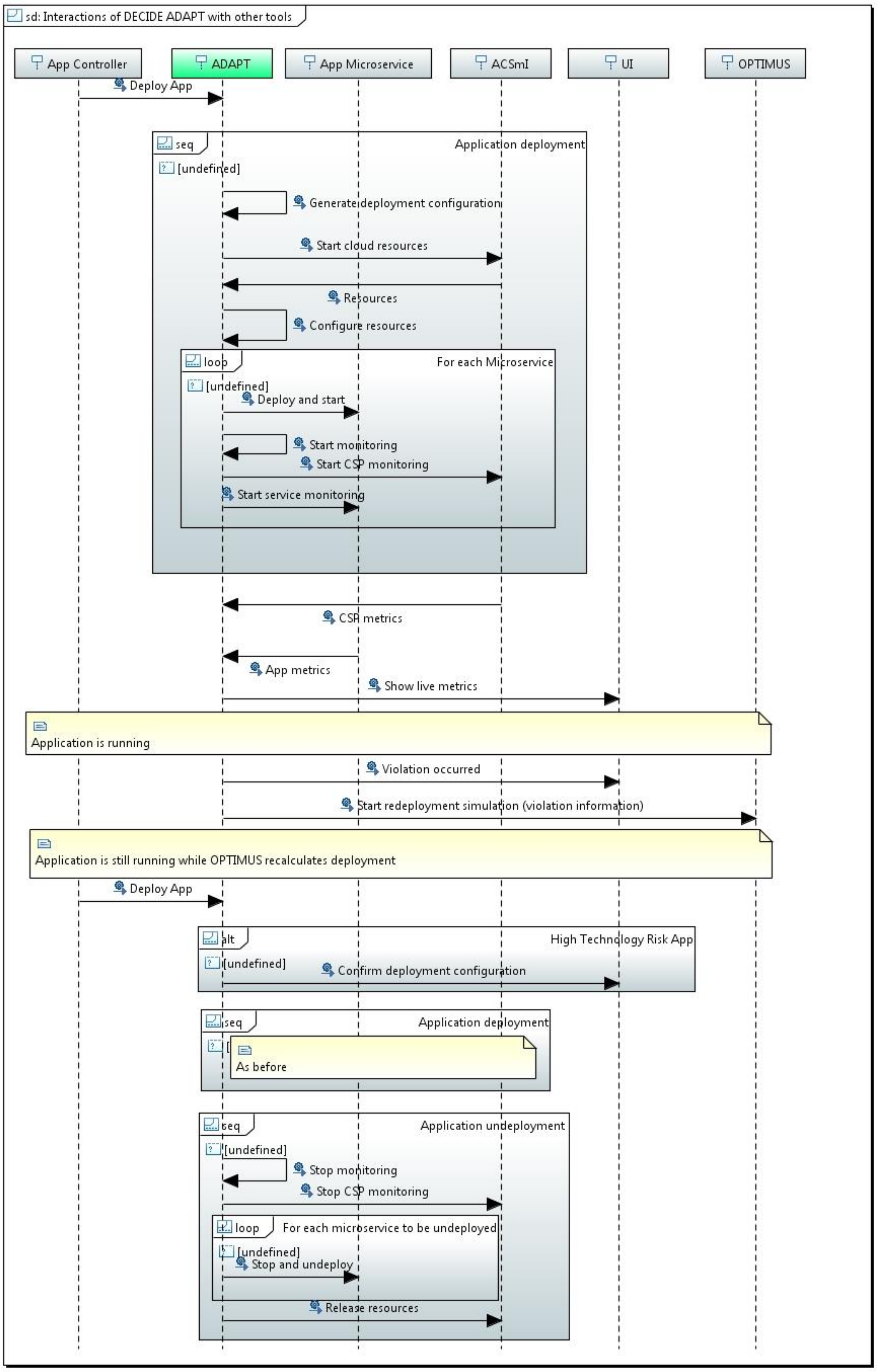

Figure 21. Interactions of ADAPT with other DECIDE tools

The main steps indicated in the sequence diagram of Figure 21 are the following.

1. When ADAPT is invoked to deploy an application, it generates a representation of the deployment configuration generated by OPTIMUS that can be understood by the used deployment tool (e.g. Terraform).
2. ADAPT contacts ACSmI to start the cloud resources indicated in the Application Description and then configures them.
3. ADAPT deploys and starts each application's microservice and initiates the related monitoring.
4. While the application is running, ADAPT collects application metrics from the microservices and from the underlying CSPs, through ACSmI, to identify any MCSLA violation.
5. As soon as a violation is identified, the operator is informed and a new redeployment simulation is triggered by contacting OPTIMUS and sending it the information about the violation.
6. When OPTIMUS finishes recalculating the new deployment configuration, ADAPT is invoked again to redeploy the application.
7. If the application level of technology risk is defined as high, the operator must confirm the new redeployment configuration before executing the actual redeployment.
8. The new configuration is deployed using the same steps as the initial deployment.
9. The previous configuration is un-deployed and the related resources are released through ACSmI.

3.4.2 MCSLA Editor

The MCSLA Editor provides a tool for the authoring of an MCSLA to be used as a contract between the user of the application and the application owner, i.e. developer. Furthermore, the MCSLA is designed in a machine-readable format that describes means to monitor and measure the application's NFRs.

3.4.2.1 Structural description

The requirements elicited for the MCSLA Editor in the project are described in D2.1 [1]. These are translated into functionalities that reside in components as denoted in the component diagram (see Figure 22). The main functionalities for MCSLA editor are:

- Provide means for the developer, to support him in the definition of the composite MCSLAs (Multi Cloud Service Level Agreement) and the corresponding SLOs (Service Level Objectives) of the application and the dependencies and needs on the underlying (combination of) cloud services in a machine-readable format for the representation.
- Provide means to translate the resulting CSLA in machine readable format (based on standards) as well as a human readable format (to be shared with the end-users, i.e. customers).
- Provide a UI (through the DevOps framework) for creating/editing CSLAs/MCSLA

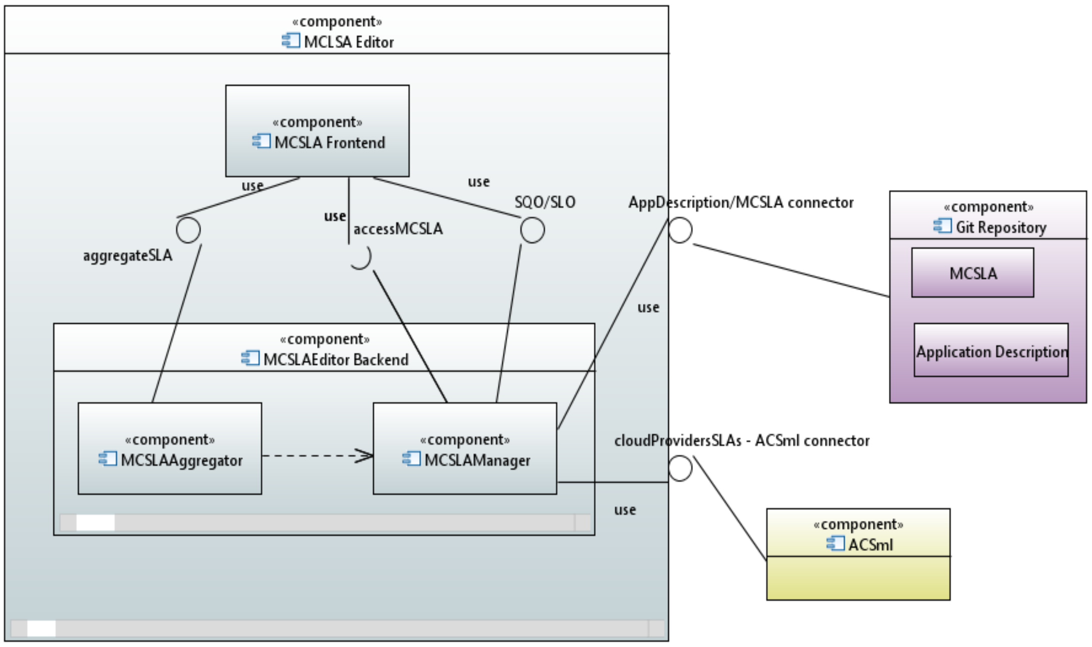
«component» MCLSA Editor
«component» MCSLA Frontend
use
use
use
aggregateSLA
accessMCSLA
SQO/SLO
AppDescription/MCSLA connector
«component» Git Repository
MCSLA
Application Description
use
«component» MCSLAEditor Backend
«component» MCSLAAggregator
«component» MCSLAManager
cloudProvidersSLAs - ACSml connector
use
«component» ACSml

Figure 22. Component Diagram for MCSLA Editor

The MSCLA Editor is a two-tier architecture represented by the *MCSLA Frontend* and the backend consisting of the *MCSLAAggregator* and *the MCSLAManager.*

MCSLA Frontend

The *MCSLA Frontend* is a user-facing component that enables the users to create, read, update and delete MCLSAs in a visual and human readable manner. The frontend will be integrated into the DevOps Framework. The Frontend communicates with the backend and uses defined interfaces for accessing available SQOs and SLOs, aggregated values of SLAs as well as existing MCSLAs. Available SLOs and SQOs are based on the ISO Standard 19086 [11] and cover terms that are application specific, rather than just provider specific.

MCSLAManager

The *MCSLA Manager* is in charge of managing the MCSLA and holds its logical information model, it communicates with the code repository in order to access the *Application Description* and receive the ids of the cloud providers the multi-cloud application is deployed on.

The *MCSLA Manager* uses this information from the *Application Description* to access cloud provider related information via the interfaces provided by ACSmI. This information is in turn used to identify the SLAs (SLOs) that need to be aggregated and represented in the MCSLA.

Furthermore, the *MCSLA Manager* is in charge of storing a tagged version of the MCSLA in the code Repository for ADAPT to access and be able to monitor the application. It does so via the Persistence Manager.

MCSLAAggregator

The *MCSLA Manager* serves the *MCSLA Aggregator* with the SLA's in order for it to accumulate and aggregate the possible values for SLOs depending on the aggregation rules defined in the component.

For each deployment scenario detailed in the *Application Description* a specific aggregation rule is specified and used to aggregate the values.

3.4.2.2 Behavioural description

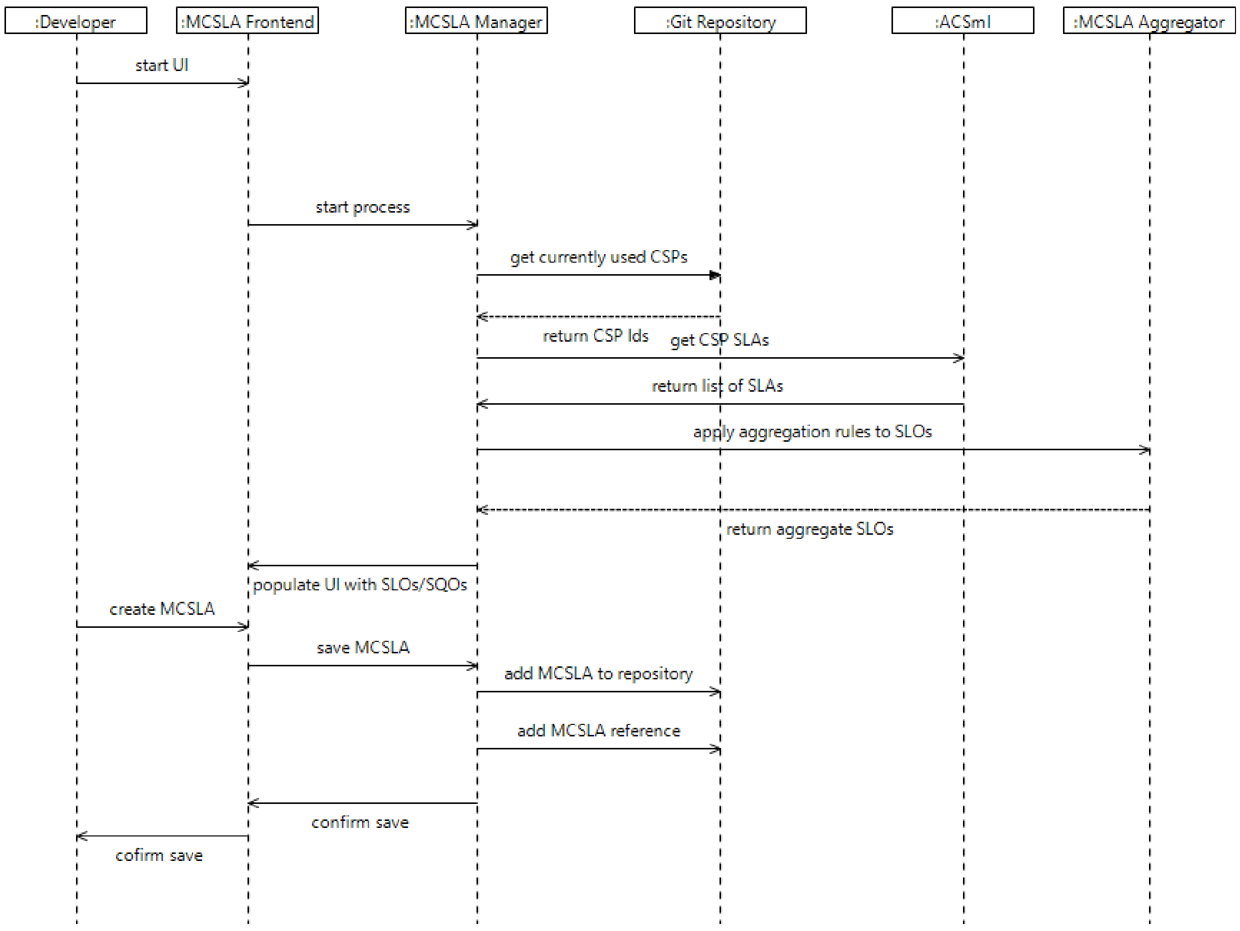

Figure 23. Sequence Diagram for creating an MCSLA

The sequences for creating an MCSLA are as follows:

1. The developer starts the *MCSLA Frontend* (GUI); this process calls the *MCSLA Manager* in order to populate the front end with the necessary values.
2. As long as the *MCSLA Editor* as a whole is integrated into the DevOps Framework, it is clear which *Application Description* is applicable at this stage. The *Application Description* residing in a repository will be accessed via the *MCSLA Manager* to retrieve the currently used deployment topology, i.e. the CSP Ids.
3. With the CSP Ids, the *MCSLA Manager* contacts *ACSmI* in order to obtain the contracted SLAs.
4. The *MCSLA Manager* then contacts the *MCSLA Aggregator* to take the necessary measures to aggregate the SLOs defined in each SLA.
5. Once this step is completed, the *MCSLA Manager* populates the frontend with the available SLO/SQOs and their possible values.
6. The developer then uses the GUI to create the MCSLA, which is in turn saved by the *MCSLA Manager* in the code Repository as well as registering it in the *Application Description.*

4 DECIDE tool suite deployment

4.1 DECIDE tools deployment options

The different DECIDE Key Results, tools and components, whose functionality and interdependencies have been described in previous sections, constitute the comprehensive DECIDE tool-suite.

This section briefly classifies DECIDE component attending to the technological framework required for its execution and deployment. According to this, DECIDE components are classified into:

- **Web tools (SaaS),**

  Are tools accessible through any compatible browser: In DECIDE some of the tools will be offered as service, which are invoked for performing different functionalities. Internally these tools can be deployed following the multi-cloud approach. It is envisioned that the following DECIDE Key results, tools and components will be offered as Web Tools (SaaS): DevOps framework, ACSmI, ADAPTand MCSLA Editor.

- **Containerized tools.**

  This case is similar to the previous one, but in this case the web application can be installed locally. The difference between this case and the previous one depends on the exploitation model to be decided for each Key Result, tool or component. The following tools fit into this category: DevOps framework, ACSmI, ADAPT, ARCHITECT and MCSLA Editor.

- **Eclipse RCP Plugins**.

  Are accessible within any compatible standalone Eclipse IDE. These tools need to be installed locally in each of the local IDEs. The tools envisioned to be deployed as Eclipse RCP plugins are: NFR Editor, ARCHITECT, and OPTIMUS.

- **Others:**

  App Controller will be delivered as a Java Library.

4.2 Information exchange between DECIDE tools

The mechanisms supporting interoperability and information exchange among the different components of the tool-suite are envisioned as follows:

- Information exchange through the Application Description: The Application Description is the main mechanism in DECIDE to share information between Key Results, tools and components in DECIDE. The Application Description is a structured JSON file containing all the relevant information about the current status of the relevant information of the multi-cloud application, focusing on the information that is relevant for the different DECIDE Key Results, tools and components. More information about the current version of the App Description is included in the Annex 1.
- Information exchange through the DECIDE DevOps Framework: DECIDE DevOps Framework, will also support interoperability among other DECIDE Key Results, tools and components.
  On the one hand, DECIDE DevOps Framework contains means to store the multi-cloud application itself (to be accessed by other DECIDE components) through the software repository, as well as related elements (configuration files, needed libraries or the App Description JSON file).

On the other hand, the DECIDE DevOps Framework will integrate the different UIs providing the integrated framework for accessing all the DECIDE components, following the complete and extended DevOps approach cycle defined.
Apart from integrating the DECIDE components from the interfaces point of view (UIs) it also will provide the means for supporting the DECIDE workflow (the activities over the different DECIDE tools) including the integration and management of the tools' triggers at logical level, through the access control and storage of the App Description and the management of the triggers of the different tools depending on the workflow (design, pre-deployment, integration and testing, operation and monitoring).

- Information exchange through direct API invocation. This approach is suitable when a direct message exchange is required between two DECIDE components.

Interactions between components within the DECIDE tool-suite could be driven by:

- User interactions: When a user requests a DECIDE component. This is done through the UI of the tool itself or the integrated UI (DevOps Framework).
- Task internal transactions: When a DECIDE tool requests information from another DECIDE tool (one component invokes another component, or the invocation is done through the DevOps framework).

5 Conclusions

This document provides a detailed description of the entire DECIDE global architecture, providing a conceptual, functional and interoperable representation.

The combination of DECIDE Key Results and related tools and components supports the extended DevOps approach proposed, from the design of the multi-cloud applications to the operation and monitoring of their working conditions.

The document also provides a deeper analysis; both structural and behavioural of each DECIDE Key Result identifying message exchange dependencies, dependencies through required and exposed interfaces, and the timeline activities conducted by the tools. This in-depth analysis has enabled an earlier identification of potential misalignments between the conceptualization and the technical design of the different tools. These misalignments where identified and addressed during the specification of this architecture, and solved and translated to the current conceptual and technical design (by their respectively work package tasks). This architecture also enabled the agreement on the work products produced and consumed by each tool, starting the discussions on the interoperability requirements between the DECIDE Key results, tools and components.

Additionally, this document addressed the challenge of supporting the deployment of the DECIDE tool-suite and the key results separately. Different possibilities are proposed here. The interoperability between the DECIDE Key results and components has been analysed in the document too, deriving in the incorporation of the App Description as one of the main mechanism for the DECIDE key results, tools and components to exchange information between them.

The architecture, deployment possibilities and interoperability requirements and mechanisms presented in this document cover the ideas, discussions and initial technical decisions taken by the DECIDE partners during the first year of the project. This document will be updated in a subsequent versions in M23, along with the advances in the implementation of the DECIDE Key Results, tools and components.

References

[1] DECIDE consortium, "D2.1 Detailed requirements specification," 2017.

[2] DECIDE Consortium, "DECIDE Annex 1 - Description of Action," 2016.

[3] DECIDE Consortium, "D5.1 ACSmI requirements and technical design," 2017.

[4] DECIDE Consortium, «D6.1 Initial Use Case Requirements Capture,» 2017.

[5] DECIDE Consortium, «D6.2 Final Use Case Requirements Capture,» 2017.

[6] Weaveworks, "Sock Shop: A microservices demo application," [Online]. Available: https://microservices-demo.github.io/. [Accessed 25 10 2017].

[7] A. Sandor, "Sock Shop application internal design," [Online]. Available: https://github.com/microservices-demo/microservices-demo/blob/master/internal-docs/design.md. [Accessed 21 11 2017].

[8] DECIDE consortium, «D3.4. Initial profiling and classification techniques,» 2017.

[9] Decide Consortium, «D3.10 Initial multi-cloud native application controller,» 2017.

[10] DECIDE Consortium, "D4.1 Initial DECIDE ADAPT Architecture," 2017.

[11] International Organization for Standardization, «ISO/IEC DIS 19086-2 Information technology -- Cloud computing -- Service level agreement (SLA) framework -- Part 2: Metric model,» ISO, 2017.

Annex 1: APP DESCRIPTION

This annex includes the current version of the App Description (M12)[4]. As explained in section 4, the App Description is one of the main means in DECIDE for interoperability between the different components.

This version of the App Description, resulting from Partners' research and discussions, includes relevant information for each of the DECIDE components, with a brief description for each field. Some all the fields are still empty as the information needs to be further discussed. Please note that the Application Description definition has evolved since the first data model shown in the DECIDE deliverable D2.1 [1], and it is still evolving in parallel with the design and implementation iterations.

Table 1. Application Description model for deployment

Field name	Nested Elements	Nested Elements	Type	Multiplicity/ Default	Description	Responsible component
id			String	1	Unique identifier for the Application Description	DevOps Framework
name			String	1	Name of the application	DevOps Framework
description			String	1	Textual description of the application	
highTechnologicalRisk			Boolean	1	Indicates if the application has high technological risk: confirmation for (re)deployment is needed	
version			String	1	Indicates the version number of the app description "schema", for compatibility purposes	
microservices			Array of Objects	1..*	List of microservices	DevOps Framework /OPTIMUS
	id		String	1	Unique Identifier for the microservice	DevOps Framework /OPTIMUS
	repo		String	1	Reference to location of microservice repo	DevOps Framework /OPTIMUS
	name		String	1	Human readable name for the microservice	Creation Wizard/OPTIMUS
	programmingLanguage		String	1	Type of programming language used for microservice (hint)	DevOps Frameworkd/OPTIMUS

[4] This version corresponds to the 17th of November 2017, when this document was last updated. The information of the App Description is evolving and will change as the technical discussions advances.

Field name	Nested Elements	Nested Elements	Type	Multiplicity/ Default	Description	Responsible component
	container_ref		String	1	Id or URI of container in which the microservice is located for ADAPT to be able to deploy it	DevOps Framework
	endpoints		Array of Objects (URI)	1..*	List of URI to access the services and their methods[5]	DevOps Framework /OPTIMUS
	stateful		Boolean	1	Is the microservice stateful or stateless?	DevOps Framework /OPTIMUS
	Type				The type of the microservice, as the result of the classification process.	OPTIMUS
	patterns		Array of Objects	0..*	List of patterns applied to the microservice	ARCHITECT
	dependencies		Array of Strings	0..*	List of microservice names the current one depends on	DevOps Framework
	nfrs		Array of Strings	1..*	List of selected NFRs per microservice	NFR Editor
	publicIP		Boolean	1	True if the microservice has a public IP address	OPTIMUS Classification
	infrastructure_requirements		Object	1	Requirements for the infrastructure hosting the microservice	OPTIMUS Classification
		disk_min	String	1		OPTIMUS Classification
		disk_max	String	1		OPTIMUS Classification
		RAM_min	String	1		OPTIMUS Classification
		RAM_max	String	1		OPTIMUS Classification
	Detachable_resource				list of elements that are going to be used by the microservice as for example external DB services	OPTIMUS Classification
		resource	Char		Defined at this time persistency and DB	OPTIMUS Classification
		SQL	Boolean			OPTIMUS Classification
		Specific_DB			When the type is DB, the user can specify which kind of DDBB will	OPTIMUS Classification

[5] Port numbers in each URI are those exposed by the microservice, the container can be configured to map them to a different port number

Field name	Nested Elements	Nested Elements	Type	Multiplicity/ Default	Description	Responsible component
					be needed (mongoDB, etc…)	
		Size			For DB resource it will be SMALL, MEDIUM or LARGE	OPTIMUS Classification
app_mcsla			Object	1	The MCSLA for the application (see **¡Error! No se encuentra el origen de la referencia.**)	MCSLA Editor
App Nfrs			Array of Strings	1..*	List of selected NFRs for the application, might apply to individual NFRs	NFR Editor
Schema					List of pairs (microservice_id, cloudservice_id)	OPTIMUS.simulation
virtual_machines			Array of Objects	0..*	Description of the VMs that will host the containers	
	id			1		ACSmI/OPTIMUS
	csp_name		String	1	Name of the CSP providing this VM	ACSmI/OPTIMUS
	csp_id		String	1	Internal UUID for the CSP providing this VM	ACSmI/OPTIMUS
	RAM		String	1	Amount of memory (in GB)	ACSmI/OPTIMUS
	cores		Integer	1	Number of cores	ACSmI/OPTIMUS
	storage		String	1	Amount of disk space (in GB)	ACSmI/OPTIMUS
	image		String	1	Name of the VM image (identifies also the OS and its version)	ACSmI/OPTIMUS
	ACSmIEndpoint		String	1	Endpoint of the Cloudbroker (ACSmI) API, to which all the cloud resource provisioning requests are sent	ACSmI
	ACSmIUsername		String	1	Username for the Cloudbroker API access	ACSmI
	ACSmIPassword		String	1	Password for the Cloudbroker API access	ACSmI
	vmSoftwareId		String	1	Id of the software resource from the ACSmI registry. Represents the OS and version of the VM (e.g. “Ubuntu 16.04”)	ACSmI

Field name	Nested Elements	Nested Elements	Type	Multiplicity/ Default	Description	Responsible component
	vmResourceId		String	1	The id of the ACSmI VM resource, which represents the underlying CSP that will perform the real provisioning	ACSmI
	vmRegionId		String	1	The id of the “Region” where the VM will run, taken from the ACSmI registry (E.g.: Zrh, US Standard, …)	ACSmI
	instanceTypeId		String	1	The id of the “instanceType” which represents the combination of resources allocated to the vm (e.g. “2 Total cores, 2GB RAM)	ACSmI
	keyPairId		String	1	The id of the keypairs needed to access ACSmI resources (associated to the ACSmI user profile)	
	openedPorts		String	0..1	The comma separated list of ports to be open on the VM	Developer
	consulJoinIp		String	1	Address of the master Consul (service registry) node; if “self", it means that this VM will act as master	TBD: it will be the address of a node running ADAPT
	dockerPrivateRegistryIp		String	0..1	IP of a Docker private registry, which will host custom container image prepared by a developer that are not published to the public Docker Hub repository	Developer
	dockerPrivateRegistryPort		Integer	0..1	Port of the private Docker registry	Developer
	dockerHostNodeName		String	1	Name of the Docker node (referenced by the same field in each container definition)	Developer / OPTIMUS
containers			Array of Objects	1..*	Description of the containers that will host the microservices	
	Id		String	1	Id of the container	

Field name	**Nested Elements**	**Nested Elements**	**Type**	**Multiplicity/ Default**	**Description**	**Responsible component**
	containerName		String	1	Name of the container	Developer
	imageName		String	1	Name of the container image	Developer
	imageTag		String	1	Tag to identify the container in the registry	Developer
	dockerPrivateRegistryIp		String	0..1	IP of a Docker private registry, which will host custom container image prepared by a developer that are not published to the public Docker Hub repository	Developer
	dockerPrivateRegistryPort		String	0..1	Port of the private Docker registry	Developer
	dockerPrivateRegistryUser		String	0..1	Username to access the private Docker registry	Developer
	dockerPrivateRegistryPassword		String	0..1	Password to access the private Docker registry	Developer
	hostname		String	1	Hostname of the container	Developer
	restart		String	1	Attribute indicating the restart policy for this container (e.g. “always”)	Developer
	command		Array of Strings	0..*	Comma-separated list of commands to be passed to the container, as for the “CMD” Dockerfile specs	Developer
	entrypoint		Array of Strings	0..*	Comma-separated list of commands and parameter to be passed to the container, as for the “ENTRYPOINT” Dockerfile specs	Developer
	DockerHostNodeName		String	1	Name of the VM hosting the container	OPTIMUS
	networks		Array of Strings	0..*	This field will trigger the creation of one or more dedicated Docker network(s) on the container to allow two containers to see each	Developer / OPTIMUS

Field name	Nested Elements	Nested Elements	Type	Multiplicity/ Default	Description	Responsible component
					other in case it does not exist	
	volumeMapping		Array of Objects	0..*	Mapping of volumes from host paths to container paths	Developer
		hostPath	String	1	Path on the host	Developer
		containerPath	String	1	Path on the container	Developer
	environment		Array of Strings	0..*	List of comma-separated KEY=VALUE environment variables to be defined before starting the container, as for the "ENV" Dockerfile specs	Developer
	consulKvProviderNodeName		String	1	Name of the node hosting the Consul Key-Value provider	(TBD: it will be the node running ADAPT)
	addConsulService		Boolean (0|1)	0..1	Specify whether to register the service to a Consul service registry (enables basic health-check)	(TBD: it may be enabled by default)
	addConsulTraefikRules		Boolean (0|1)		Specify whether to add reverse proxy routing rules to the Consul K/V store (based on "Host:" header)	Developer
	portMapping		Array of Objects	0..*	List of ports to be published by this container	Developer
		hostPort	String	1	Port to be exposed on the host	Developer
		containerPort	String	1	Port exported by the container	Developer
	endpoints		Array of Objects	0..*	List of endpoints for this container	Developer
		protocol	String	1	Typically "http", but can be something else according to URL syntax	Developer
		port	Integer	1	The port to which the endpoint is bound	Developer
		skipRule	Boolean (0|1)	0..1	Set to 1 to discard the routing rule based on hostname ("Host:" header)	Developer

Field name	Nested Elements	Nested Elements	Type	Multiplicity/ Default	Description	Responsible component
		containerNameOverride	String	0..1	Overrides the standard routing rule based on hostname; hence, it allows to consider a different hostname for this service	Developer

The following tables describe the Application Description model for monitoring with a brief description for each field. Table 2 describes the nested elements for the field app_mcsla of the Application Description. The MCSLA Editor is responsible for eliciting this information from the user.

Table 2. Application Description model for monitoring the application via its MCSLA (nested elements for "app_mcsla")

Element Name	**app_mcsla**		
Description	General information about the MCSLA		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
Id	String	1	Unique Identifier for the MCSLA
description	String	1	This is MCSLA description line.
visibility	String	1	public or private
validityPeriod	Integer	1	The validity period of the MCSLA in days
microservice_SLAs	Microservice_SLAs	1..*	The list of SLAs for each microservice

The following table describes the fields nested in the microservice_SLAs field of the MCSLA.

Table 3. Nested elements for microservice_SLAs

Element Name	**Microservice_SLAs**		
Description	The general information about the SLAs for each microservice		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
Id	String	1	Unique Identifier for the microservice_SLA
ms_id	String	1	Unique Identifier of the microservice this SLA belongs to
csp_id	String	1	Unique Identifier of the CSP from which the SLA comes from
visibility	String	1	public or private
validityPeriod	Integer	1	The validity period of the SLA in days, should not be higher than that of the MCSLA
microservice_SLO	microservice_SLO	1..*	List of microservice SLOs
microservice_SQO	microservice_SQO	1..*	List of microservice SQOs

The following table describes the fields nested in the microservice_SLO and microservice_SQO fields of microservice_SLAs.

Table 4. Nested elements for microservice_SLO and microservice_SQO

Element Name	**microservice_SLO and microservice_SQO**		
Description	The general information about the slo or sqo defined for a microservice		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
Id	String	1	Unique Identifier for the microservice_SLA
termName	String	1	Name of the term to which it refers to
value	Integer	1	Term value that should not be violated based on calcualtion formula
unit	String	1	Term unit
comparisonOperator	String	1	Comparison operator for monitoring the SLO
violationTriggerRule	ViolationTriggerRule	1	The violation Trigger Rule
remedy	Remedy	0..1	the compensation available to the cloud service customer in the event the cloud service provider fails to meet a specified cloud service level objective
metrics	Metrics	1..*	The definition of how to measure the SLO or SLA
violation_report	String	0..1	Indicates where to report violations for this application (optional)

The following table describes the fields nested in the violationTriggerRule field of microservice_SLO and microservice_SQO.

Table 5. Nested elements for ViolationTriggerRule

Element Name	**ViolationTriggerRule**		
Description	The general information about the violation trigger rule		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
interval	string	1	Indicates the monitoring frequency for each SLO
breaches_count	Integer	1	The count of how many breaches have taken place

The following table describes the fields nested in the remedy field of microservice_SLO and microservice_SQO.

Table 6. Nested elements for Remedy

Element Name	**Remedy**		
Description	The general information about the compensation available to the cloud service customer in the event the cloud service provider fails to meet a specified cloud service level objective		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
type	String	1	The type of remedy the cloud service provider will be offering the cloud service customer
value	Integer	1	The value of the type of remedy offered by the cloud service provider
Unit	String	1	The unit for the value offered
validity	Integer	1	The validity period for this remedy

The following table holds the fields (taken directly from ISO 19086-2 Metric Model [11]) that are nested within the metrics field of microservice_SLO and microservice_SQO. The MCSLA Editor is responsible for eliciting this information from the user.

Table 7. MCSLA Metric data model for monitoring

Element Name	Metric		
Description	The general information about the metric		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
Descriptor	String	0..1	a short description of the metric
Id	String	1	a unique identifier for the metric within a context
Source	String	1	the individual or organization who created the metric
Scale	enumeratedList	1	classification of the type of measurement result when using the metric. The value of scale shall be “nominal, ordinal, interval, or ratio”. SLOs shall use either the “interval” or “ratio” scale. SQOs shall use the “nominal” or “ordinal” scales.
Note	String	0..1	additional information about the metric and how to use it.

category	String	0..1	a grouping of metrics with similar expressions, rules, and parameters
expression	Expression	0..1	The expression of the calculation of the Metric and supporting information. An SLO metric shall have an expression while an SQO may or may not have an expression (e.g., specified using natural language). It shall be written using the ids to represent UnderlyingMetrics, Parameters, and Rules.
parameters	Parameter	0..*	a Parameter is used to define a constant (at runtime) needed in the expression of an Metric. A Parameter may be used by more than one Metric if it is defined using a unique ID within the set of metrics it is used in.
rules	Rule	0..*	a Rule is used to constrain a Metric and indicate possible method(s) for measurement.
underlyingMetrics	Metric	0..*	a metric element that is used within an expression element to define a variable. The Expression shall use the Underlying Metric id to reference the Underlying metric within the expression.

The following table describes the fields nested in the expression field of a Metric.

Table 8. Nested elements for Expression

Element Name	**Expression**		
Description	The expression of the calculation of the **Metric** and supporting information		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
Id	String	1	a unique identifier (within the context of the metric) for the expression
expression	String	1	the expression statement written using the ids to represent UnderlyingMetrics, parameters, and rules.
expressionLanguage	String	1	the language used to express the metric (i.e. ISO80000)
note	String	0..1	additional information about the expression
unit	String	0..1	

		required when scale is ratio or interval	real scalar quantity, defined and adopted by convention, with which any other quantity of the same kind can be compared to express the ratio of the two quantities as a number.
subExpression	Expression	0..*	an associated element of type element that is used within the expression to define a variable. The Expression shall use the SubExpression id to reference the SubExpression within the expression.

The following table describes the fields nested in the parameters field of a Metric.

Table 9. Nested elements for Parameter

Element Name	**Parameter**		
Description	A Parameter is used to define a constant (at runtime) needed in the expression of a Metric. A Parameter may be used by more than one Metric if it is defined using a unique ID within the set of metrics it is used in.		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
id	String	1	the unique identifier of the parameter
parameterStatement	String	1	the statement or value of the parameter
unit	String	1	the unit of the parameter
note	String	0..1	additional information about the parameter

The following table describes the fields nested in the rules field of a Metric.

Table 10. Nested elements for Rule

Element Name	**Rule**		
Description	A Rule is used to constrain a Metric and indicate possible method(s) for measurement. For instance, an "AvailabilityDuringBusinessHour" Metric could be defined with a scope that constrains some piece of a generic "Availability" Metric element that limits the measurement period to defined business hours. A Rule describes constraints on the metric expression. A constraint can be expressed in many different formats (e.g. plain English, ISO 80000, SBVR)		
attribute -or- Element	**Type**	**Multiplicity / Default**	**Definition**
Id	String	1	the unique identifier for the rule
ruleStatement	String	1	a constraint on the metric
ruleLanguage	String	1	the language used to express the rule in the ruleStatement
Note	String	0..1	additional information about the rule