MULTICLOUD APPLICATIONS TOWARDS THE DIGITAL SINGLE MARKET

Deliverable D3.15

Final multi-cloud native application composite CSLA definition

Editor(s):	Simon Dutkowski
Responsible Partner:	Fraunhofer
Status-Version:	Final – v1.0
Date:	29/05/2019
Distribution level (CO, PU):	CO

Project Number:	GA 726755
Project Title:	DECIDE

Title of Deliverable:	D3.15 Final multi-cloud native application composite CSLA definition
Due Date of Delivery to the EC:	31/05/2019

Workpackage responsible for the Deliverable:	WP3 - Continuous Architecting
Editor(s):	Fraunhofer
Contributor(s):	Lena Farid (Fraunhofer) Simon Dutkowski (Fraunhofer)
Reviewer(s):	Marisa Escalante (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP5, WP4, WP2

Abstract:	This software deliverable will comprise the final version of a tool to derive composite SLAs from elementary ones. For this, a description formalism will be defined and extended if needed. Range definition for SLA metric values, composition and matching rules will be defined and implemented.
Keyword List:	MCSLA, SLA, SLO, SQO, Editor, Multi Cloud,
Licensing information:	The software is licensed under the GNU Affero General Public License Version 3. The document itself is delivered as a description for the European Commission about the released software, so it is not public.
Disclaimer	This deliverable 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	20/05/2019	First draft version	Simon Dutkowski (Fraunhofer)
v0.2	24/05/2019	PDF Export	Simon Dutkowski (Fraunhofer)
v0.3	25/05/2019	ACSmI interface usage	Simon Dutkowski (Fraunhofer)
v0.4	27/05/2019	Initialize MCSLA patterns	Simon Dutkowski (Fraunhofer)
v0.5	29/05/2019	Incorporate comments	Simon Dutkowski (Fraunhofer)
V1.0	30/05/2019	Ready for submission	Leire Orue-Echevarria (TECNALIA)

Table of Contents

List of Figures

List of Tables

Terms and abbreviations

ACSmI	Advance Cloud Service meta-Intermediator
ADAPT	Application Deployment and Adaptation
API	Application Programming Interface
CO	Confidential
CRUD	Create, read, update, delete
CSLA	Composite Service Level Agreement
CSP	Cloud Service Provider
DECIDE	DEvOps for trusted, portable and interoperable multi-Cloud applications towards the Digital singlE market
DevOps	Development and Operation
EC	European Commission
GA	Grant Agreement
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
IaaS	Infrastructure as-a-Service
ID	Identifier
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
MCSLA	Multi-cloud Application Service Level Agreement
MTBF	Meantime between failure
MTTR	Meantime to recover
NFR	Non-Functional Requirement
OMG	Open Management Group
PDF	Portable Document Format
PU	Public
QoS	Quality of Service
REST	Representational State Transfer
SaaS	Software as-a- Service
SBVR	Semantics of Business Vocabulary and Rules
SLA	Service Level Agreement
SLO	Service Level Objective
SPA	Single Page Application
SQL	Structured Query Language
SQO	Service Qualitative Objective
ToC	Table of Content
UI	User Interface
URL	Uniform Resource Locator
WS	Web Service

Executive Summary

The MCSLA Editor plays a vital role in the DECIDE project as it defines the agreement between the multi-cloud native application developer and the end-user of the application services. Furthermore, it is in a standards-based machine-readable form that allows for other DECIDE tools such as ADAPT to monitor the application and assess whether the expected QoS is guaranteed.

One of the main innovations of this component is the adaptation of the upcoming ISO/IEC 19086 standard, especially of part 2 [1], which describes a technical machine-readable model for metrics. By providing SLAs in a standardized format, cloud service providers and their services are better comparable for customers. A common machine-readable format also enables the implementation of aggregation patterns for SLAs of different cloud service providers in service composition scenarios [2].

The document at hand describes the final version of the prototype and the representation of the machine-readable MCSLA definition adapted from the ISO standard. It is a revised version of the intermediate document with the same title [3] and contains content that is reused. The prototype includes a backend and frontend that communicate via a restful interface. The prototype allows developers to define the MCSLA through a convenient graphical user interface.

The previous intermediate version of the document described the conceptual work done for the MCSLA, including the makeup of the MCSLA and its properties. Furthermore, it already laid out the functional and technical properties of the prototype along with the build and installation instructions. A user manual was added to the document to explain the usage of the user frontend.

In addition to the intermediate version described in "*Intermediate multi-cloud native application composite CSLA definition*", D3.14 [3], this final document describes the extension of the prototype with the ability to export the MCSLA as PDF document. Furthermore, a deeper insight is given to the communication with the ACSmI discovery service. The processes of automatic inferring of aggregated service objectives out of NFRs and microservices for an initial MCSLA proposal to the developer is outlined in detail. Aggregation patterns identified and defined so far are now practically implemented. In addition, the portfolio of service objective metrics and calculation expressions is completed to support all monitoring scenarios in the context of DECIDE.

1 Introduction

1.1 About this deliverable

The document at hand represents the documentation for the prototype delivered at M30 for the task T3.5 Multi-cloud native application composite SLA description. It also presents concepts that have been defined within this task for the MCSLA definition.

1.2 This document amends, updates and extends previous version of this deliverable [3], by including corrections, extensions and modifications throughout the whole text. Complete new sections in this document are 3.1.2, 3.2.5, 3.2.6, and 4.2.2.6.Document structure

This document is divided into four main sections. Section 2 presents the MCSLA concept defined in the project. It describes the makeup of an MCSLA, its properties and the Aggregation Patterns for the different deployment topologies. Section 3 describes the implementation details from a functional and technical perspective and section 4 describes the build and installation instructions as well as the user manual for using the tool.

Finally, at the end of the document section 5 concludes on the outcome of M30 and gives some recommendations for further improvements.

2 MCSLA Concept

It is defined in the DECIDE project that once the multi-cloud native application is implemented and is ready for deployment, i.e. the most optimal deployment topology has been selected, an MCSLA has to be defined. The purpose of the MCSLA is twofold i) it acts as the contract between the end-users and the developer of the multi-cloud native application and ii) it is used for monitoring purposes by ADAPT and ACSmI and will be assessed in runtime to ensure it is being accomplished.

In order for the latter to be realised, the task T3.5 of WP3 is responsible for implementing the following two main points:

1. Enable the seamless composition of an MCSLA via an editor. This should also support the composition of MCSLAs when an application is self-adapted to a new deployment topology.
2. Define a standard-based machine-readable format for an MCSLA in order to be processed by the DECIDE tools.

In the following sections the set up and structure of a general MCSLA for use within the DECIDE project is outlined. The second part develops and describes the fundamental concepts for aggregation patterns that form the base of the multi-cloud aspects of an application service level agreement.

2.1 Make up of a MCSLA

The accumulation of a number of SLAs from different CSPs is defined in the DECIDE project as a multi-cloud native application composite SLA (MCSLA).

A cloud SLA is typically composed of a number of Service Qualitative Objectives (SQO) and Service Level Objectives (SLO) as defined in ISO/IEC 19086-1 [4]. The SLOs and SQOs represent, among others, the non-functional requirements of an application and its underlying infrastructure. We will refer to them as terms.

In the multi-cloud context, the deployment of the microservices of an application take place on several CSPs. Each CSP contracted shares with the developer an SLA that guarantees an expected quality of service (QoS) of the cloud service in use. Therefore, in a multi-cloud deployment scenario there will be at least two of these agreements. These agreements might differ in their content but might also include same terms (SLOs or SQOs) but with different values.

An MCSLA must therefore act as an aggregator of all terms defined in the various SLAs. If term occurs in several different SLAs, the values of the term must be aggregated (based on defined mathematical functions). For example, if the SLO Availability occurs in one SLA with a value of 99% and in another with a value of 99%. Then the MCSLA should contain the SLO Availability with the value of 98% (formula is presented in Section 2.2). This is in essence the maximum value for availability that the developer may offer to the end-user, as it is not guaranteed that an outage would take place across all microservices (or CSPs for that matter) at the same time.

Concerning end-users, the developer may also define application specific terms. These additional terms pertain to the application, are consumer-oriented and not derived from the CSP SLAs. These can be terms the developer needs monitored and/or agreed with the end-user. An example would be the application's response time. A graphical representation of this approach is shown in ***Figure 1***.

Furthermore, it is important for the MCSLA to reflect the diversity in the contracted SLAs on CSP level and the system hierarchy (IaaS, SaaS) – these need to be consolidated.

Moreover, an SLA term can be hard or soft. This is important for monitoring purposes. Hard terms are to be observed at all time, those declared as soft do not pose a major risk.

Another aspect concerning an MCSLA is re-deployment. Once an MCSLA has been set and communicated to the end-user, certain terms should not be changed. A solution would be to define two layers: an external one, which has to be respected and cannot be changed when a re-deployment should take place, and an "internal" one that collects the SLAs from the various providers where the application has been or will be deployed. The external SLA is a composition of the SLAs in the internal part, plus the application SLOs. In case of a candidate redeployment involving different services or CSPs, the internal SLAs change accordingly, but their composition should still satisfy the external SLA for the candidate to be acceptable. If no such candidate exists, the adaptation (i.e. re-deployment) fails.

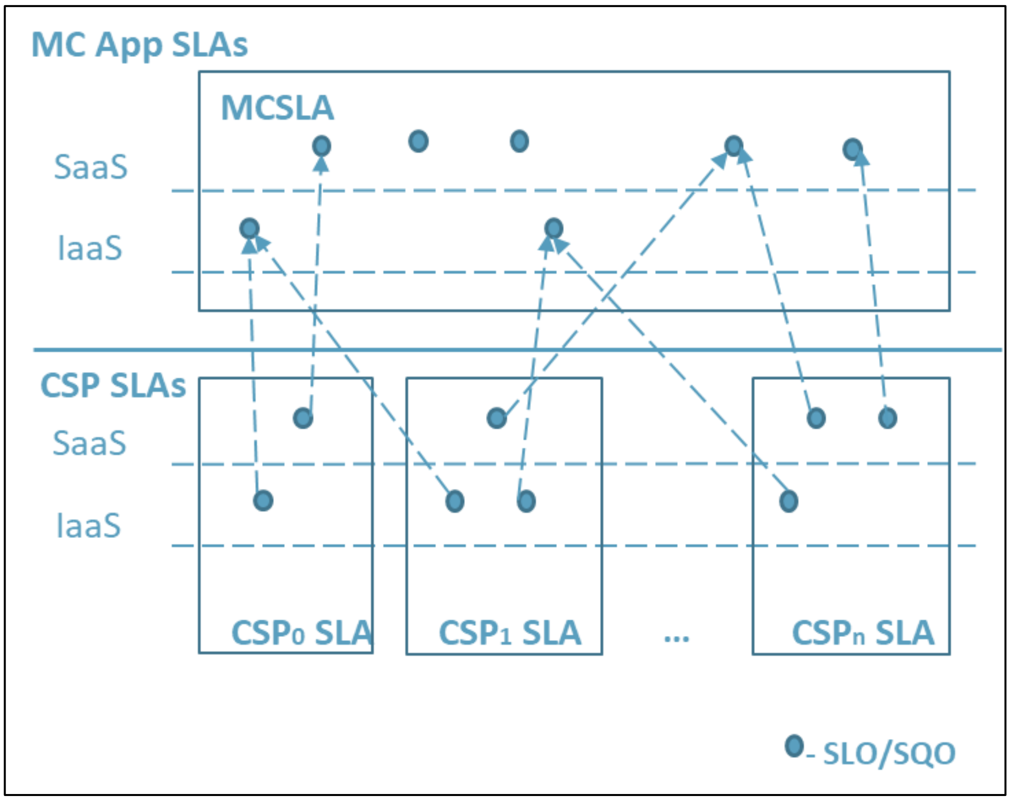


Figure 1. Conceptual Idea – Make up of an MCSLA

2.2 SLA Aggregation Patterns

In a multi-cloud deployment scenario, a minimum of two microservices are expected to be deployed on different CSPs or on different cloud services of the same CSP. In this case, there will be at least two SLAs contracted for the developer of the multi-cloud native application. As previously stated, these agreements might differ in their content but might include the same terms (SLOs) but with different values.

In order to reduce the complexity of managing a multitude of cloud services and cloud services provider, SLA Aggregation Patterns complemented with an aggregation engine are needed.

An SLA Aggregation Pattern [2] is a mathematical function that computes several terms into one aggregated term.

In [2], they introduce a type which extends the WS-Agreement [5] specification, which labels specific SLA terms with a type in order to be able to calculate and help automate the SLAs.

As the DECIDE project is not following the WS-Agreement specification but ISO/IEC 19086 [1] (parts 1-4) [4, 1, 6, 7], it may result fruitful to check if such a type is also required there. In any case, these types can also be internally depicted in ACSmI and the MCSLA Editor. This is one of the aspects to be investigated for improvements as part of future work.

In [2] seven Aggregation Patterns are defined as types[2]

$$Types = \{sumtype, maxtype, mintype, neutral, ORtype, ANDtype, XORtype\}$$

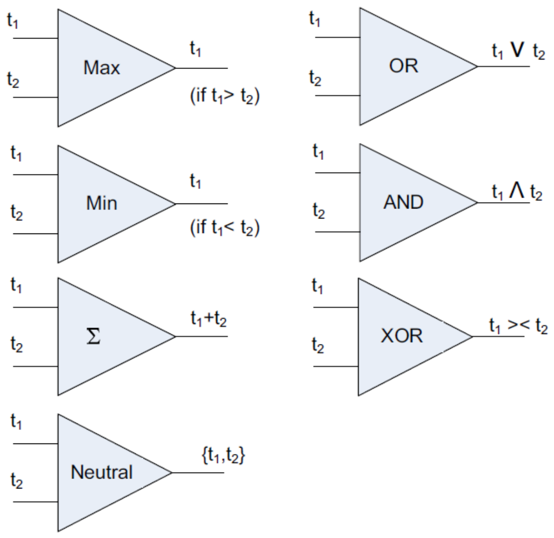


Figure 2. Term composition using Aggregation Patterns [2]

Figure 2 depicts the aforementioned types, the following three types are relevant in DECIDE context:

The *sumtype* function (denoted as $\sum$ in ***Figure 2***) defined as

$$sumtype \in Types(\Leftrightarrow sumtype : P(Terms) \rightarrow Terms)$$
$$sumtype(term_1, \dots term_n) = \sum_{i=1}^{n} term_i.term_s.value$$

can be used to calculate terms for storage space, memory, availability and cost in a deployment environment where all microservices are deployed on the same machine. Moreover, it assumes that all microservices will fail simultaneously, which is rarely the case. Therefore, it makes sense to extend this list with an additional type to fulfil DECIDE's needs in a multi-cloud context.

The *mintype* function defined as

$$mintype \in Types(\Leftrightarrow mintype : P(Terms) \rightarrow Terms)$$
$$mintype(term_1, \dots term_n) = \min_{1 \le i \le n} term_i.term_s.value$$

is an aggregation function that aggregates a number of terms into one term. The minimum of these terms is picked up and ultimately represents the aggregation of the input terms. Therefore, the only term having the minimum value will contribute to the final term in the MSCLA. A good example is given in [2], which is that for the bandwidth: "In a sequence of

[1] Part 2 and 4 are still under development.
[2] *For the full formalisation please see* [3]

activities the activity pertaining to the minimum bandwidth will become the bottleneck for the whole sequence making other activities with higher bandwidth ineffective.”

The *ORType* function defined as

$$ORtype \in Types \ (\Leftrightarrow ORtype : P(Terms) \rightarrow Terms)$$

$$ORtype(term_1, \dots term_n) = \bigvee_{i=1}^{n} term_i.term_s.value$$

ORtype is an aggregation function that aggregates a number of terms into one or more terms. It does so by applying a logical OR function on these terms and the result represents the aggregation of all the input terms. For instance, an application developer who wants to aggregate services of varying qualities but would also like to segregate them under different levels of SLAs, may use the *ORtype* aggregation function to fulfil his needs.

An example could be a reseller who buys computational resources of different speeds and qualities from different vendors and aggregates them using *ORtype* function so that later, he can offer SLAs of different levels such as gold, silver or bronze, etc. to its consumers and might prove interesting for developers.

2.2.1 SLA Aggregation Patterns for Availability

In this section we look at how to use these Aggregation Patterns for the NFR availability. The following patterns also take into consideration the different deployment topologies. Investigation to find more Aggregation Patterns can be part of future work.

Availability is probably the most important single metric that can be used to measure the performance of a service. It shows the time or percentage the service is operational and responding.

The following section gives examples using the three selected Aggregation Patterns for Availability.

Aggregated availability the sumType pattern in a basic multi-cloud environment

This example (see ***Figure 3***) for availability includes a web site, a SQL database and a table storage. The deployment has taken place on three different cloud services on three different CSPs. For the application to function as intended, each of these components must be working. They also each have a 99.9% availability guaranteed in their SLA. It cannot be assumed that the components will fail simultaneously, but at different times. This means that the summation of all terms using the *sumtype* function described above is insufficient and would yield a false value for Availability.

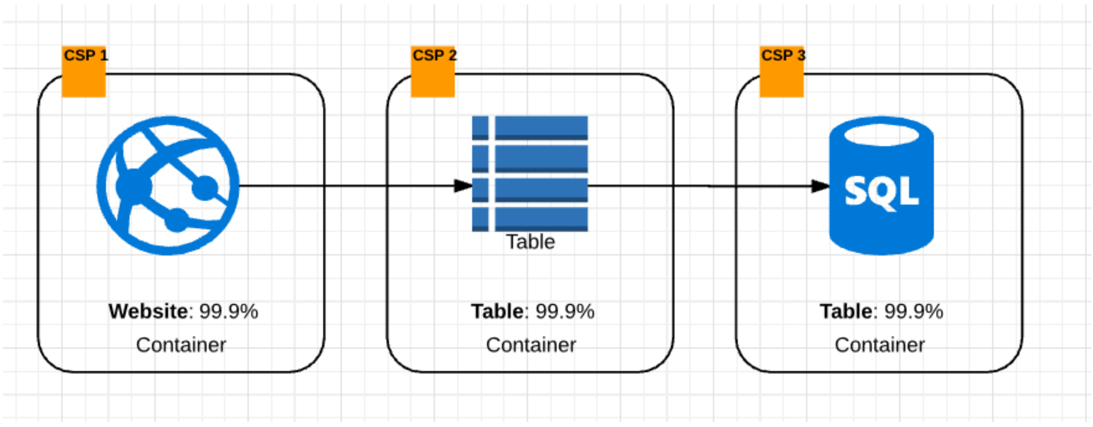


Figure 3. Basic multi-cloud deployment topology

Therefore, it is necessary to extend the list of types presented above with the following function to be applied for this deployment topology:

$$mcAvailability(term_1, \dots term_n) = 100\% - \sum_{i=1}^{n}(100\% - term_i)$$

The function takes a number of terms and creates a sum of the "unavailability" of all terms and deducts it from the optimal value for availability.

Example result would be as follows:

$$mcAvailability(99{,}9\%, 99{,}9\%, 99{,}9\%) = 99{,}7\%$$

<u>Aggregated availability the MINtype pattern in replication deployment topology</u>

In this example for availability, there is a web site, which is replicated on two CSPs in different Regions.

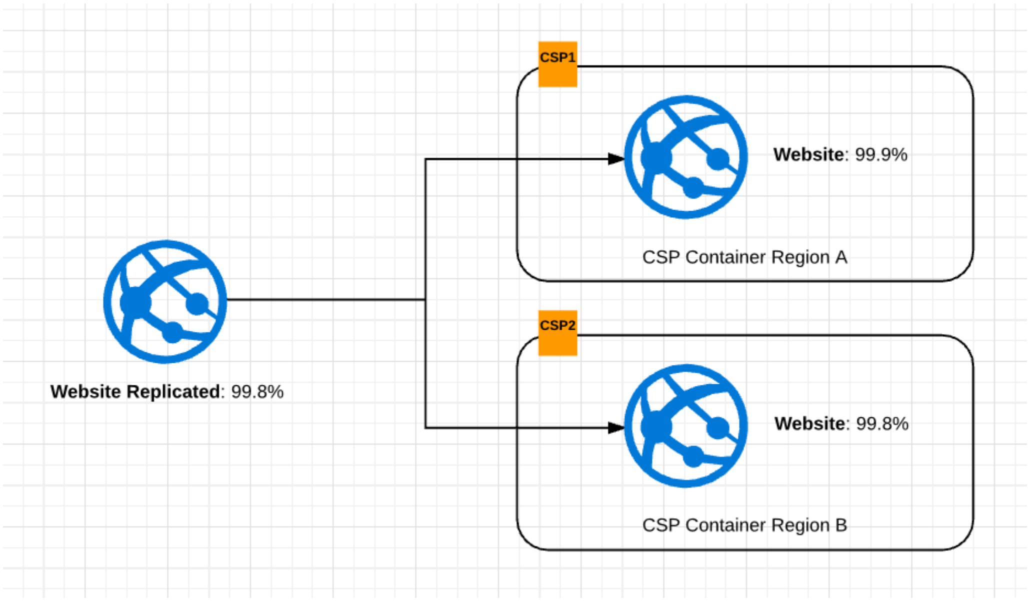


Figure 4. Multi-cloud replication deployment topology

In this example it is only viable to select the minimum term value based on the deployment topology. Otherwise, the availability term in the MCSLA would be false as it cannot be guaranteed.

The function to be applied for this deployment topology is as follows:

$$mintype(term_i, \dots term_n) = \min_{1 \le i \le n} \; term_i.term_s.value$$

Example result would be as follows:

$mintype(99{,}9\%, 99{,}8\%) = 99{,}8\%$

<u>Aggregated SLAs uptime the ORtype pattern for service composition from different vendors</u>

In this example, a generic database accesses two different SQL servers. One enjoys a 99,9% availability and the other only 99,7% availability. The deployment as depicted in ***Figure 5*** is across two different CSPs.

The developer may include, as described before, different plans for the consumers (e.g. bronze, silver, gold) and derive these from the different guaranteed quality for availability. Therefore, the developer can choose which server is part of which plan by integrating the respective availability value in the MCSLA.

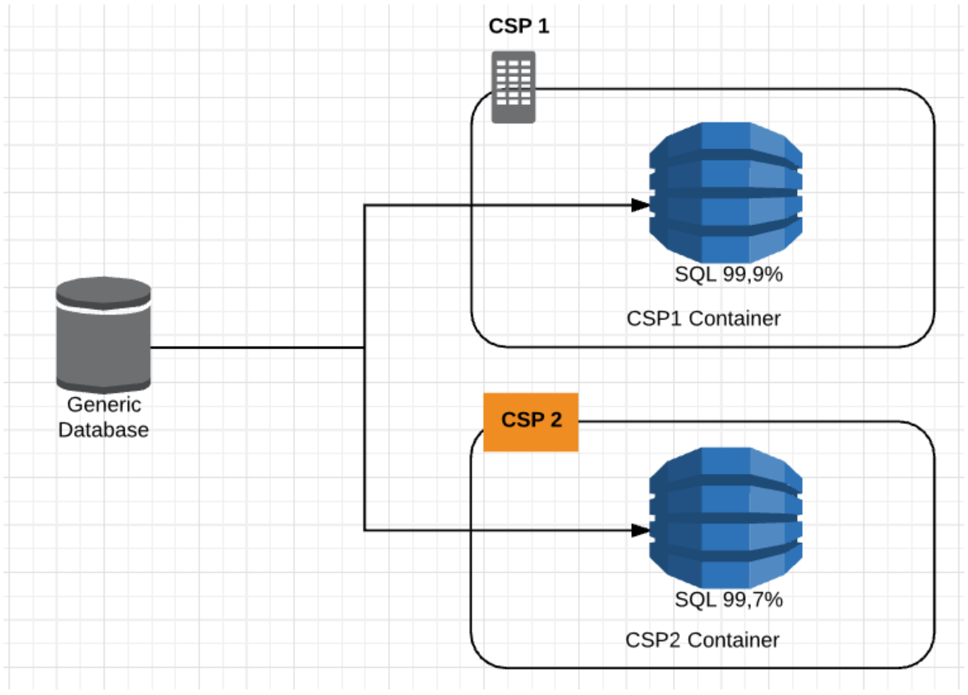


Figure 5. Different vendors topology

The function to be applied for this deployment topology is as follows:

$$ORtype(term_1, \dots term_n) = \bigvee_{i=1}^{n} term_i.term_s.value$$

Example results would be as follows:

$ORtype(99{,}7\%, 99{,}9\%) = 99{,}9\%$

$ORtype(99{,}7\%, 99{,}9\%) = 99{,}7\%$

3 Implementation

3.1 Functional description

The MCSLA Editor provides a tool for the authoring of a MCSLA to be used as a contract between the end-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 in respect to all contracted SLAs of the cloud services.

The main functionalities for the MCSLA Editor are as follows:

F1. To provide supportive means for the developer to define the composite MCSLAs and the corresponding service objectives (SLOs and SQOs) of the application. This includes:
   a. Aggregation of the available terms in the various contracted SLAs using defined mathematical formulas mapped to deployment topologies.
   b. Allowing for editing an existing MCSLA after a re-deployment is recommended whilst respecting the initial SLA

F2. To provide an interactive user interface for authoring an MCLSA

F3. To translate the MCSLA into a standards-based machine-readable form that includes a metrics definition. The MCSLA adapts the ISO/IEC 19086-2 [1] specification for metrics.

F4. To translate the MCSLA into a human readable form.

F5. To maintain an interface to ACSmI for accessing the contracted SLAs

F6. Maintain access to the git repository of the application.

F7. To store the MCSLA definition as part of the Application Description in a git repository to be accessed by the different DECIDE tools.

F8. To integrate the MCSLA Editor in the DECIDE DevOps Framework.

The MCSLA Editor has been implemented incrementally. The first version (M12) included the functionalities F1 (partly), F2, F3, F6, and F7. The second version (M24) contained improvements and finalisation of the previous list of functionalities and, in addition, F5 and F8. The final release (M30) includes more detailed elements also from a usability perspective, especially the coverage of F4, which is a PDF export of the MCSLA for better readability. ***Table 1*** opposes the defined requirements [8] for the MCSLA Editor to the listed functionalities F1 to F8, the coverage and the status in M30.

Table 1. Functionalities opposed to Requirements

Functionality	Req. ID	Coverage	Status
F1	WP3-CSLA-REQ1	The MCSLA Editor provides the model and CRUD functionality for the file and the mechanism for storing and accessing the MCSLA. Aggregation rules are integrated into the implementation of this prototype (M24).	Satisfied
F2	WP3-CSLA-REQ10 WP3-CSLA-REQ11	The MCSLA Editor is composed of a frontend and backend. The frontend is a web-based tool.	Satisfied
F3	WP3-CSLA-REQ6 WP3-CSLA-REQ7	The MCSLA definition is in machine-readable form and follows the standard ISO/IEC 19086-2 [1].	Satisfied
F4	WP3-CSLA-REQ6	The frontend allows the generation and export of the MCSLA as a PDF document.	Satisfied
F5	WP3-CSLA-REQ1	The MCSLA Editor utilizes the ACSmI discovery interface to retrieve the cloud service SLAs for deployment scenarios.	Satisfied

Functionality	Req. ID	Coverage	Status
F6	WP3-CSLA-REQ1	All the mechanisms for accessing the git repository are in place.	Satisfied
F7	WP3-CSLA-REQ1	All the mechanisms for storing the MCSLA in the defined application repository are in place.	Satisfied
F8	WP3-CSLA-REQ9	The MSCLA Frontend is integrated in adjusted form into the dashboard using the HTTP iframe tag.	Satisfied

The following list compiles the functionality that are added or extended in M30:

- The MCSLA Editor has a web-based UI that allows the user to view all available SQOs and SLOs and select from these terms the ones relevant for the application.
- The MCSLA Editor initializes all service objectives depending of the developer's defined NFRs and the contracted SLAs for the different cloud services. It applies the appropriate aggregation patterns to the metrics and adds the required formula expressions. The used metrics and formula expressions to calculate the values are shown to the user for transparency reasons.
- The MCSLA Editor frontend allows the end user to export the MCSLA external part as a PDF document as an offer to his customers.

3.1.1 Fitting into overall DECIDE Architecture

The MSCLA Editor is crucial for the DECIDE DevOps Framework as it is part of the continuous operation phase and lays the foundation for monitoring the multi-cloud native application as well as the contracted cloud services, which may lead to imperative re-adaptation and re-deployment of the application.

Furthermore, it serves as an interface (UI) through which the developers specify the multi-cloud SLAs agreed with the end-users of the application. The MCSLA Editor provides the developer with all possible SLOs and SQOs, which may partly incorporate default values, aggregated values or overwritten values depending on those resulting from the contracted cloud services. This resulting MCSLA serves as the contract between the developer and the end-users of the application.

The tool ADAPT is the main DECIDE tool that is dependent on the output of the MCSLA Editor. But also, the MCSLA Editor is dependent on the ACSmI as it provides the initial set of SLAs that have been contracted for a multi-cloud deployment scenario. When a re-deployment takes place another round of interactions between the MCSLA Editor and ACSmI is required (see ***Figure 6***).

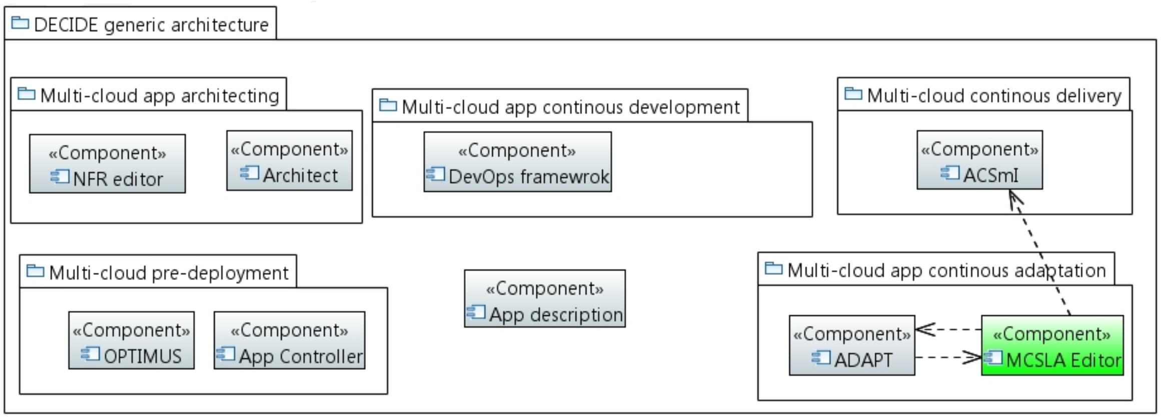

DECIDE generic architecture
Multi-cloud app architecting
«Component» NFR editor
«Component» Architect
Multi-cloud app continous development
«Component» DevOps framewrok
Multi-cloud continous delivery
«Component» ACSmI
Multi-cloud pre-deployment
«Component» OPTIMUS
«Component» App Controller
«Component» App description
Multi-cloud app continous adaptation
«Component» ADAPT
«Component» MCSLA Editor

Figure 6. DECIDE generic architecture

3.1.2 Initialization and used Aggregation Patterns

Each time the MCSLA is loading an Application Description, two possible initial situations can be in place. First, there is no MCSLA at all because the MCSLA is called the first time for this project, or, second, an MCSLA is already provided, due to earlier sessions or any other way.

In the first case, the MCSLA Service is responsible for generating an initial MCSLA which will be calculated based on all defined NFRs and configured microservices. In the second case, the already provided MCSLA needs to be checked to see if it is still aligned with the current available service provider SLAs. In both cases, the first step for the MCSLA Service is to get in contact with the ASCmI service discovery in order to get a list of SLAs of all cloud services the current deployment scenario requires. This is described in detail in chapter 3.2.5

The list of cloud service SLAs relevant for the application will be added to the MCSLA element in the Application Description for further reference. As previously described, this list is neither meant or relevant for the end customer of the application nor can the contained SLAs be edited or changed. They are used for setting up the application level SLA in an aggregated form if applicable, and will be displayed during the MCSLA edit session to support the developer and for informational reasons.

Usually, SLAs from the cloud service providers should be as descriptive as possible, and they should especially contain metrics that allows the customer to monitor the fulfillment. The DECIDE project, and the MCLSA Editor, decided to utilize the ISO/IEC 19086-2 [1] for a common technical interchangeable format. Unfortunately, the cloud service SLAs provided by the ACSmI service discovery are rudimentary without any metric information available, with the exception of availability. Instead, for each service objective derived from the defined NFRs a fixed aggregation method is assigned and used during initialization, as can be seen in **Table *2***:

Table 2. Service Objectives and their assigned Aggregation Patterns

Service objective	Aggregation pattern
Availability	Availability sumType
Performance	maxType
Scalability	Not applicable yet
Location	sumType
Legal Level	maxType
Cost	sumType

The aggregation patterns are fully described in chapter 2.2. Depending of the pattern used, the metrics that are used to measure and monitor each individual microservice, needs to be defined. Whereas most patterns trivially use the given fixed value, at least Availability requires a more complex metric for measuring the microservices.

To set up an MCSLA, for each NFR defined in the *Application Description* that is tagged with “Application”, an SLO is defined and added to the application level SLA. This list of service objectives is presented to the developer just as a recommendation and therefore should be revised, modified or extended by the developer to be applicable to the end customer. It is possible to add more service objectives, remove them or modify already contained one.

Availability

Initially, the MCSLA Service generates a service objective of the term “Availability” with an aggregation pattern “Availability sumType” as described in chapter 2.2.1. The reference value and unit are set from the corresponding NFR definition. The condition for monitoring is set to be greater or equal. For the aggregation, each microservice is assigned a metric for calculating the availability as meantime between failure and meantime to recover (MTBF/MTTR). The calculation formula is:

*100 * MTBF/(MTBF + MTTR)*

Performance

For any given Performance NFR, a service objective with the aggregation pattern sumType is generated. The aggregated value is calculated by just adding all measured performance values of each microservice. There is no special metric to calculate these values, they are just put from the monitoring as is. The condition statement that is monitored is less or equal.

Scalability

Currently there is no aggregation pattern assigned. It requires the definition of an appropriate measurement for each microservice at runtime. This is topic for further evaluation and future extensions.

Location

The service objective derived from a Location NFR is simply a sumType aggregation, where all locations that are part of the cloud service SLAs for each microservice is aggregated into one list. The condition statement for monitoring is equality. The generated service objective is just a very simplified assumption and should be reviewed and modified by the developer if necessary.

Cost

The recommended service objective derived from the Cost NFR is a simple sumType aggregation. The metric for each microservice is the sum of all required could services costs. The condition statement for monitoring is less or equal.

3.2 Technical description

This section describes the technical details of the implemented software for the current prototype of the MCSLA Editor.

3.2.1 Prototype Architecture

The MSCLA Editor is a two-tier architecture represented by the MCSLA frontend in the first place and, secondly, the backend consisting of the MCSLA Service and the MCSLA Core Library. **Figure *7*** depicts the component diagram for the MCSLA Editor. The architecture has not changed signifcantly since the previous deliverable [3].

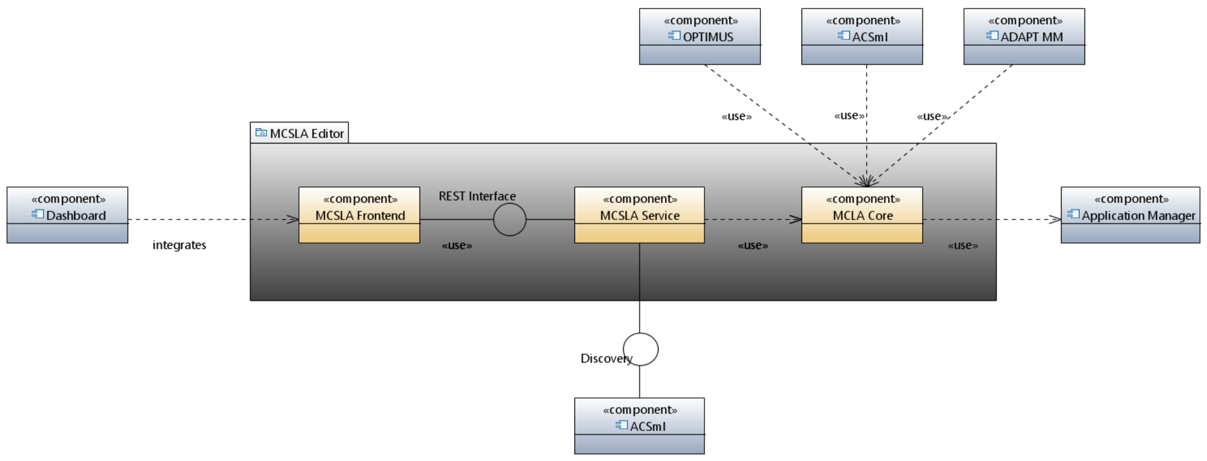


Figure 7. Component diagram for MCSLA editor

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 has been integrated into the DevOps Dashboard. For this, the frontend provides a slim UI where the header bar and its own navigation menu is omitted. The frontend communicates with the backend, and uses defined REST 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/IEC 19086-2 [1] and cover terms that are application specific, rather than just provider specific.

MCSLA Service

The MCSLA Service is in charge of managing the MCSLA and holds its logical information model. It communicates with the code git repository via the Application Controller in order to access the *Application Description,* and receive the ids of the cloud services where the multi-cloud application is deployed on.

The MCSLA Service uses this information from the *Application Description* to access cloud services 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 Service 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.

MCSLA Core Library

The MCSLA Core Library serves the MCSLA Service with the SLAs in order to accumulate and aggregate the possible values for Service Objectives 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.

The following sequence diagram (**Figure 8**) depicts the communication and message exchange that takes place between the MCSLA Editor components, external repositories and DECIDE tools (ACSmI).

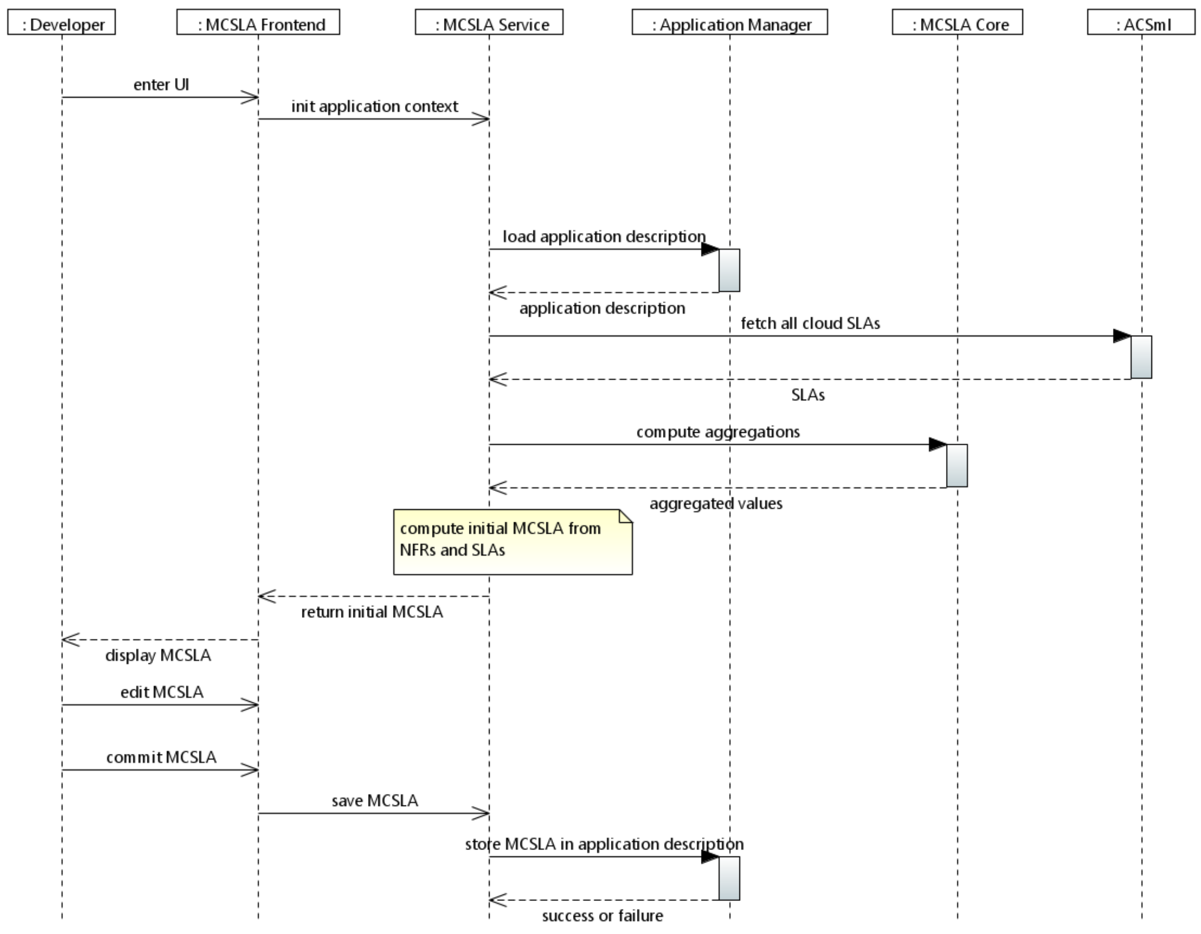


Figure 8. Sequence diagram for creating an MCSLA

The sequence for creating an MCSLA is as follows:

1. The developer starts the MCSLA Frontend (GUI); this process calls the MCSLA Service in order to populate the frontend with the necessary values.
2. As long as the MCSLA Editor as a whole is integrated into the dashboard, it is clear which Application Description is applicable at this stage. The Application Description, residing in a repository, will be accessed via the Application Controller, to retrieve the currently used deployment topology, i.e. the cloud service Ids.
3. With the cloud service Ids, the MCSLA Service contacts ACSmI in order to obtain the contracted SLAs.
4. The MCSLA Service then utilizes the MCSLA Core Library to take the necessary measures to aggregate the SLOs defined in each SLA.
5. Once this step is completed, the MCSLA Service populates the frontend with the available SLO/SQOs and their possible values.

In summary, the developer uses the GUI to create the MCSLA, which is in turn saved by the MCSLA Service in the code repository as well as registered in the *Application Description.*

3.2.2 MCSLA Data Model

The data model for an MCSLA is depicted below in ***Figure 9*** and serves as a reference.

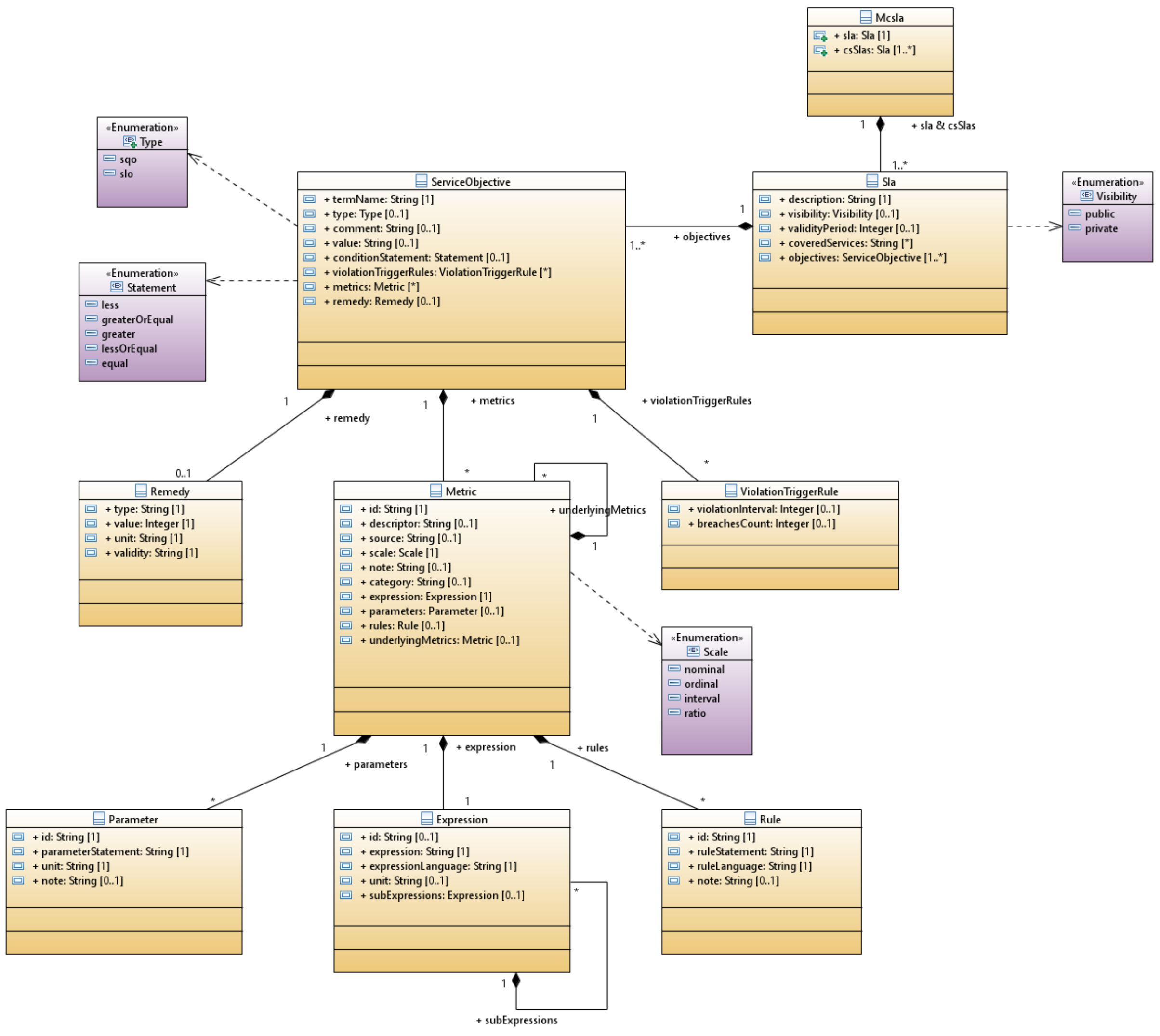


Figure 9. MCSLA Data Model

The following tables describe the MCSLA model for monitoring with a brief description for each field based on the standard ISO19086-2 [1]. ***Table 3*** describes the nested elements for the MCSLA. The MCSLA editor is responsible for eliciting this information from the user.

Table 3. Application description model for monitoring the application via its MCSLA

Element Name	**Mcsla**		
Description	The aggregated SLAs as MCSLA		
Property	**Type**	**Cardinality**	**Definition**
sla	Sla	1..1	The SLA towards the end-customer of the application
csSlas	Array of Sla	1..n	The cloud service SLAs as provided by the cloud service providers

The following ***Table 4*** describes the properties of element type Sla.

Table 4. Properties of element type Sla

Element Name	**Sla**

Description	General information about an SLA		
Property	**Type**	**Cardinality**	**Definition**
description	String	1..1	A description about the context of the SLA
visibility	String	0..1	public or private
validityPeriod	Integer	0..1	The validity period of the SLA in days
coveredServices	Array of String	0..n	Named services that the SLA covers
objectives	Array of ServiceObjective	1..n	The list of service objectives composing the SLA

The following ***Table 5*** describes the properties of element type *ServiceObjective*.

Table 5. Properties of element type ServiceObjective

Element Name	**ServiceObjective**		
Description	General information about service objectives		
Property	**Type**	**Cardinality**	**Definition**
termName	String	1..1	Name of the term to which it refers to
type	Enumeration	0..1	- slo - sqo
comment	String	0..1	A short textual comment about the service objective.
value	String	0..1	Term value that should not be violated based on calculation formula
unit	String	0..1	The unit of the value property
conditionStatement	Enumeration	0..1	- greater - less - greaterOrEqual - lessOrEqual - equal
violationTriggerRules	Array of ViolationTriggerRule	0..n	The violation trigger rules
metrics	Array of Metric	0..n	The definition of how to measure the SLO or SQO
remedy	Remedy	0..1	The compensation available to the cloud service customer in the event the cloud service provider fails to comply a specified cloud service level objective

The following Table 6 describes the properties of element type *ViolationTriggerRule*.

Table 6. Properties of element type ViolationTriggerRule

Element Name	**ViolationTriggerRule**
Description	General information about a violation trigger rule

Property	Type	Cardinality	Definition
violationInterval	Number	0..1	Indicates the monitoring frequency for the service objective
breachesCount	Number	0..1	The count of how many breaches have happened

The following Table 7 describes the properties of element type *Remedy*.

Table 7. Properties of element type Remedy

Element Name	**Remedy**		
Description	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 objective		
Property	**Type**	**Cardinality**	**Definition**
type	String	1..1	The type of remedy the cloud service provider will be offering the cloud service customer
value	Number	1..1	The value of the type of remedy offered by the cloud service provider
unit	String	1..1	The unit for the value offered
validity	String	1..1	The validity period for this remedy

The following ***Table 8*** describes the properties (taken directly from ISO/IEC 19086-2 Metric Model [1]) of element type *Metric*. The MCSLA Editor is responsible for eliciting this information from the user.

Table 8. Properties of element type Metric

Element Name	**Metric**		
Description	General information about the metric		
Property	**Type**	**Cardinality**	**Definition**
id	String	1..1	A unique identifier for the metric within a context
descriptor	String	0..1	A short description of the metric
source	String	0..1	The individual or organization who created the metric
scale	Enumeration	1..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	1..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 underlying metrics, parameters, and rules.
parameters	Array of Parameter	0..n	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.
rules	Array of Rule	0..n	A rule is used to constrain a metric and indicate possible method(s) for measurement.
underlyingMetrics	Array of Metric	0..n	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 9 describes the properties for the *Expression* element type.

Table 9. Properties of the Expression element type

Element Name	**Expression**		
Description	The expression of the calculation of the metric and supporting information		
Property	**Type**	**Cardinality**	**Definition**
id	String	0..1	A unique identifier (within the context of the metric) for the expression
expression	String	1..1	The expression statement written using the ids to represent underlying metrics, parameters, and rules.
expressionLanguage	String	1..1	The language used to express the metric (i.e. ISO80000 [9])
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.

subExpressions	Array of Expression	0..n	Associated elements of type Expression 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 10 describes the properties of the *Parameter* element type.

Table 10. Properties of the Parameter element type

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.		
Property	**Type**	**Cardinality**	**Definition**
id	String	1..1	The unique identifier of the parameter
parameterStatement	String	1..1	The statement or value of the parameter
unit	String	1..1	The unit of the parameter
note	String	0..1	Additional information about the parameter

The following Table 11 describes the properties of the *Rule* element type.

Table 11. Properties of the Rule element type

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 [9], SBVR [12])		
Property	**Type**	**Cardinality**	**Definition**
id	String	1..1	The unique identifier for the rule
ruleStatement	String	1..1	A constraint on the metric
ruleLanguage	String	1..1	The language used to express the rule in the ruleStatement property
Note	String	0..1	Additional information about the rule

3.2.3 Predefined Expressions

In order to avoid implementing a complete ISO80000 [9] parser and interpreter, the MCSLA Editor defines a set of predefined expressions that can be referenced from within the metrics description of an applications MCSLA. They are neither part of the ISO 19086-2 [1] specification nor any other standard. They are only used within the DECIDE project context.

EMPTY – An empty expression that does nothing and returns always a null evaluation result. It needs to be checked if this can be aligned with a neutral aggregation pattern.

AVAILABILITY_UPTIME_BC – An expression that calculates the uptime (availability) during a specific billing cycle. The corresponding formula is *100 - (billingCycle - totalDowntime)/billingCycle*. Whereas the total downtime is the sum of all downtimes during the billing cycle. Usually this is measured in seconds or milliseconds where the service is not reachable.

AVAILABILITY_MTBFMTTR – An expression that calculates the availability based on the meantime between failure and meantime to recover values. The formula is *100 * MTBF/(MTBF + MTTR)*.

AVAILABILITY_AGGREGATION_SUMTYPE – An expression that calculates the special sumtype aggregation pattern for availability as described in chapter 2.2.1.

AGGREGATION_MINTYPE – An expression that calculates the mintype aggregation pattern as described in chapter 2.2.

AGGREGATION_MAXTYPE – An expression that calculates the maxtype aggregation pattern as described in chapter 2.2.

AGGREGATION_SUMTYPE – An expression that calculates the standard sumtype aggregation pattern as described in chapter 2.2.

To install a predefine expression in a metric as part of a service objective you need to set the expressionLanguage property of the Expression element to “predefined”. The value of the property “expression” identifies the predefined expression that should be used in the metric:

```
{
        "id": "CSA_AV_001",
        "descriptor": "Aggregation of microservices",
        "scale": "ratio",
        "expression": {
                "expression": "AVAILABILITY_AGGREGATION_SUMTYPE",
                "expressionLanguage": "predefined",
                "unit": "percentage"
        }
}
```

The list of predefined expressions is currently limited mostly to the prioritised patterns that are required for the addressed scenarios *Availability* and *Performance*.

3.2.4 REST Interface

The backend provides the following operations described below in brief. Each operation produces and consumes JSON. The interface documentation will be generated using the OpenAPI specification version 3 [10] and is available online.

Table 12. REST interface provided by the MCSLA Service

HTTP Verb	URL	Description
GET	/applications/init?{query params}	Initialize an application context for further processing. It returns the applications mcsla context to the client.

GET	/applications/{name}	Get the mcsla context of the application with the name {name}.
GET	/applications/{name}/reset	Resets any modifications and revert to the head of the remote project repository.
POST	/applications/{name}/mcsla	Update the mcsla element for the application with the name {name}.
GET	/applications/{name}/mcsla/{termName}/aggregate	Aggregate a specific service objective identified through the term name.

The MCSLA Service provides also a Web page displaying the documentation of this OpenAPI 3 [10] based REST API using the ReDoc [11] framework. The page can be reached on the root context ("/") (see chapter 4.1.2.3).

3.2.5 ACSmI Interface Usage

The MCSLA Service uses the following REST interface to receive a list of SLAs for a list of cloud services. The list is usually fetched when an MCSLA edit session starts, either to initially generate an MCSLA or to update an existing one if changes needs to be applied. **Table *13*** describes the only method that is used by the MCSLA Service.

Table 13. REST Interface used from ACSmI discovery service

HTTP Verb	URL	Description
GET	/acsmiservices/api/services/nfr?serviceids=id1,id2,...	Get a list of service related objectives describing a cloud services SLA as offered by the provider.

The ACSmI service discovery returns a JSON array of objects describing the available SLOs per service id. Each object in the array contains the following information:

- *serviceid* - The related service id.
- *servicename* - The related service name.
- *serviceclassname* – A classification of the service
- *nfrname* - The service objective term.
- *nfrmetric* - A named metric.
- *nfrvalue* - A concrete value.

Although, some properties contain "nfr" in their names, they are not treated as requirement, because they describe an agreement.

For this prototype the following *nfrnames* are possible, representing service objectives:

- Region – A geographical region (location)
- Zone – A geographical zone (location)
- Provider – The name of the provider
- Availability – Contains percentage availability
- Performance – The offered performance of the service in response time (milliseconds)
- Scalability – The scalability of the offered service
- Legal Level – The offered legal level, 1, 2 or 3

- Cost/Currency – The cost of the service provided

If there are more values available, e.g. different zones or regions, each value is returned as a separate object. An example JSON element looks like this:

```
{
  "serviceid": 1,
  "servicename": "Database Service 0001",
  "serviceclassname": "Database",
  "nfrname": "Availability",
  "nfrmetric": "Availability",
  "nfrvalue": "99.4"
}
```

This format is temporarily used and should be replaced by the ISO 19086-2 [1] standard which is adapted by the DECIDE project as a common technical description format of SLAs and metrics, in order to be interchangeable and comparable between different cloud service providers. As long as this is not the case, the MCSLA service assumes a fixed set of metrics, one for each service objective that can be returned (see chapter 3.1.2). If additional information, like human readable explanations or annotations, are added, the MCSLA editor needs to be extended and should then be able to display them to the developer.

3.2.6 PDF Generation

The MCSLA frontend is able to generate a PDF document from the Application Multi-Cloud SLA. It will not contain any part or information about the contracted cloud services SLAs of the underlying cloud services.

The implementation of the mcsla-ui component utilizes a JavaScript library *pdfmake* [13] that allows the generation of a PDF document by defining a JSON structure. A basic template is defined to reflect the Application MCSLA as a list of service objectives:

```
var docDefinition = {
    pageSize: 'A4',
    pageMargins: [85, 60, 85, 60],
    fontSize: 11,
    styles: {
        title: {
            fontSize: 22,
            bold: true,
            lineHeight: 2
        },
        subtitle: {
            fontSize: 20,
            lineHeight: 2
        },
        header: {
            fontSize: 16,
            italics: true
        },
        header2: {
            fontSize: 14,
            margin: [0, 0, 0, 10]
        },
        header3: {
            fontSize: 12,
            margin: [0, 0, 0, 10]
        }
    },
    header: {},
    content: "Placeholder",
    footer: {}
};
```

When the export order is received, the frontend transforms the underlying MCSLA in a format that fits into the format that the *pdfmake* library understands, and extends the template accordingly. The library then generates the PDF document from this JSON object and initiates the download process.

4 Delivery and usage

All parts that are necessary to build and run the MCSLA Editor are available either in a zip file or can be cloned from the DECIDE public repository (https://git.code.tecnalia.com/DECIDE_Public):

mcsla-ui (MCSLA Frontend)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA/mcsla-ui.git

mcsla-service (MCSLA Service)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA/mcsla-service.git

mcsla-core (MCSLA Core Library)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA/mcsla-core.git

4.1 Configuration and Installation instructions

In order to build, configure and run the MCSLA Editor, you need to do different steps for the three different components that build up the editor. Whereas the frontend is written in JavaScript and runs completely in the browser as a single page application, the backend is a standalone Java program including a web server. In addition, the backend relies on the shared library for metric calculations. Therefore, you should at first build and install the mcsla-core library, then build and run the mcsla-service application, and finally build and install the mcsla-ui single page web application (SPA).

4.1.1 MCSLA Core (Shared Library)

Before the library can be build the dependency to the *Application Controller* needs to be installed in an accessible maven repository, usually in the local repository of the developer. A detailed installation guide is available in the deliverable D3.11 [14]

4.1.1.1 Build the library

The project is available via its git repository. If you have access, do the following steps:

```
$ git clone https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components.git
$ cd mcsla
$ cd mcsla-core
```

The project uses Maven as build tool. To build use the following command:

```
$ mvn clean package
```

After the build succeeded the package jar and a packaged fat jar can be find in the target directory.

4.1.1.2 Install the library

For non-Maven based projects, you can take the build jar file located in the target directory after executing the build command and put it in the classpath of your application. There is also a fat jar provided containing all dependencies if required.

For Maven based projects you need to install it in a maven repository which your application can access. E.g. to put it in your local maven repository, the user can simply call

```
$ mvn install
```

Finally, add the dependency to the applications pom.xml file. The correct version needs to be set:

```
<dependency>
    <groupId>eu.DECIDEh2020</groupId>
    <artifactId>mcsla-core</artifactId>
    <version>0.0.3</version>
</dependency>
```

For further usage details and interface description see chapter 4.2.2

4.1.2 MCSLA Service (Backend)

The MCSLA Service is implemented based on Vert.x [15], a tool-kit for building reactive application for the Java virtual machine.

4.1.2.1 Build the service

The project is available via its git repository. If you have access do the following steps:

```
$ git clone https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components.git
$ cd mcsla
$ cd mcsla-service
```

The project uses Maven as build tool. To build use the following command:

```
$ mvn clean package
```

You will find the built artefacts in the *target* directory.

4.1.2.2 Service configuration

The following environment variables are read:

- DECIDE_ACSMI_DISCOVERY_SERVICE_URI (mandatory) - The address of the ACSmI discovery service
- DECIDE_REPOS_BASE_DIR (optional) - The path to the base for cloning remote repositories

4.1.2.3 Run the service

No installation required. To run the service, execute the following command:

```
$ java -jar target/mcsla-service
```

4.1.2.4 Docker

The MCSLA Service can be run also in a docker container [16]. A *Dockerfile* is provided in order to build a docker image. To build and run the docker image execute the following commands from within the project folder:

```
$ docker build -t decide/mcsla-service
$ docker run -it -p 8080:8080 decide/mcsla-service
```

4.1.3 MCSLA UI (Frontend)

The MCSLA Editor user interface is implemented as single page application using the React JavaScript framework [17] initially developed by Facebook.

4.1.3.1 MCSLA UI configuration

The project is available via its git repository. If you have access do the following steps:

```
$ git clone https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components.git
$ cd mcsla
$ cd mcsla-ui
```

Before the frontend can use the REST web services of the MCSLA Service backend you have to first configure the application. It needs to be done before the application is actually build. Edit the file *mcsla-ui/public/app-conf.json* and put there the correct URL to the MCSLA Service API endpoint.

```
"serviceUri": "http://localhost:8080"
```

4.1.3.2 Build the UI

The project uses npm, the package manager from node.js, for building. The easiest way is to install node.js [18]. npm is an integrated part of the installation. After npm is installed simply call:

```
$ npm install
$ npm run build
```

Based on the package.json file, `npm install` will resolve all the required dependencies that the application needs, in order to build the mcsla-ui application. After `npm run build` the “compiled” application can be find inside the “build” directory.

4.1.3.3 Install and run

In order to run the frontend, it needs to be served from an http endpoint. Any http server can do, or alternatively the *serve* package can be used after installed via npm:

```
$ npm install -g serve
```

Just call `serve -s build` to run an http endpoint and to serve the single page application. Enter localhost:5000 in a browser and the following web app should be seen:

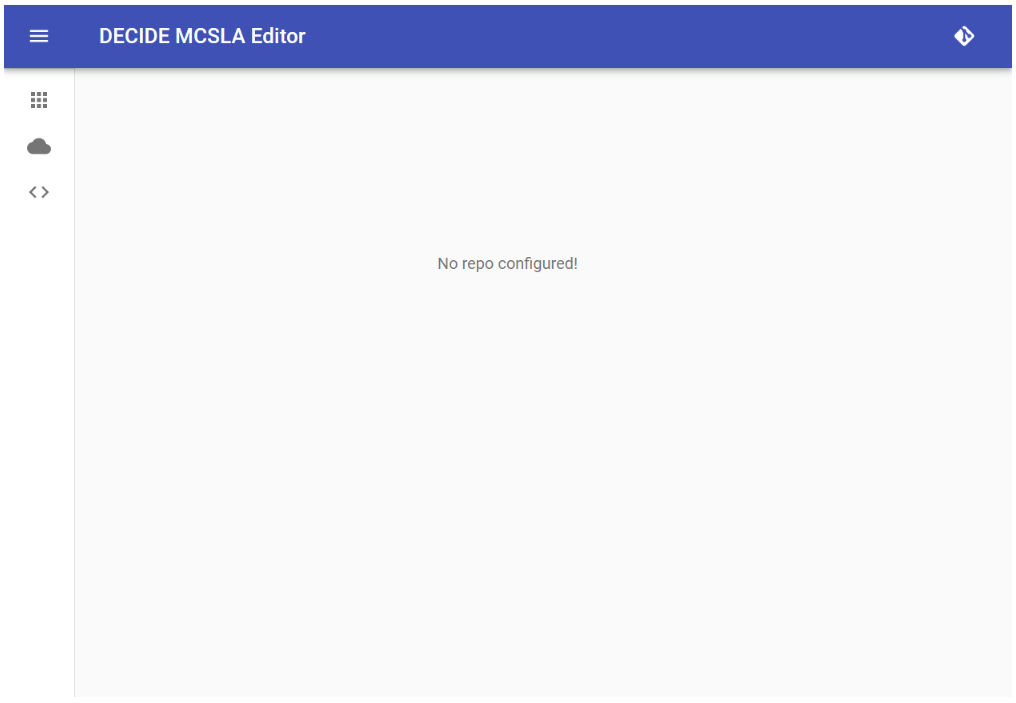


Figure 10. The initial MCSLA Editor page

To integrate the frontend into another web pages, e.g. in an HTTP iframe tag, there is a reduced version without the title bar and without the left-hand navigation menu. It can be requested via the URL *http://localhost:5000/iframe*.

For further usage details see chapter 4.2.2

4.1.3.4 Docker

To build a docker image [16] a *Dockerfile* is provided. To build and run the image the following commands needs to be executed from inside the project folder:

```
$ docker build -t decide/mcsla-ui .
$ docker run -i -p 8080:8080 -e "DECIDE_MCSLA_SERVICE_URI=<your_mcsla_service_uri>" decide/mcsla-ui
```

4.2 User Manual

4.2.1 MCSLA Core Library usage

The MCSLA Core library encapsulates the model and calculation of metrics. Additionally, it defines and implements predefined expressions for convenient usage of specific metrics especially in the context of the DECIDE project.

4.2.1.1 Getting started

This is a small example how to create a metrics context and evaluate a cloud service objective:

```
// create context
MetricsContext context = MetricsContext.create(appDescription);

// get sla service objective metrics for a apecific cloud service
Map<String, ServiceObjectiveMetrics> metrics = context.getCsMetrics(csId);

// get the service objective metrics for a specific term name, e.g. "Availability"
ServiceObjeciveMetrics availabilityMetrics = metrics.get("Availability");

// now evaluate against a concrete monitoring result
EvaluationResult<Double> evaluationResult = availabilityMetrics.evaluate(monitoringResult);

if (!evaluationResult.isError()) {
    log.info("condition met: {}, result: {}", evaluationResult.isConditionMet(),
evaluationResult.getMeasurementResultValue());
} else {
    log.error(evaluationResult.getErrorMessage());
}
```

Further examples can be found in the test classes located in the *src/test/java* directory.

4.2.1.2 The Interface

In general, there are three main classes:

- *MetricsContext*
- *ServiceObjectiveMetrics*
- *EvaluationResult<T>*

MetricsContext

This class provides static methods for creating context objects. Created context can be used to get service objective metrics of each SLA as a map. Furthermore, it contains a memory for cloud service evaluation results, so specific aggregation expressions can be implicitly applied. And finally, it allows the aggregation of service objectives or raw values.

Aggregation

The easiest way to use the *MetricsContext* is to aggregate some raw values:

```
List<Double> values = Arrays.asList(99.9, 99.8, 99.95);
Double value = MetricsContext.aggregate(values, Predefined.AVAILABILITY_AGGREGATION_SUMTYPE);
```

See *Predefined Aggregation Expressions* below for available aggregation expressions.

To start working with the library in the context of an application description you need to create a *MetricsContext* object:

```
AppDescription appDescription = ...
MetricsContext context = MetricsContext.create(appDescription);
```

Now all agreement values of all cloud services for a specific term name can be aggregated:

```
Double value = MetricsContext.aggregate(Predefined.AVAILABILITY_AGGREGATION_SUMTYPE, "Availability");
```

Note, the return type depends on the predefined aggregation expression to be used.

Evaluation

The *MetricsContext* object has two methods for retrieving a *ServiceObjectiveMetrics* object which is required for evaluation:

```
Map<String, ServiceObjectiveMetrics> appMetrics = context.getAppMetrics();
Map<String, ServiceObjectiveMetrics> cloudserviceMetrics = context.getCsMetrics(csId);
```

The return value is a map of term names and corresponding metrics of either a specific cloud service or the application.

Now you can get the *ServiceObjectiveMetrics* object for a specific term name:

```
ServiceObjectiveMetrics objectiveMetrics = appMetrics.get("Availability");
```

ServiceObjectiveMetrics

You need objects s of this class to evaluate the metrics. The class has the following evaluation functions:

- *evaluate()*
- *evaluate(double[] measuredValues)*
- *evaluate(int[] measuredValues)*
- *evaluate(List\<Integer\> measuredValues)*
- *evaluate(Map\<String, List\<Long\>\> measuredValues)*
- *evaluate(double measuredValue)*

They all return an *EvaluationResult<T>* object.

The first method without passing any monitoring results utilizes the memory function of the metrics context. It therefore requires that each monitoring result to be aggregate must be evaluated beforehand.

This makes sense where you have to evaluate first each cloud service metrics and then the aggregated application metric:

```
Map<String, ServiceObjectiveMetrics> cloudserviceOneMetrics = context.getCsMetrics("<csId>");
ServiceObjectiveMetrics oneMetrics = cloudserviceOneMetrics.get("Availability");
EvaluationResult<Double> resultOne = OneMetrics.evaluate(99.9);
...
Map<String, ServiceObjectiveMetrics> cloudserviceTwoMetrics = context.getCsMetrics("1");
ServiceObjectiveMetrics twoMetrics = cloudserviceTwoMetrics.get("Availability");
EvaluationResult<Double> resultTwo = twoMetrics.evaluate(99.9);
```

```
...
Map<String, ServiceObjectiveMetrics> appMetrics = context.getAppMetrics();
ServiceObjectiveMetrics availabilityMetrics = appMetrics.get("Availability");
EvaluationResult<Double> resultApp = availabilityMetrics.evaluate();
```

EvaluationResult<T>

The first thing you should do is to check if the evaluation was successful:

```
EvaluationResult<Doube> result = ...
if (result.isError()) {
    log.error(result.getErrorMesssge());
} else {
    ...
}
```

A successful evaluation result contains the following information:

- The evaluation date and time
- The underlying term name
- The measurement result value
- The used condition statement
- And finally, if the condition is met (see *Statement*)

```
EvaluationResult<T> result = ...
Date time = result.getMeasurementTime();
String termName = result.getTermName();
T measurementValue = result.getMeasurementResultValue();
Statement conditionStatement ? result.getConditionStatement();
assert result.isConditionMet() : "Condition is not met.";
```

Predefined Aggregation Expressions

The class *Predefined* defines the following expressions that can be referenced by an enumeration value.

Table 14. Enumeration values for predefined expressions

Expression	
EMPTY	The empty aggregation returns always null
AVAILABILITY_UPTIME_BC	Evaluation of downtimes in seconds to a percentage uptime during a billing cycle
AVAILABILITY_MTBFMTTR	Evaluation based on meantime between failure and meantime between recovery
AVAILABILITY_AGGREGATION_SUMTYPE	An aggregation that summarizes percentage values
AGGREGATION_MINTYPE	Returns the min value
AGGREGATION_MAXTYPE	Returns the max value
AGGREGATION_SUMTYPE	Returns the sum up of values

Condition Statements

Table 15 shows the possible condition statement values which are used to compare the agreed service objective value and the actual measured and calculated monitoring result.

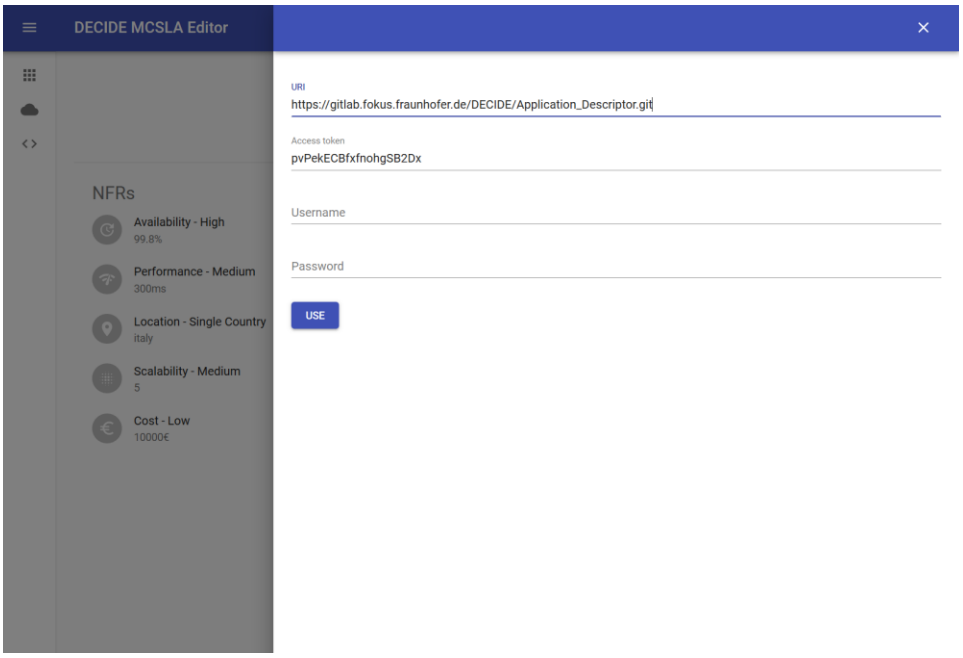
Table 15. Enumeration values for condition statements

Expression	
greater	If the monitored value is greater, then the condition is complied
less	If the monitored value is less, then the condition is complied.

greaterOrEqual	If the monitored value is greater or equal, then the condition is complied
lessOrEqual	If the monitored value is less or equal, then the condition is complied
equal	If the monitored value is equal, then the condition is complied

4.2.2 MCSLA UI

When the frontend is called without parameterizing the DECIDE project that should be used, the application will display the page as shown in ***Figure 10***. To specify the DECIDE project to use, the corresponding git repository needs to be provided. Open the git repository dialog by clicking on the git logo button on the upper right. The dialog opens from the right as shown in ***Figure 11***.



URI
https://gitlab.fokus.fraunhofer.de/DECIDE/Application_Descriptor.git

Access token
pvPekECBfxfnohgSB2Dx

Username

Password

Figure 11. Git repository Dialog

It asks for the usual values like the repository URL and some access information. For authentication there are two possibilities. Either provide an access token which can usually be defined via the git repository provider, or username and password credentials.

For bookmarking a shortcut way is possible in providing this information via query parameters of the URL:

Table 16. Query parameters for the frontend

Query Parameter	Description
uri	The repository URL
token	The access token
username	The credentials username
password	The credentials password

Here is an example:

?uri=https://gitlab.com/MyGroup/myproject.git&token=pvQekECBfxfn1hgSC2Dx

After the frontend is configured to use a specific DECIDE project is displays some general information about the application.

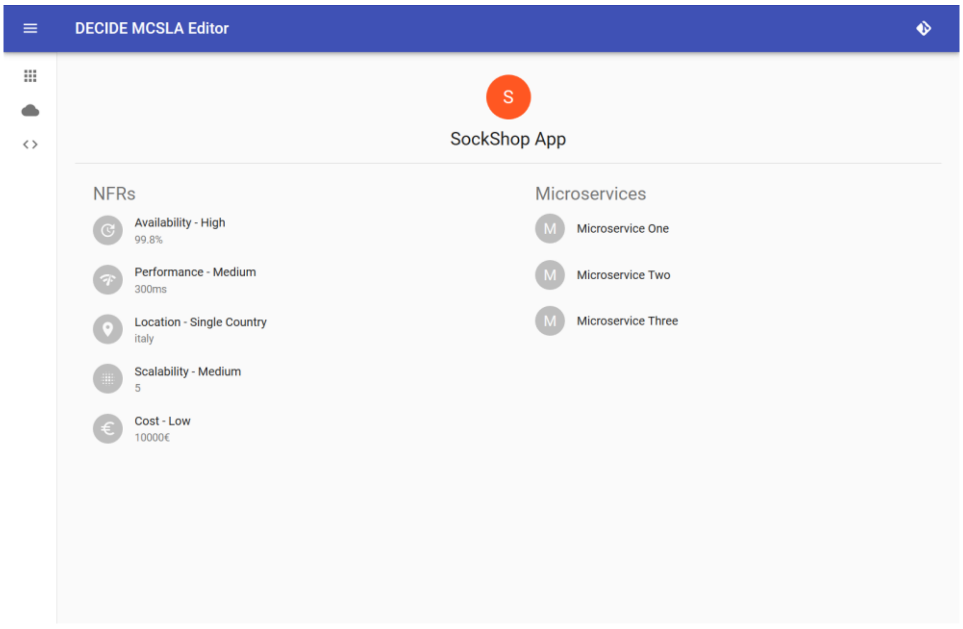
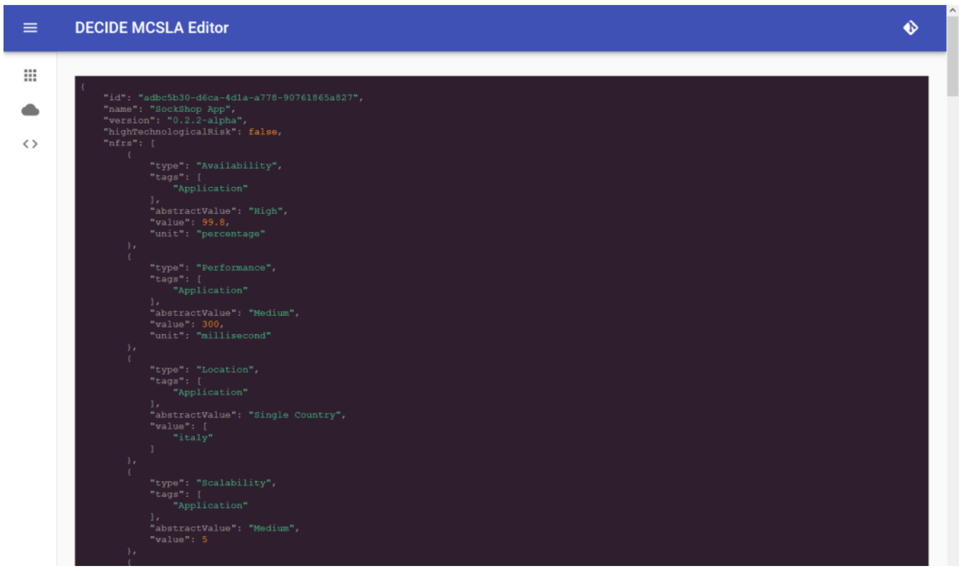


Figure 12. General information page

In the content area the title and some general information is displayed: the title of the DECIDE project currently selected, a list of NFRs defined in the project and a list of microservices. The left-handed navigation menu shows the following options:

The general information about the application derived from the app description.

{…} The complete application description displayed in raw JSON format with some syntax highlighting (***Figure 13***).

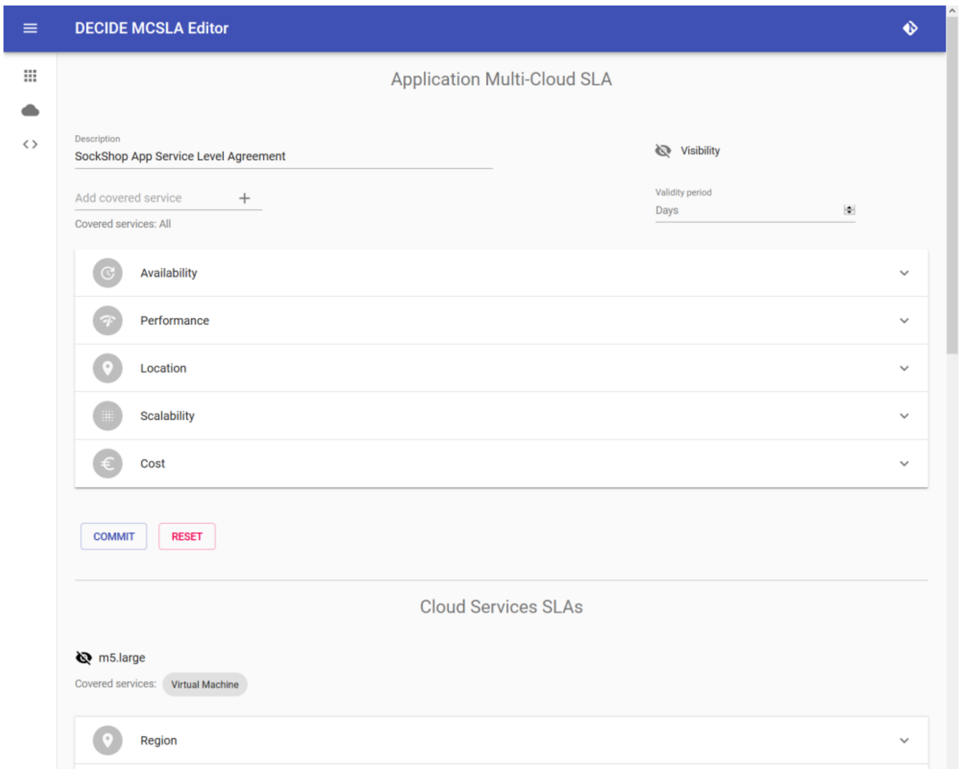


```
{
    "id": "adbc5b30-d6ca-4d1a-a778-90761865a827",
    "name": "SockShop App",
    "version": "0.2.2-alpha",
    "highTechnologicalRisk": false,
    "nfrs": [
        {
            "type": "Availability",
            "tags": [
                "Application"
            ],
            "abstractValue": "High",
            "value": 99.8,
            "unit": "percentage"
        },
        {
            "type": "Performance",
            "tags": [
                "Application"
            ],
            "abstractValue": "Medium",
            "value": 300,
            "unit": "millisecond"
        },
        {
            "type": "Location",
            "tags": [
                "Application"
            ],
            "abstractValue": "Single Country",
            "value": [
                "italy"
            ]
        },
        {
            "type": "Scalability",
            "tags": [
                "Application"
            ],
            "abstractValue": "Medium",
            "value": 5
        },
        {
```

Figure 13. The JSON view page

The actual SLA editor

Figure 14 shows the actual MCSLA Editor page where the user can edit the SLA for the end-customer (MCSLA) and view the contracted SLAs for the cloud services of the current deployment scenario. These parts are separated in the two sections: *Application Multi-Cloud SLA* and *Cloud Services SLAs*



Description: SockShop App Service Level Agreement

Add covered service

Covered services: All

Validity period: Days

m5.large

Covered services: Virtual Machine

Figure 14. The SLA editor page

On top of the first section the user can edit and change general information of the SLA for the end customer, like validity period, a description and a list of covered services this SLA is defined for (***Figure 14***). Below this general information is a list of Service Objectives in a folded list format.

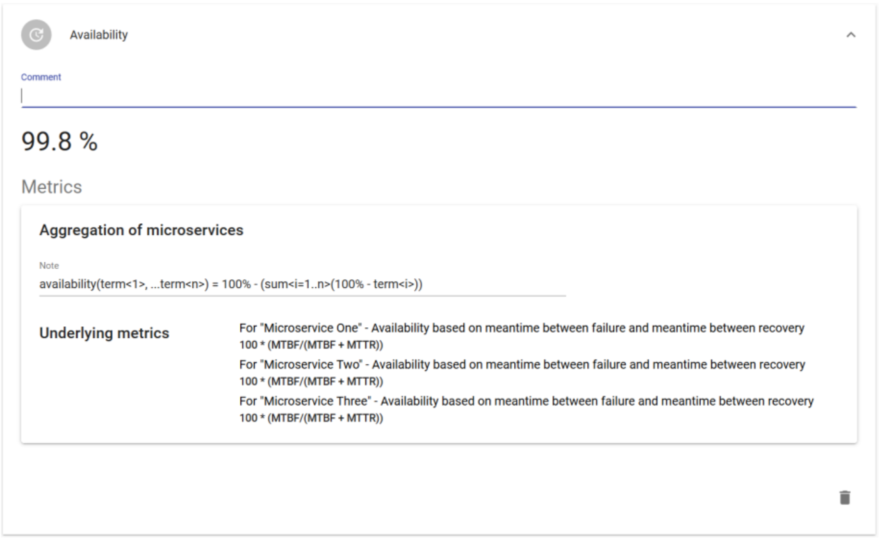
4.2.2.1 Add Service Objectives

In the bottom right corner there is always a button that allows the user to add Service Objectives that are currently not in the list. These are:

- Cost
- Location
- Scalability
- Performance
- Availability
- Add new Service Objective

4.2.2.2 Edit Service Objectives

The Service Objective must be expanded for editing purpose via the chevron on the right side of the list entry. ***Figure 15*** shows an expanded Service Objective for editing.


Availability

Comment

99.8 %

Metrics

Aggregation of microservices

Note
availability(term<1>, ...term<n>) = 100% - (sum<i=1..n>(100% - term<i>))

Underlying metrics	For "Microservice One" - Availability based on meantime between failure and meantime between recovery 100 * (MTBF/(MTBF + MTTR)) For "Microservice Two" - Availability based on meantime between failure and meantime between recovery 100 * (MTBF/(MTBF + MTTR)) For "Microservice Three" - Availability based on meantime between failure and meantime between recovery 100 * (MTBF/(MTBF + MTTR))

Figure 15. Expanded Service Objective

The editing features in this release are still not complete and needs to be enhanced when more features like a fully integrated ISO/IEC 19086 parser and interpreter is implemented as part of future work. E.g. there is currently no possibility to define metrics and all included concepts like expressions, parameters, rules etc. for a service objective.

4.2.2.3 Deleting Service Objectives

In the right bottom corner of the expanded Service Objective is the trash icon for deleting the Service Objective.

4.2.2.4 Commit or reset the SLA

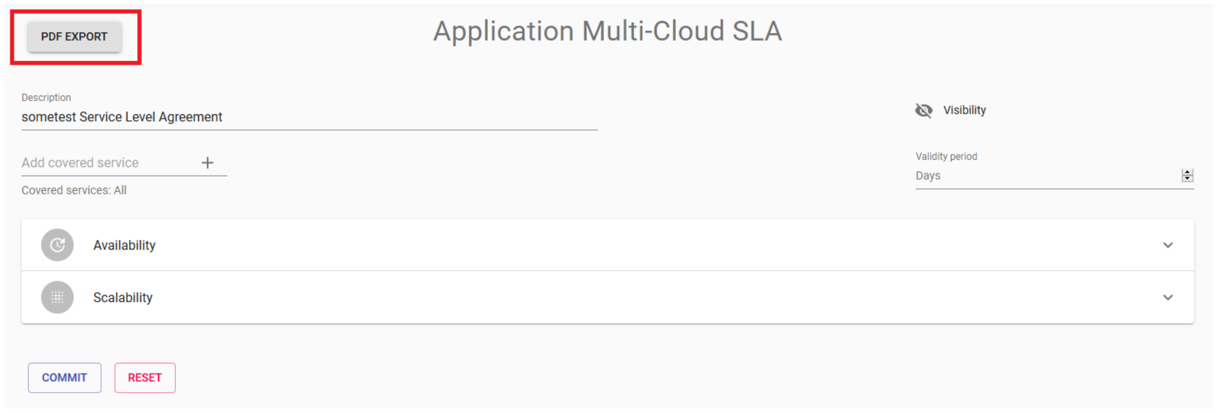
After editing the MCSLA's general information and the list of service objectives they need to be persisted and synchronized with any remote repositories. Press the "Commit" button located on the bottom of the *Application Multi-Cloud SLA* section. To revert any changes after the last commit and switch back to the original state press the "Reset" button nearby (see ***Figure 14***).

4.2.2.5 Cloud Services SLAs

The SLAs from the section Cloud Services SLAs are read-only and visualized for informative reasons. They are separated by each cloud service. Each cloud service paragraph contains a folded list of Service Objectives. More details can be viewed when expanded via the chevron on the right side.

4.2.2.6 Export MCSLA as PDF document

The frontend always contains a button in the top left corner, left hand to the headline, that allows to export and download the MCSLA in a PDF document. The document is generated on the fly as a current snapshot of the actual definition within the *Application Description*. If you click the button, the usual download process of your browser starts. Depending of your browser configuration it will either be downloaded immediately and stored somewhere, opened automatically and being displayed (either inside the browser or using an assigned application), or open a modal save dialog.



4.3 Licensing information

The source code is licensed under the Eclipse Public License version 2.0.

See https://www.eclipse.org/org/documents/epl-2.0/EPL-2.0.html

4.4 Download

The complete source code can be downloaded as a zip file from https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA. In addition, the source projects can be cloned or forked directly from the repositories. The three repositories are:

mcsla-ui (MCSLA Frontend)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA/mcsla-ui***mcsla-service*** (MCSLA Service)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA/mcsla-service***mcsla-core*** (MCSLA Core Library)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/MCSLA/mcsla-core
And if not already made available through the other DECIDE tools, the application controller library is required as well (see deliverable [14]):

app-controller (Application Controller Library)
https://git.code.tecnalia.com/DECIDE_Public/DECIDE_Components/tree/master/AppController

5 Conclusions

This document presented the MCSLA task and the outcome of several discussions and research. The outcome has been presented in Section 2 of the document. The main points relevant for DEDICE MCSLA definition are, among others:

- A MCSLA involves different SLOs and SQOs that can be declared as soft or hard and that maintain an unchangeable external and changeable internal structure. The former must be respected during a re-adaption and re-deployment of the application.
- In multi-cloud deployment scenarios SLAs must be aggregated, removing the complexity of managing a multitude of SLAs from different CSPs
- Aggregation patterns are required.

A selection of aggregation patterns has been presented along with the proposition of a custom aggregation pattern that fulfils our needs in terms of aggregating e.g. the availability of an application dispersed across several CSPs or cloud services of different CSPs. An important aspect of this pattern is that it considers that the dispersed microservice will most probably not fail simultaneously, resulting in a lower availability value than that of an individual microservice.

Furthermore, the functional and technical description of the prototype is detailed. The prototype consists of two main blocks, namely, the frontend and the backend. These components communicate with one another using a restful interface and have been designed to be easily integrated into the DevOps Framework.

The Data Model for the MCSLA has also been presented, it is based on the ISO/IEC 19086 [4, 1, 6, 7] and includes a metric definition for each SLO in order to enable monitoring.

Finally, all information related to building, installing and using the prototype has been described in section 4 of this document.

The development of this document has passed three stages. This document represents the final one and in addition to the intermediate document version, this final document extends the previous version with the description of how the MCSLA Editor generates an initial MCSLA out of NFRs and microservices containing aggregated service objectives. The necessary communication and information exchange with the ACSmI service discovery to retrieve SLAs from cloud services is also explained. And finally, the requirement of providing a human readable form of the edited Application wide Multi-Cloud Service Level Agreement is implemented by the ability to export the MCSLA as a PDF document via the frontend.

5.1 Future work

Some features that are implemented as prototypical demonstrator, can be extended and should be addresses for further enhancements. The following is an excerpt of the open issues:

- Improvements to the UI
- Improve the generation of a more complete and more well-designed PDF document that can be hand out to end customers.
- Investigation of more aggregation patters for other NFRs, such as scalability.
- Alignment of WS-Agreement [5] types with the ISO/IEC 19086 specification.

Furthermore, regarding the conceptual work for the MCSLA task, the following aspects need to be investigated in the future:

- Hierarchical structures of SLAs due to sub-contracting and how that affects our implementation.

- Expecting fully ISO/IEC 19086 compliant SLAs from the ACSmI discovery service and implementing an interpreter to be able to handle them.
- Consideration regarding developing an implementation of the ISO/IEC 19086 separate from the tools, as a library to be integrated in different projects. A first step is already done in separating the expression calculation inside the MCSLA Core Library.

6 References

[1] International Standards Organisation, “ISO/IEC 19086-2:Information technology -- Cloud computing -- Service level agreement (SLA) framework -- Part 2: Metric model,” 2017.

[2] I. Ul Haq and E. Schikuta, “Aggregation Patterns of Service Level Agreements,” FIT '10 8th International Conference on Frontiers of Information Technology, Islamabad, Pakistan, 2010.

[3] DECIDE, “Deliverable 3.14 - Intermediate multi-cloud native application composite CSLA definition”.

[4] International Standards Organisation, “ISO/IEC 19086-1: Information technology -- Cloud computing -- Service level agreement (SLA) framework -- Part 1: Overview and Concepts,” 2016.

[5] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke and M. Xu, “Web Services Agreement Specification (WS-Agreement),” Open Grid Forum.

[6] International Standards Organisation, “ISO/IEC 19086-3: Information technology -- Cloud computing -- Service level agreement (SLA) framework -- Part 3: Core conformance requirements,” 2017.

[7] International Standards Organisation, “ISO/IEC 19086-4: Information technology -- Cloud computing -- Service level agreement (SLA) framework -- Part 4: Security and privacy,” 2017.

[8] DECIDE, “Deliverable 2.2 - Detailed requirements specification V2,” 2018.

[9] International Standards Organisation, “Standards catalogue - ISO/TC2 - Quantities and units,” [Online]. Available: https://www.iso.org/committee/46202/x/catalogue/. [Accessed 26 11 2017].

[10] “The OpenAPI Specification,” [Online]. Available: https://github.com/OAI/OpenAPI-Specification. [Accessed 2018].

[11] “ReDoc - OpenAPI/Swagger-generated API Reference Documentation,” [Online]. Available: https://github.com/Rebilly/ReDoc. [Accessed 2018].

[12] OMG, “About the Semantics Of Business Vocabulary And Rules Specification Version 1.0,” [Online]. Available: https://www.omg.org/spec/SBVR/1.0/About-SBVR/. [Accessed 2018].

[13] «pdfmake - Client/server side PDF printing in pure JavaScript,» [En línea]. Available: http://pdfmake.org. [Último acceso: 2019].

[14] DECIDE, “Deliverable 3.11 - Intermediate multicloud native application controller,” 2018.

[15] “Eclipse Vert.x,” [Online]. Available: https://vertx.io/. [Accessed 2018].

[16] “Docker - Enterprise Container Platform,” [Online]. Available: https://www.docker.com/. [Accessed 2018].

[17] “React - A JavaScript library for building user interfaces,” [Online]. Available: https://reactjs.org/. [Accessed 2018].

[18] “Node.js,” [Online]. Available: https://nodejs.org/en/. [Accessed 2018].