Secure Provisioning of Cloud Services
based on SLA Management

SPECS Project - Deliverable 1.3

Module Interaction Protocols

Version no. 1.1
15 February 2016

The activities reported in this deliverable are partially supported
by the European Community's Seventh Framework Programme under grant agreement no. 610795.

Deliverable information

Deliverable no.:	D1.3
Deliverable title:	Module Interaction Protocols
Deliverable nature:	Report
Dissemination level:	Public
Contractual delivery:	15 February 2016
Actual delivery date:	15 February 2016
Author(s):	Massimiliano Rak, Alessandra De Benedictis (CeRICT)
Contributors:	Jolanda Modic (XLAB), Silviu Panica (IeAT), Adrian Spataru (IeAT), Madalina Erascu (IeAT), Ruben Trapero (TUDA), Alain Pannetrait (CSA)
Reviewers:	Umberto Villano (CeRICT), Silvio La Porta (EMC)
Contributors version 1.1:	Jolanda Modic (XLAB), Silviu Panica (IeAT), Adrian Spataru (IeAT), Madalina Erascu (IeAT), Ruben Trapero (TUDA), Alain Pannetrait (CSA)
Reviewers version 1.1:	Umberto Villano (CeRICT), Silvio La Porta (EMC)
Task contributing to the deliverable:	T1.3
Total number of pages:	43
Annexes	7

Executive summary

This deliverable is aimed at illustrating the interaction protocols designed to ensure the communication among the SPECS Framework's main modules (i.e., the SLA Platform, the Negotiation module, the Enforcement module and the Monitoring module) during the different phases of the SLA life-cycle.

As pointed out in D1.1.3, the main SPECS modules expose proper REST APIs, used for inter-modules communication. In this deliverable, the complete documentation of such APIs is provided along with the whole *SPECS data flow*, which defines the content and the format of the information shared among modules in order to accomplish the tasks related to SLA negotiation, enforcement and monitoring.

To summarize, this deliverable presents:

- The SLA API, the Services API and the Interoperability API, provided by the SLA Manager, the Service Manager and the Interoperability layer of the SLA Platform, respectively;
- The Negotiation API, offered by the SLO Manager of the Negotiation module;
- The Enforcement API, offered by the Planning, Implementation, Diagnosis and RDS components of the Enforcement module;
- The Monitoring API offered by the Monitoring module and the Monitoring Public API, also offered by the Monitoring module but based on the CSA's Cloud Trust Protocol;
- The Log API, belonging to the SPECS Vertical layer (cf. D1.1.3) and offered by the Auditing component of the Enforcement module.

In addition to these APIs, two further APIs belonging to the Vertical layer must be considered, namely the User Manager API, the Credentials API and the Security Tokens API. The User Manager API, offered by the User Manager component of the Vertical layer, will be discussed in D4.3.3, while Credentials API and Security Tokens API will be presented in D4.4.2.

Table of contents

Index of figures

Index of tables

1. Introduction

This document aims to illustrate in detail the interactions among the SPECS framework's main modules (Negotiation, Enforcement, Monitoring and SLA Platform) during the SLA life cycle and to provide a complete documentation of the SPECS REST APIs supporting these interactions. Hence, the objective of this document is to provide a technical reference for the developers.

The remainder of this document is structured as follows:

- Section 2 highlights the relationships with other deliverables before going into the details of the API specification;
- Section 3 illustrates the SPECS data flow and presents a detailed description of the behaviour and interactions of the core modules and of the SLA Platform during the five SLA's life cycle phases of Negotiation, Implementation, Monitoring, Remediation and Renegotiation;
- Section 4 reports an overview of the SPECS REST API with the aim of introducing the main functionalities provided by the specific calls to support the interactions shown in Section 3. In particular, the following APIs are introduced:
    - the *SLA API* (Section 4.1),
    - the *Services API* (Section 4.2),
    - the *Interoperability API* (Section 4.3),
    - the *Negotiation API* (Section 4.4),
    - the *Enforcement API* (Section 4.5),
    - *Log API* (Section 4.6),
    - *Monitoring API* (Section 4.7),
    - *Monitoring Public API* (Section 4.8).
- Section 5 reports the guidelines that have been followed by the SPECS Consortium partners when developing APIs, including the identification of resources, the specification of exchanged messages and a well-defined guidance on the use of response codes. Note that such guidelines are based on common conventions, best practices and standards for the development of REST APIs.

The complete specification of the SPECS REST APIs is provided in the Annexes A, B, C, D, E, F and G to this deliverable. For each API, we report the covered requirements, the involved resources, the detailed API calls and the adopted data models.

2. Relationship with other deliverables

This deliverable presents the activities conducted in Task 1.3, "Design of Module Interactions Protocols", and is based on the results achieved within WP1, WP2, WP3, and WP4 related to the design of the SLA Platform, the Negotiation module, the Enforcement module and the Monitoring module.

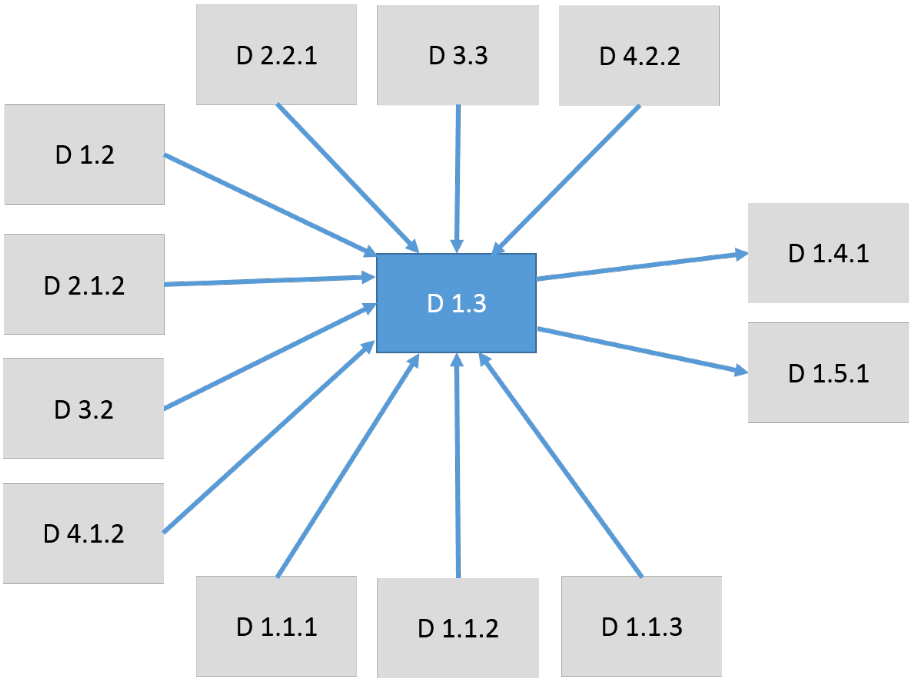

Figure 1. Relationship with other deliverables

Figure 1 shows the relationship between this deliverable and the others. As illustrated, the interaction protocols' definition is essentially based on the analysis of the behaviour of the involved modules as resulting from the requirements elicited in D1.2 (Platform requirements), D2.1.2 (Negotiation requirements), D3.2 (Monitoring requirements) and D4.1.2 (Enforcement requirements). Furthermore, it also depends on the design activities conducted within work packages from WP1 to WP4 (cf. D1.1.1, D1.1.2, D1.1.3, D2.2.1, D3.3, D4.2.2), which recognized relevant components belonging to each module and identified the offered high-level APIs, detailed in this deliverable.

D1.3 will constitute an input for deliverable D1.4.1, which will finalize the API specification and will also present a detailed list of the core services. Finally, D1.3 is an input for deliverable D1.5.1, which is aimed at the definition of integration scenarios.

3. Module Interactions

In this section, we recall the behaviour of the Negotiation, Enforcement and Monitoring modules and of the SLA Platform during the main phases of the SLA life cycle. In particular, in Section 3.1 we illustrate the SPECS data flow, which shows the information generated and processed in each of the SLA life cycle phases and outlines the main relationships existing among this data.

In Sections from 3.2 to 3.6, we detail such flow by presenting all the interactions among involved modules and components in each phase in form of sequence diagrams. In the diagrams, we also report the actual API calls involved in such interactions for completeness's sake. As mentioned in the Introduction, the complete documentation of APIs is provided in the Annexes to this document, to which the interest reader is redirected.

Further implementation details on internal negotiation, enforcement, and monitoring processes are available in dedicated deliverables (in D2.3.3 for negotiation, in D3.4.2 for monitoring, and in D4.3.2 for enforcement).

3.1. SPECS data flow

The whole SPECS data flow is depicted in Figure 2. It shows the main information involved in the different SLA life cycle phases and the components responsible for their generation and processing. All adopted data models, represented in XML and/or JSON format, will be discussed in the API documentation reported in the Annexes.

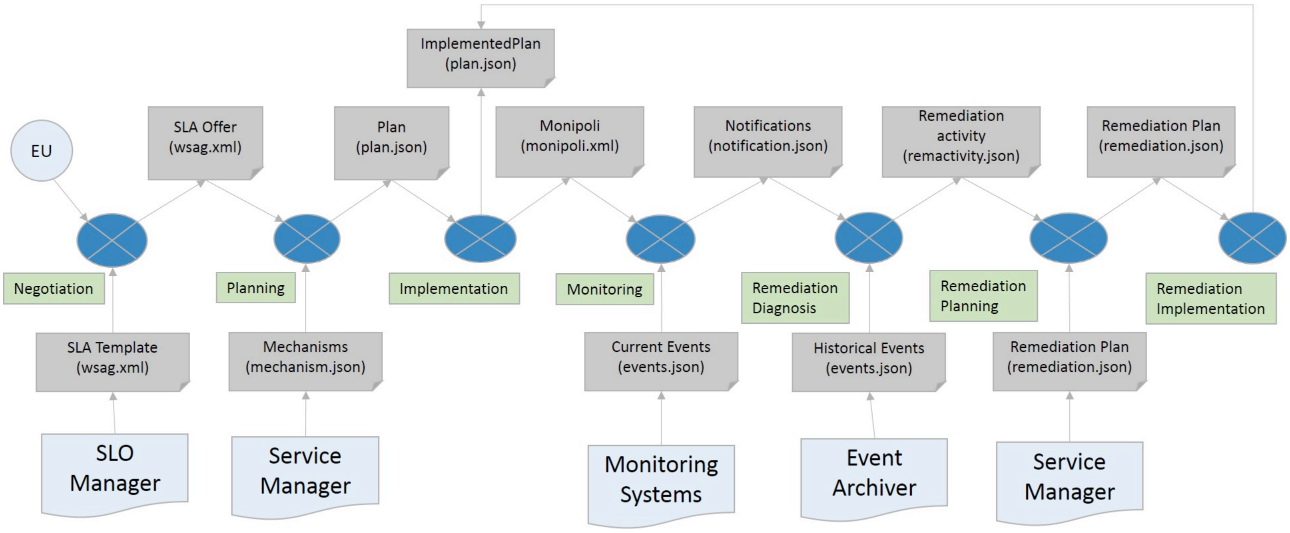
ImplementedPlan (plan.json)
EU
SLA Offer (wsag.xml)
Plan (plan.json)
Monipoli (monipoli.xml)
Notifications (notification.json)
Remediation activity (remactivity.json)
Remediation Plan (remediation.json)
Negotiation
Planning
Implementation
Monitoring
Remediation Diagnosis
Remediation Planning
Remediation Implementation
SLA Template (wsag.xml)
Mechanisms (mechanism.json)
Current Events (events.json)
Historical Events (events.json)
Remediation Plan (remediation.json)
SLO Manager
Service Manager
Monitoring Systems
Event Archiver
Service Manager

Figure 2. SPECS data flow

As shown in Figure 2, the SLA Negotiation phase relies upon an SLA Template, specified in a format compliant with WS-Agreement and produced by the SPECS Owner based on what it is willing to offer. SLA Templates are managed by the SLO Manager of the Negotiation module, and are used to build SLA Offers according to requirements specified by the End-user (EU).

After Negotiation, the resulting SLA Offer (i.e., the signed SLA) enters the SLA Implementation phase. Here, the Planning component builds an implementation plan for the proper deployment, activation and configuration of security mechanisms and related monitoring systems (maintained by the Service Manager) needed to actuate the signed SLA. Afterward, the Implementation component implements the plan by acquiring and configuring needed resources.

The final step of the SLA Implementation consists in updating the Monitoring Policy (*monipoli*), which is used to actually monitor the state of the SLA in the Monitoring phase. During the Monitoring phase, events generated by the deployed monitoring systems are collected and analysed. In case of suspicious behaviour, proper notifications are generated and sent to the Diagnosis component. These event notifications are analysed, together with the historical events stored by the Event Archiver, to check whether an alert or a violation occurred. In case of an alert or a violation, the actual Remediation phase takes place, where a remediation plan must be built and implemented according to the security mechanisms' metadata provided by the Service Manager for the mechanisms deployed in the SLA implementation phase.

The described process is detailed in the following sections, where we illustrate *all* the interactions among SPECS modules during the different SLA life cycle phases: as previously said, the interactions are presented through a set of detailed sequence diagrams that show all needed method invocations, corresponding to the REST API calls reported in the Annexes to this document.

3.2. SLA Negotiation

The SLA Negotiation phase is illustrated in the diagram in Figure 3. Note that, for each depicted invocation, we also reported the corresponding API call for completeness' sake: the reader is referred to the Annexes for a complete description of APIs.

As shown in figure, an End-user (EU) accesses the SPECS Application to start negotiation and is returned with a set of service offers, each based on one of the available SLA Templates, which are provided by the Negotiation module (to be precise, by the SLO Manager component). An SLA Template refers to a particular service (e.g. secure web container, secure storage) and contains: (i) the available cloud resources (i.e., providers, zones, types and maximum acquirable number of virtual machines), (ii) the capabilities the SPECS Owner is willing to offer with the service, (iii) the related security metrics and (iv) a set of default SLOs. SLA Templates are built by the SPECS Owner and maintained by the SLO Manager of the Negotiation module, which provides them to the SPECS Application.

The End-user, through the SPECS' Application interface, selects one of the service offers: the corresponding SLA Template is retrieved from the Negotiation module and an SLA is created for the End-user in the *pending* state. The SLA Template is used by the SPECS Application to show to the End-user all available security features: the End-user selects the capabilities he/she is interested in and specifies the related requested controls, selects the desired metrics and sets related SLOs.

The End-user's choices are forwarded by the SPECS Application to the Negotiation module, which starts a process that will end with the building of a set of compliant SLA Offers, returned to the SPECS Application. In particular:

1. a set of valid *supply chains,* representing different allocation combinations of mechanisms' components over needed resources that are required to fulfil the End-user's requests, is built with the help of Enforcement module (see Figure 4 for details);
2. for each supply chain, an SLA Offer is created;
3. for each SLA Offer, a new SLA is created in the SLA Platform;
4. created SLA Offers are ranked (through the Security Reasoner services) and then returned to the SPECS Application.

Ranked SLA Offers are first validated by the SPECS Application (the CSP's signature is verified) and then presented to the End-user.
The End-user selects the SLA Offer he/she prefers: such Offer is used to update the SLA created for the End-user in the SLA Platform at the beginning of negotiation. All other SLA Offers are removed from the SLA Platform, and corresponding supply chains are also deleted in the Enforcement module.
The selected SLA Offer, formally accepted by the End-user (SLA signature), is ready to be implemented, according to the process described in Section 3.3.

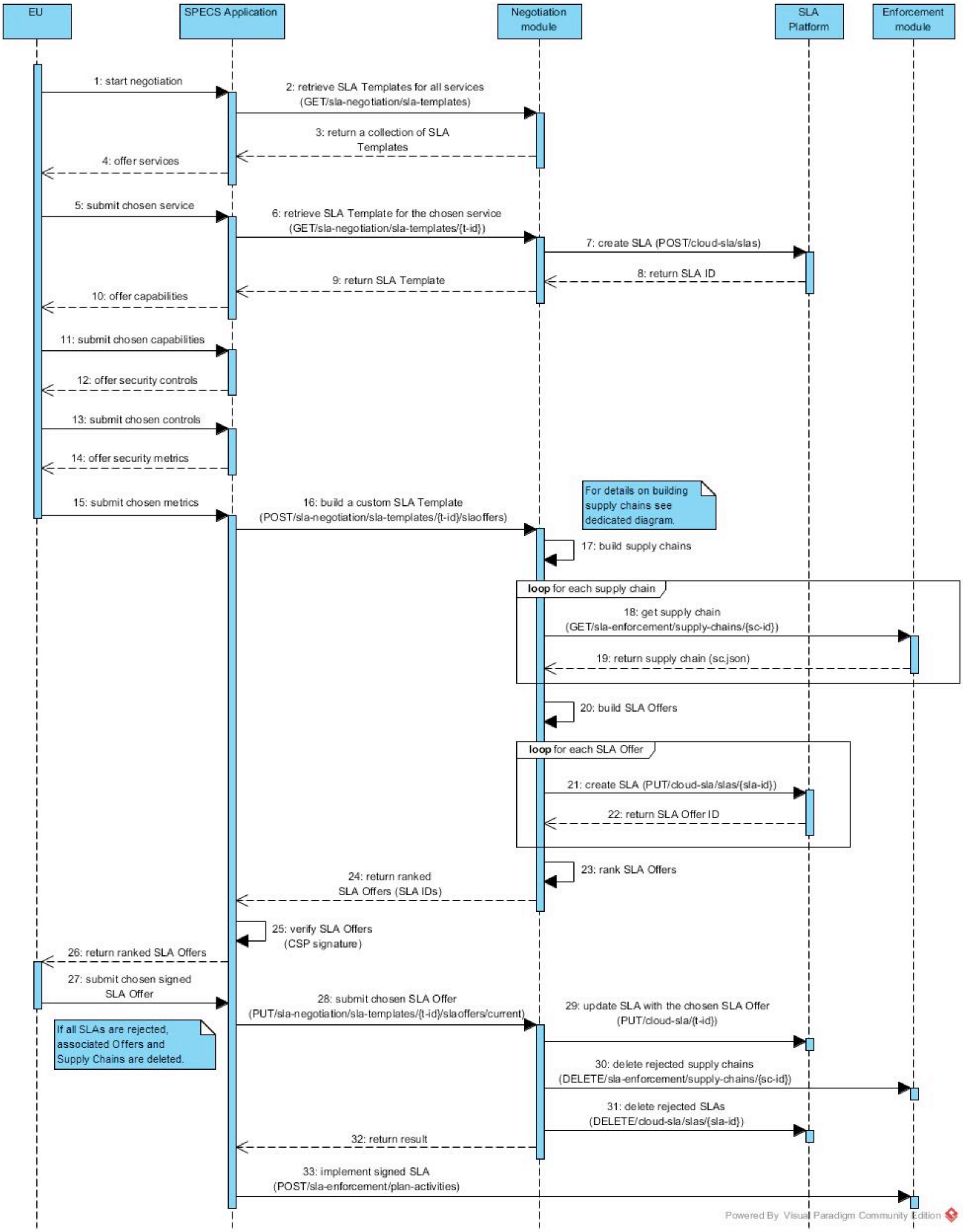
EU
SPECS Application
Negotiation module
SLA Platform
Enforcement module
1: start negotiation
2: retrieve SLA Templates for all services (GET/sla-negotiation/sla-templates)
3: return a collection of SLA Templates
4: offer services
5: submit chosen service
6: retrieve SLA Template for the chosen service (GET/sla-negotiation/sla-templates/{t-id})
7: create SLA (POST/cloud-sla/slas)
8: return SLA ID
9: return SLA Template
10: offer capabilities
11: submit chosen capabilities
12: offer security controls
13: submit chosen controls
14: offer security metrics
15: submit chosen metrics
16: build a custom SLA Template (POST/sla-negotiation/sla-templates/{t-id}/slaoffers)
For details on building supply chains see dedicated diagram.
17: build supply chains
loop for each supply chain
18: get supply chain (GET/sla-enforcement/supply-chains/{sc-id})
19: return supply chain (sc.json)
20: build SLA Offers
loop for each SLA Offer
21: create SLA (PUT/cloud-sla/slas/{sla-id})
22: return SLA Offer ID
23: rank SLA Offers
24: return ranked SLA Offers (SLA IDs)
25: verify SLA Offers (CSP signature)
26: return ranked SLA Offers
27: submit chosen signed SLA Offer
If all SLAs are rejected, associated Offers and Supply Chains are deleted.
28: submit chosen SLA Offer (PUT/sla-negotiation/sla-templates/{t-id}/slaoffers/current)
29: update SLA with the chosen SLA Offer (PUT/cloud-sla/{t-id})
30: delete rejected supply chains (DELETE/sla-enforcement/supply-chains/{sc-id})
31: delete rejected SLAs (DELETE/cloud-sla/slas/{sla-id})
32: return result
33: implement signed SLA (POST/sla-enforcement/plan-activities)
Powered By Visual Paradigm Community Edition

Figure 3. SLA Negotiation phase: overview

The process of supply chain building is illustrated in Figure 4. The Negotiation module (the Supply Chain Manager component) parses the SLA Template and extracts security metrics from the SLOs specified by the End-user.

For each specified metric, the associated security mechanisms are retrieved from the SLA Platform (from the Service Manager component). Given all these mechanisms, the Negotiation module identifies all different subsets of them that are able to fulfil End-user's requests.

For each set of mechanisms, the Negotiation module launches the process of supply chain building on the Enforcement module, which

1. keeps track of the *supply chain activity* by storing all the information relevant to the supply chain building process (SLA Template ID, available cloud resources, security mechanisms, security capabilities and SLOs);
2. retrieves mechanisms' metadata, including the list of software components that implement the mechanisms, and all deployment constraints that have been declared by the mechanisms' developers;
3. builds a set of valid (i.e., feasible) supply chains (cf. deliverable D4.3.2 for a detailed discussion of the process) and returns them to the Negotiation module.

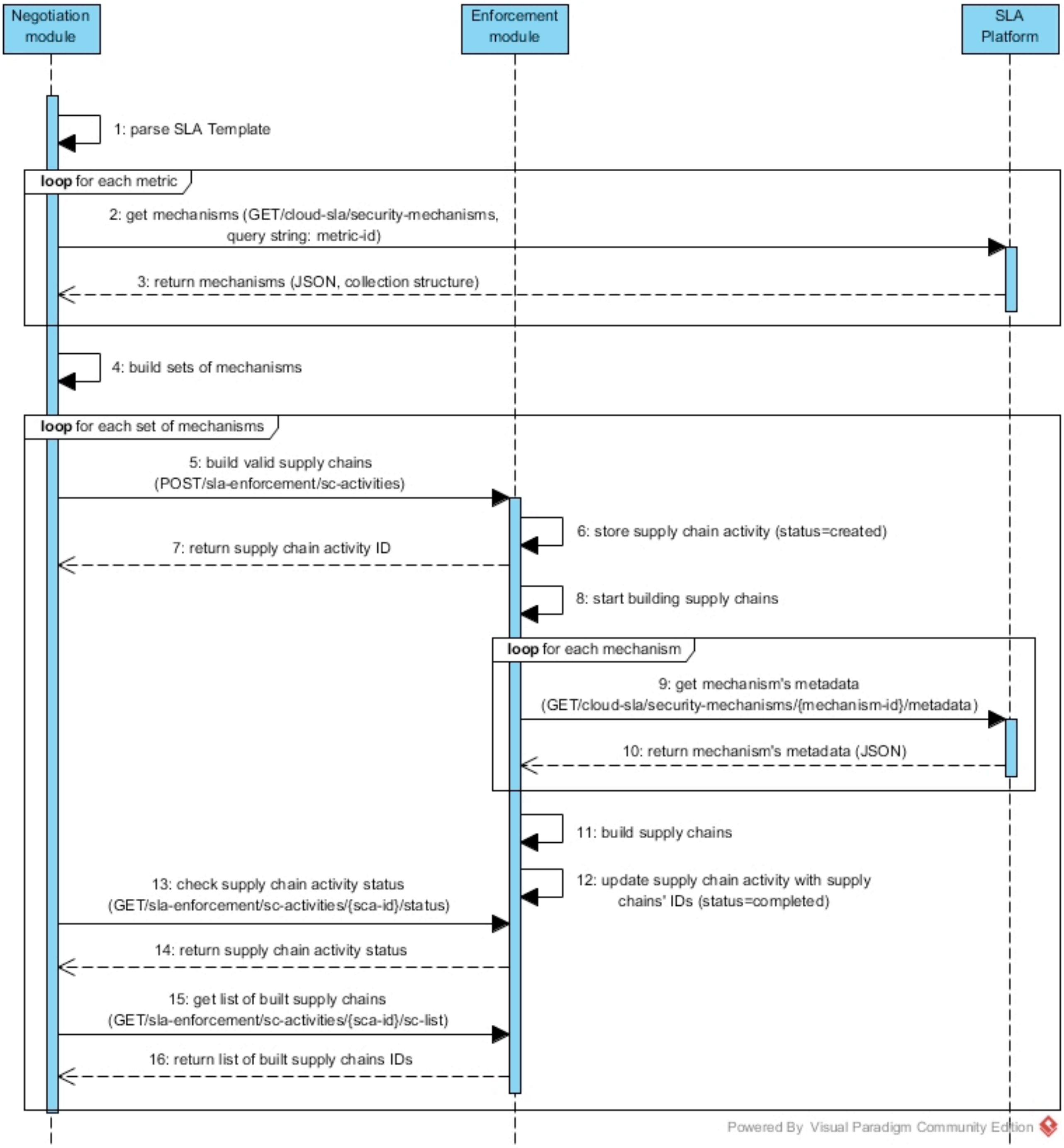
Negotiation module
Enforcement module
SLA Platform
1: parse SLA Template
loop for each metric
2: get mechanisms (GET/cloud-sla/security-mechanisms, query string: metric-id)
3: return mechanisms (JSON, collection structure)
4: build sets of mechanisms
loop for each set of mechanisms
5: build valid supply chains (POST/sla-enforcement/sc-activities)
6: store supply chain activity (status=created)
7: return supply chain activity ID
8: start building supply chains
loop for each mechanism
9: get mechanism's metadata (GET/cloud-sla/security-mechanisms/{mechanism-id}/metadata)
10: return mechanism's metadata (JSON)
11: build supply chains
12: update supply chain activity with supply chains' IDs (status=completed)
13: check supply chain activity status (GET/sla-enforcement/sc-activities/{sca-id}/status)
14: return supply chain activity status
15: get list of built supply chains (GET/sla-enforcement/sc-activities/{sca-id}/sc-list)
16: return list of built supply chains IDs
Powered By Visual Paradigm Community Edition

Figure 4. SLA Negotiation phase: supply chain building

3.3. SLA Implementation

SLA Implementation consists in deploying, activating and configuring security mechanisms and associated monitoring systems according to the supply chain that was built during negotiation. The Implementation phase is split into a *planning* step and the actual *implementation* step, orchestrated by respective components of the Enforcement module.

As shown in Figure 5, after the SLA signature, the SPECS Application invokes the Enforcement module to implement the SLA: it keeps track of the *planning activity* and:

1. retrieves the SLA to implement from the SLA Platform;
2. identifies the corresponding supply chain, containing information on the resources to acquire and on the allocation of components on these resources;
3. extracts the information about the resources and about SLOs from the SLA and retrieves the mechanisms to deploy;
4. builds the implementation plan for the supply chain and stores the plan;
5. implements the plan.

The plan is implemented by the Implementation component of the Enforcement module. Details about the actual Implementation are given in Figure 6. As shown, the Implementation component keeps track of the *implementation activity* and:

1. retrieves the implementation plan;
2. parses the implementation plan and, for each cloud resource in the plan, it configures the resource with a proper recipe;

After implementation, as shown in Figure 5, the Enforcement module updates the Monitoring Policy (*monipoli*) in the Monitoring module: the SLA enters the *observed* state.

Finally, note that, during the whole process, all component activations and deactivations as well as service activations are logged by the Auditing component.

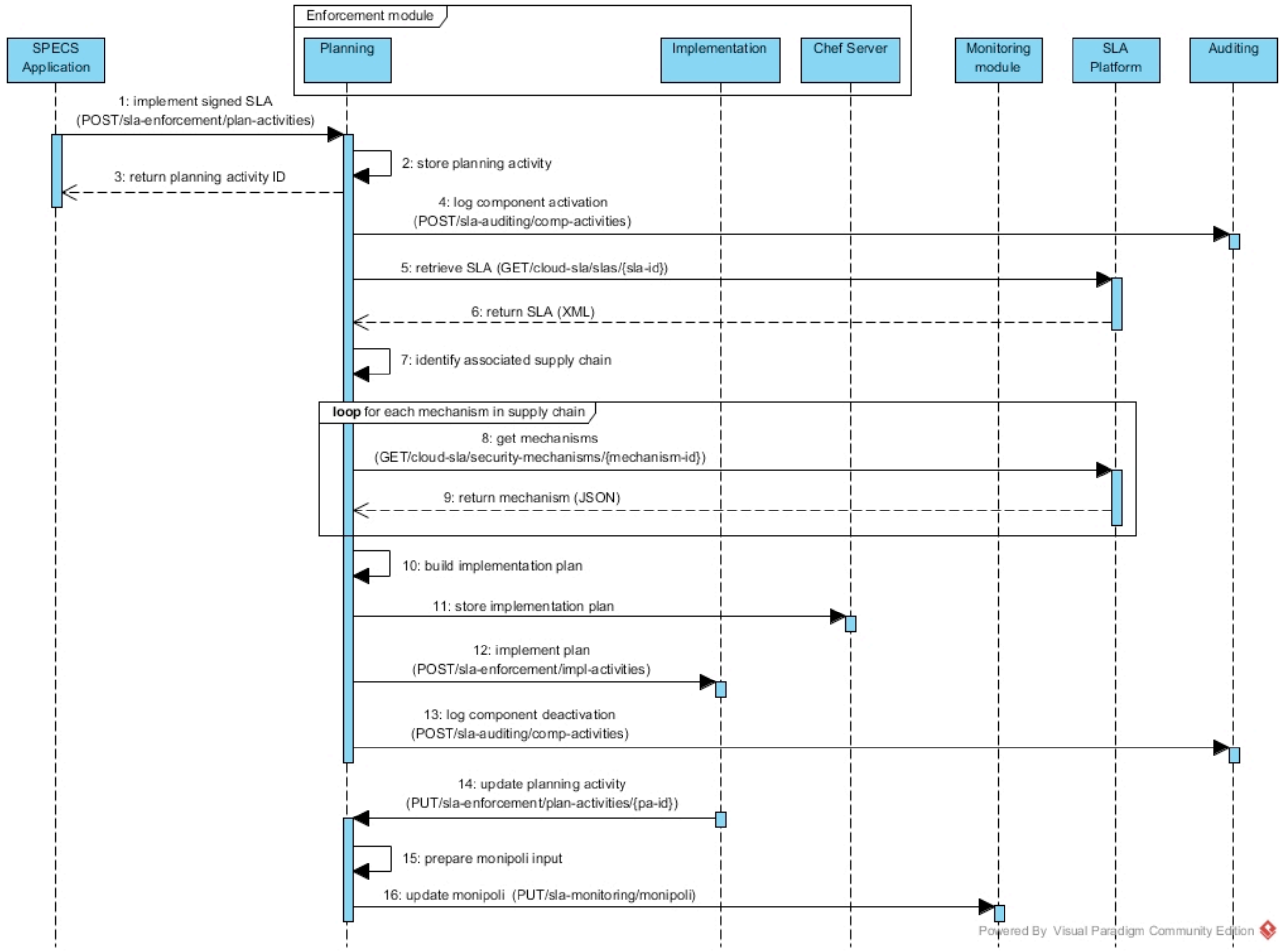
Enforcement module
SPECS Application
Planning
Implementation
Chef Server
Monitoring module
SLA Platform
Auditing
1: implement signed SLA (POST/sla-enforcement/plan-activities)
2: store planning activity
3: return planning activity ID
4: log component activation (POST/sla-auditing/comp-activities)
5: retrieve SLA (GET/cloud-sla/slas/{sla-id})
6: return SLA (XML)
7: identify associated supply chain
loop for each mechanism in supply chain
8: get mechanisms (GET/cloud-sla/security-mechanisms/{mechanism-id})
9: return mechanism (JSON)
10: build implementation plan
11: store implementation plan
12: implement plan (POST/sla-enforcement/impl-activities)
13: log component deactivation (POST/sla-auditing/comp-activities)
14: update planning activity (PUT/sla-enforcement/plan-activities/{pa-id})
15: prepare monipoli input
16: update monipoli (PUT/sla-monitoring/monipoli)
Powered By Visual Paradigm Community Edition

Figure 5. SLA Implementation phase: building the plan

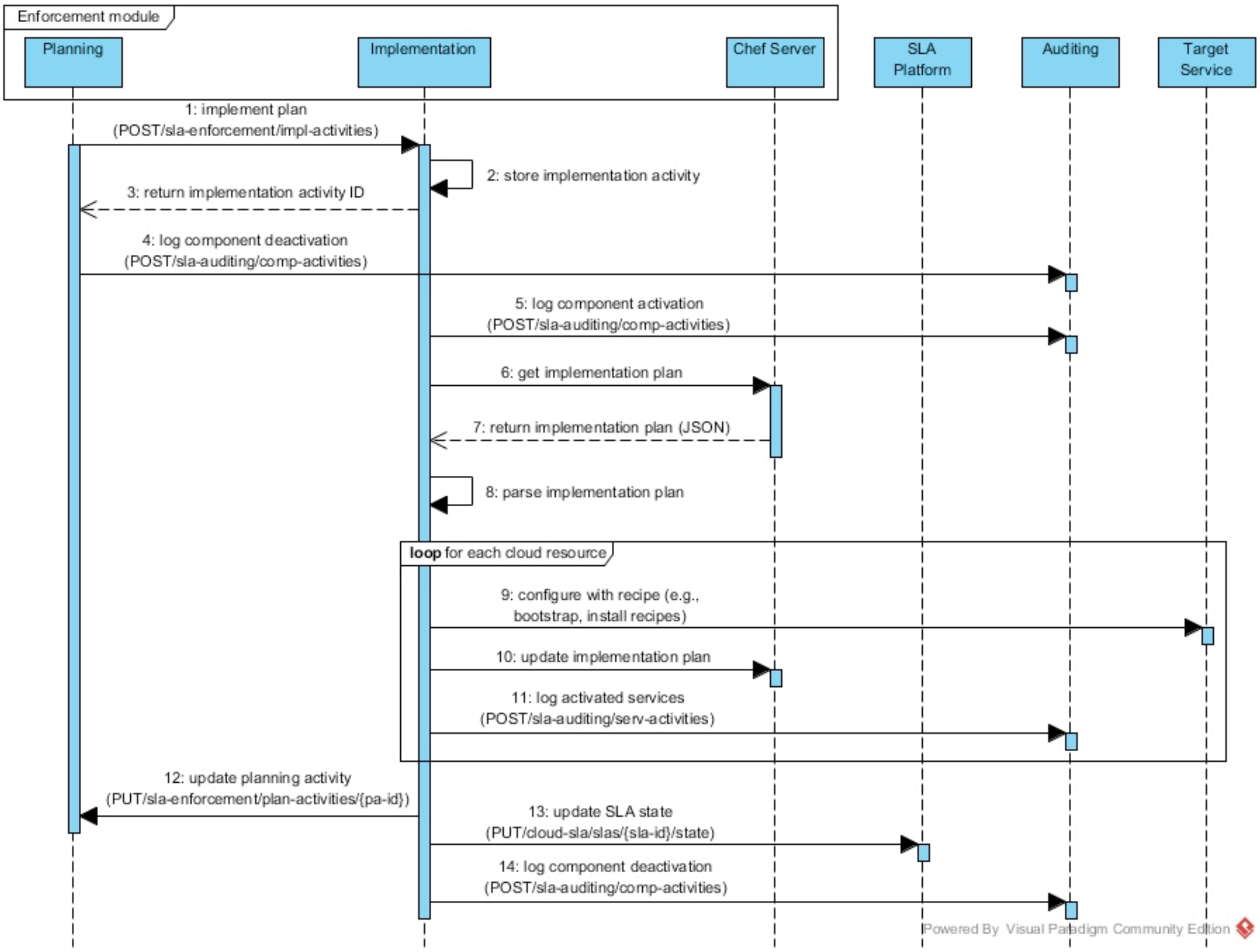
Enforcement module
Planning
Implementation
Chef Server
SLA Platform
Auditing
Target Service
1: implement plan (POST/sla-enforcement/impl-activities)
2: store implementation activity
3: return implementation activity ID
4: log component deactivation (POST/sla-auditing/comp-activities)
5: log component activation (POST/sla-auditing/comp-activities)
6: get implementation plan
7: return implementation plan (JSON)
8: parse implementation plan
loop for each cloud resource
9: configure with recipe (e.g., bootstrap, install recipes)
10: update implementation plan
11: log activated services (POST/sla-auditing/serv-activities)
12: update planning activity (PUT/sla-enforcement/plan-activities/{pa-id})
13: update SLA state (PUT/cloud-sla/slas/{sla-id}/state)
14: log component deactivation (POST/sla-auditing/comp-activities)
Powered By Visual Paradigm Community Edition

Figure 6. SLA Implementation phase: implementing the plan

3.4. SLA Monitoring

As previously discussed, after the SLA is signed, it is passed from the Negotiation module to the Enforcement module. The Enforcement module extracts from the SLA all the information needed to implement it and the information required to configure the related monitoring components. For every new signed SLA, the Enforcement module will configure the required monitoring components (deployed on acquired resources) with new rules, and will update the Monitoring Policy (element of the Monitoring module) to be adapted to the new SLA to monitor. Therefore, the monitoring process consists of applying a set of rules and filters that depend on the SLA content.

As shown in Figure 7, proper Monitoring Adapters (belonging to the Enforcement module and deployed on the target services' resources) keep collecting raw monitoring information from the target services and push related events to the Monitoring module. The information collected by adapters is relevant for the SLA to monitor and defined by the Enforcement module at configuration time.

As depicted, events are processed and aggregated within the Monitoring module. Moreover, all events (both raw and aggregated ones) are logged (in the Event Archiver) for further processing (e.g., during the Diagnosis phase), and they are continuously analysed against the current Monitoring Policy. In case the policy is not respected, the Monitoring module notifies the detected suspicious *monitoring events* to the Enforcement module for their classification and handling.

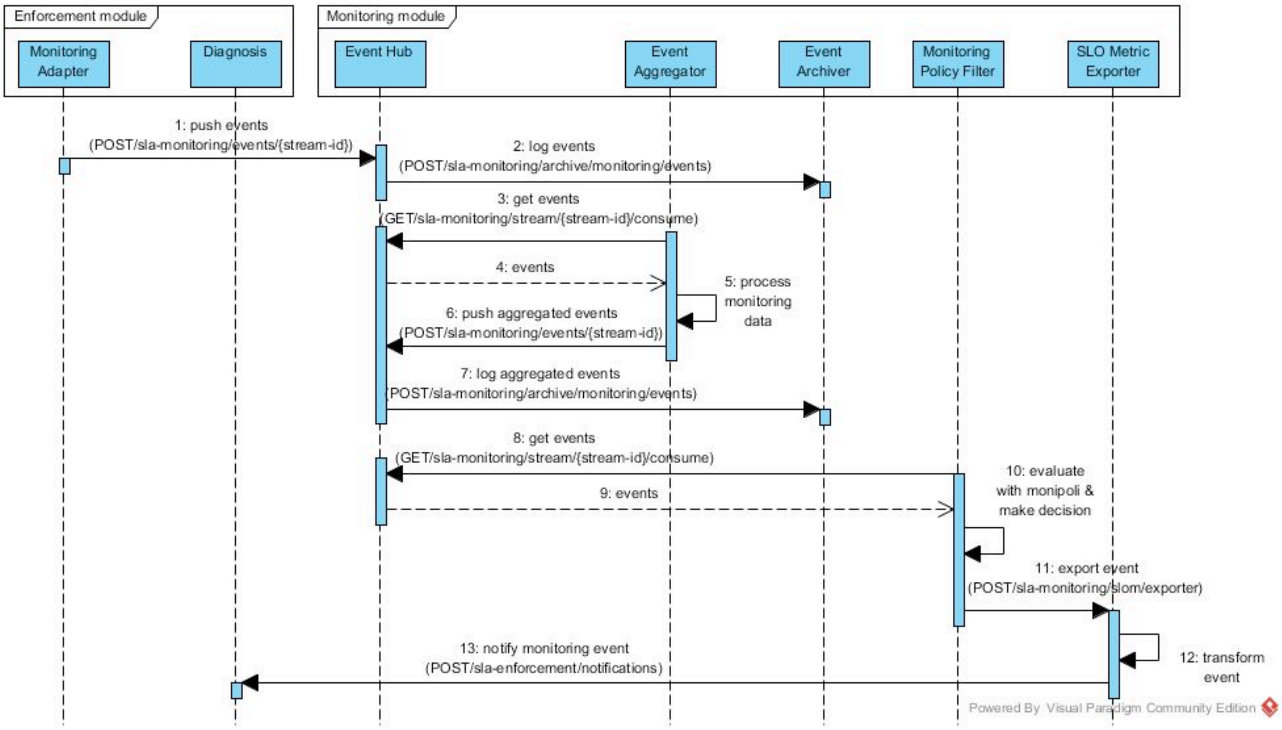
Enforcement module
Monitoring Adapter
Diagnosis
Monitoring module
Event Hub
Event Aggregator
Event Archiver
Monitoring Policy Filter
SLO Metric Exporter
1: push events (POST/sla-monitoring/events/{stream-id})
2: log events (POST/sla-monitoring/archive/monitoring/events)
3: get events (GET/sla-monitoring/stream/{stream-id}/consume)
4: events
5: process monitoring data
6: push aggregated events (POST/sla-monitoring/events/{stream-id})
7: log aggregated events (POST/sla-monitoring/archive/monitoring/events)
8: get events (GET/sla-monitoring/stream/{stream-id}/consume)
9: events
10: evaluate with monipoli & make decision
11: export event (POST/sla-monitoring/slom/exporter)
12: transform event
13: notify monitoring event (POST/sla-enforcement/notifications)
Powered By Visual Paradigm Community Edition

Figure 7. Monitoring phase: overview

3.5. SLA Remediation

Each monitoring event that can potentially report violation of some SLA has to be carefully analysed. The SLA remediation phase consists in the analysis of the monitoring events (performed by the Diagnosis component) and in the possible activation of redressing/remediation activities (performed by the Remediation Decision System component).

As shown in Figure 8, when the Enforcement module (through the Diagnosis component) receives monitoring events from the Monitoring module, it tracks the *diagnosis activity* and retrieves the affected SLA from the SLA Platform in order to start the analysis. From the SLA, the Enforcement module component retrieves all information about the planning activity related to the SLA and the implementation plan. Afterwards, the Enforcement module identifies affected SLOs and performs the following steps:

1. Classification: it determines whether the monitoring event represents an alert, a violation, or a false positive. The SLA state is updated accordingly;
2. Analysis: when false positives are discarded, it evaluates the effect that the alert/violation has on the SLA by calculating the risk/severity level of the alert/violation. This is done on the basis of impact that the alert/violation has on each affected SLO defined in the SLA. The results of the analysis are stored in the Auditing component;
3. Prioritization: according to the risk/severity level of an alert/violation, the affected SLA is put in a priority queue which guarantees that the violated SLAs with the highest risk/severity level are remediated first.

Before each alerted/violated SLA is pushed to the RDS component to find the best remediation action, the Enforcement module has to verify if the conditions of the alert/violations still persist, i.e., it has to check the measurement results related to the affected SLOs. For this purpose, the Enforcement module communicates with the Monitoring module.

In case the alert/violation of an SLA persist, the actual remediation takes place. As shown in Figure 9, the Enforcement module (through the Remediation Decision System component) tracks the *remediation activity* and

1. updates the SLA state from *Alerted*/*Violated* to *proactive redressing/remediating;*
2. retrieves the associated implementation plan;
3. determines which mechanisms were affected and, for each affected mechanism, extracts the associated remediation plan, stored in the SLA Platform;
4. builds an SLA remediation plan.

The remediation plan is implemented by the Implementation component of the Enforcement module (details are reported in Figure 10). If the remediation is successful, the state of the SLA is updated from *proactive redressing/remediating* to *Observed*. If none of the available reactions is able to recover from an alert/violation, the End-user is notified about the conditions of the event. Communication with the End-user is carried out through the SPECS Application.

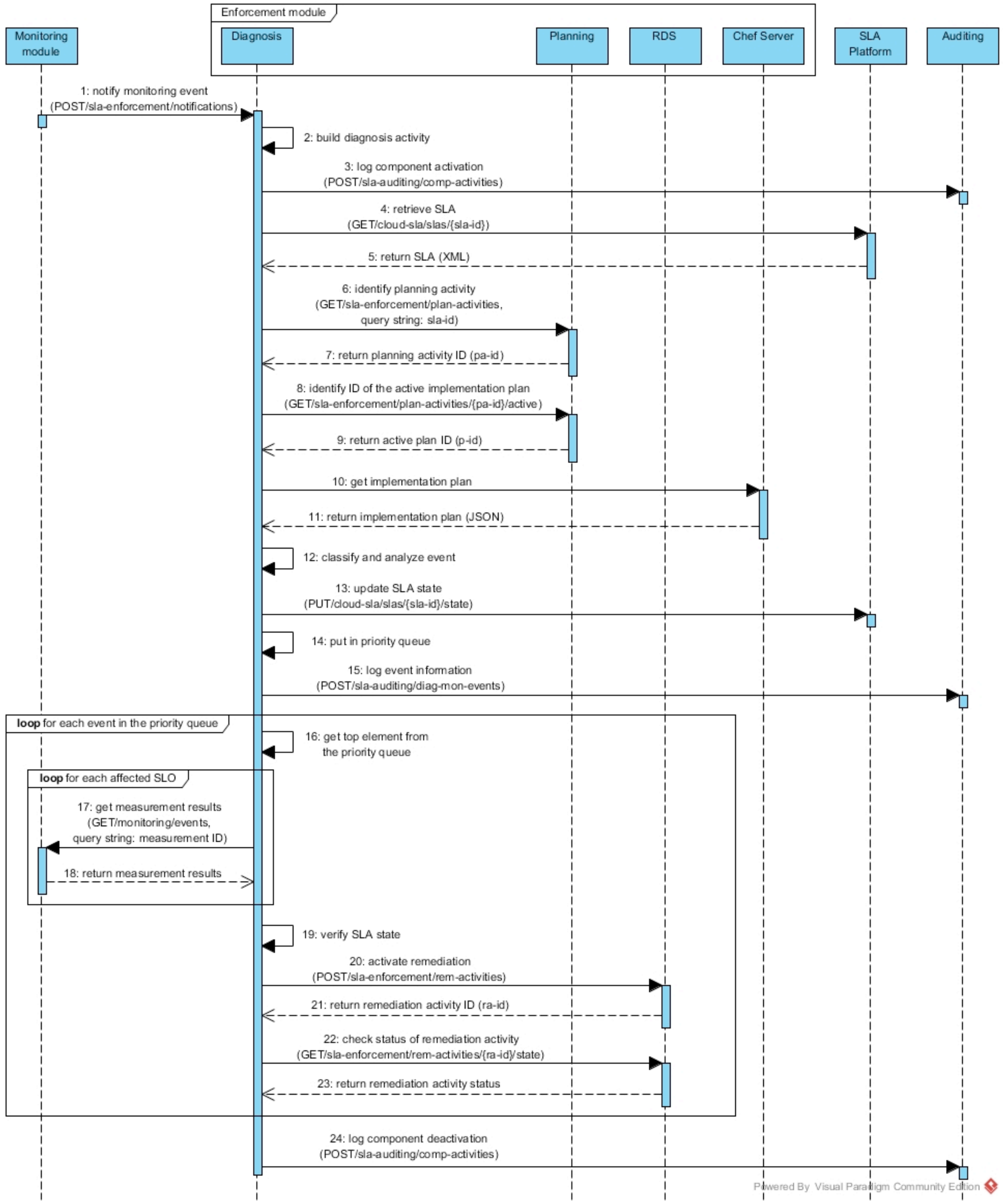
Enforcement module
Monitoring module
Diagnosis
Planning
RDS
Chef Server
SLA Platform
Auditing
1: notify monitoring event (POST/sla-enforcement/notifications)
2: build diagnosis activity
3: log component activation (POST/sla-auditing/comp-activities)
4: retrieve SLA (GET/cloud-sla/slas/{sla-id})
5: return SLA (XML)
6: identify planning activity (GET/sla-enforcement/plan-activities, query string: sla-id)
7: return planning activity ID (pa-id)
8: identify ID of the active implementation plan (GET/sla-enforcement/plan-activities/{pa-id}/active)
9: return active plan ID (p-id)
10: get implementation plan
11: return implementation plan (JSON)
12: classify and analyze event
13: update SLA state (PUT/cloud-sla/slas/{sla-id}/state)
14: put in priority queue
15: log event information (POST/sla-auditing/diag-mon-events)
loop for each event in the priority queue
16: get top element from the priority queue
loop for each affected SLO
17: get measurement results (GET/monitoring/events, query string: measurement ID)
18: return measurement results
19: verify SLA state
20: activate remediation (POST/sla-enforcement/rem-activities)
21: return remediation activity ID (ra-id)
22: check status of remediation activity (GET/sla-enforcement/rem-activities/{ra-id}/state)
23: return remediation activity status
24: log component deactivation (POST/sla-auditing/comp-activities)
Powered By Visual Paradigm Community Edition

Figure 8. Remediation phase: diagnosis process

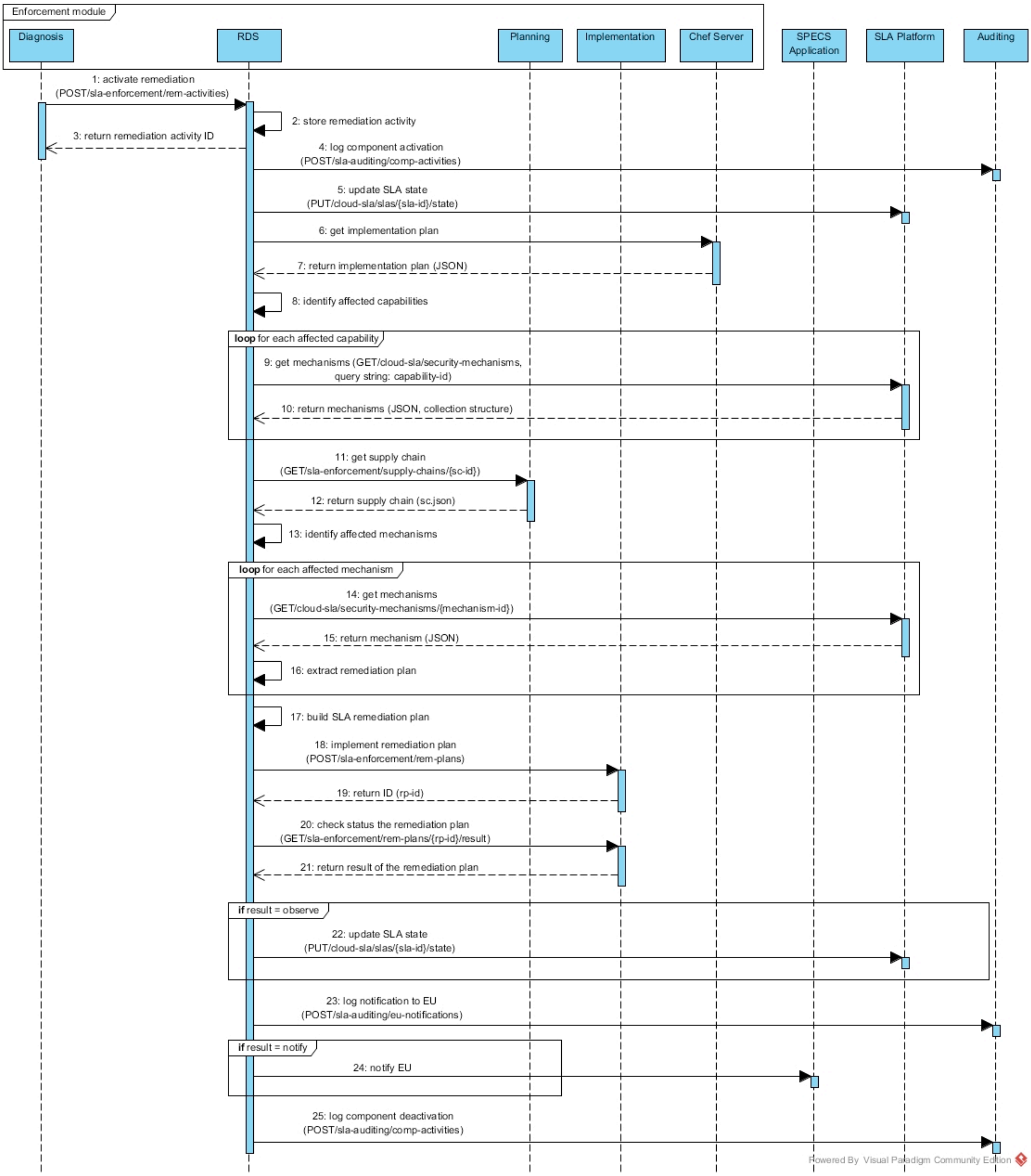
Enforcement module
Diagnosis
RDS
Planning
Implementation
Chef Server
SPECS Application
SLA Platform
Auditing
1: activate remediation (POST/sla-enforcement/rem-activities)
2: store remediation activity
3: return remediation activity ID
4: log component activation (POST/sla-auditing/comp-activities)
5: update SLA state (PUT/cloud-sla/slas/{sla-id}/state)
6: get implementation plan
7: return implementation plan (JSON)
8: identify affected capabilities
loop for each affected capability
9: get mechanisms (GET/cloud-sla/security-mechanisms, query string: capability-id)
10: return mechanisms (JSON, collection structure)
11: get supply chain (GET/sla-enforcement/supply-chains/{sc-id})
12: return supply chain (sc.json)
13: identify affected mechanisms
loop for each affected mechanism
14: get mechanisms (GET/cloud-sla/security-mechanisms/{mechanism-id})
15: return mechanism (JSON)
16: extract remediation plan
17: build SLA remediation plan
18: implement remediation plan (POST/sla-enforcement/rem-plans)
19: return ID (rp-id)
20: check status the remediation plan (GET/sla-enforcement/rem-plans/{rp-id}/result)
21: return result of the remediation plan
if result = observe
22: update SLA state (PUT/cloud-sla/slas/{sla-id}/state)
23: log notification to EU (POST/sla-auditing/eu-notifications)
if result = notify
24: notify EU
25: log component deactivation (POST/sla-auditing/comp-activities)
Powered By Visual Paradigm Community Edition

Figure 9. Remediation phase: building a remediation plan

Figure 10 shows the implementation of the remediation plan. The Implementation component stores the remediation plan and, for each action in the plan, properly reconfigures resources. After each reconfiguration, it updates the implementation plan with latest changes and verifies, by means of measurements collected on the new configuration and provided by the Event Archiver of the Monitoring module, if the remediation action was successful. As mentioned before, if remediation fails, the End-user is notified.

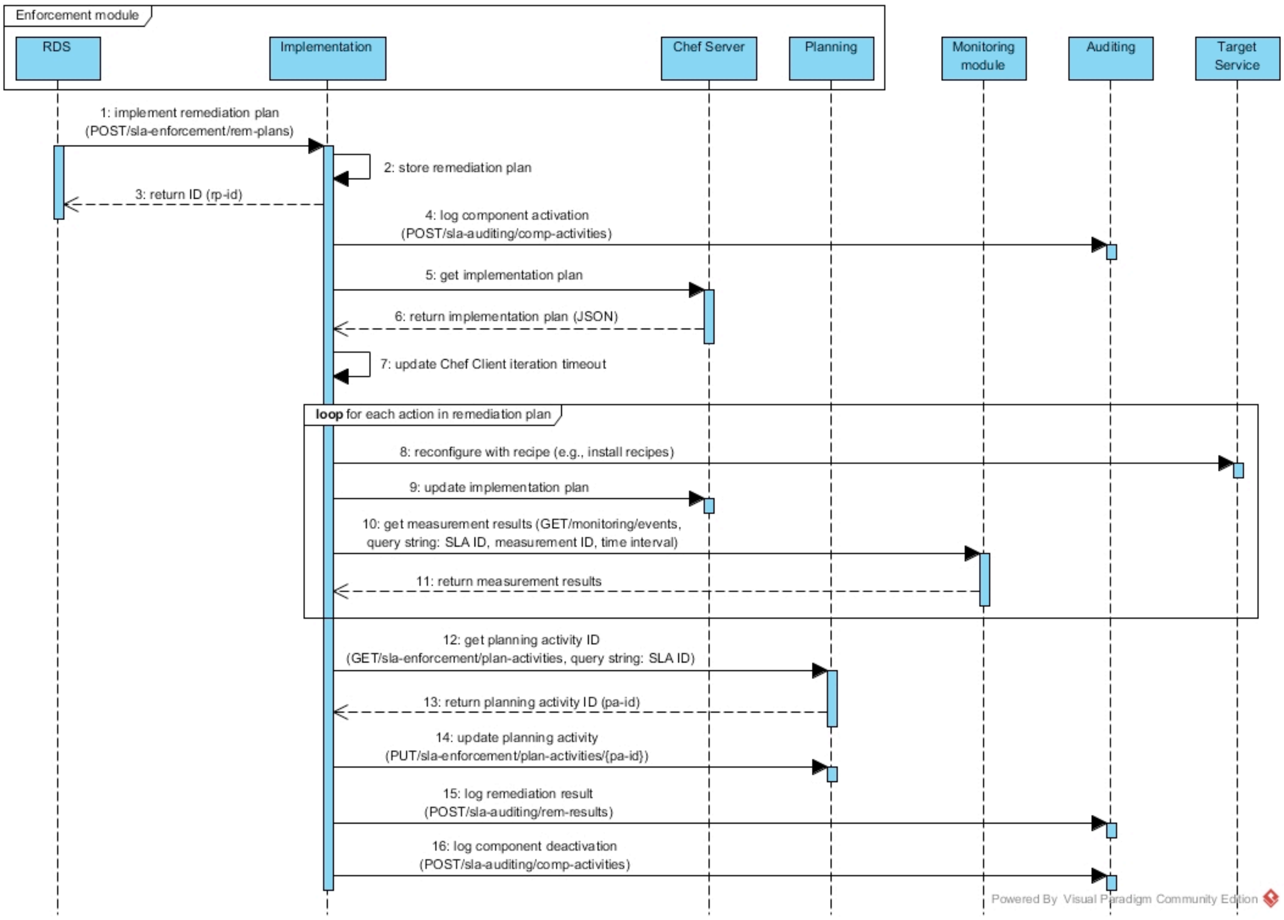
Enforcement module
RDS
Implementation
Chef Server
Planning
Monitoring module
Auditing
Target Service
1: implement remediation plan (POST/sla-enforcement/rem-plans)
2: store remediation plan
3: return ID (rp-id)
4: log component activation (POST/sla-auditing/comp-activities)
5: get implementation plan
6: return implementation plan (JSON)
7: update Chef Client iteration timeout
loop for each action in remediation plan
8: reconfigure with recipe (e.g., install recipes)
9: update implementation plan
10: get measurement results (GET/monitoring/events, query string: SLA ID, measurement ID, time interval)
11: return measurement results
12: get planning activity ID (GET/sla-enforcement/plan-activities, query string: SLA ID)
13: return planning activity ID (pa-id)
14: update planning activity (PUT/sla-enforcement/plan-activities/{pa-id})
15: log remediation result (POST/sla-auditing/rem-results)
16: log component deactivation (POST/sla-auditing/comp-activities)
Powered By Visual Paradigm Community Edition

Figure 10. Remediation phase: implementing the remediation plan

3.6. SLA Renegotiation

SLAs are subject to renegotiations. These changes can occur due to the fact the some of the services, SLOs, or SLO values are no longer available (i.e., due to unresolved SLA violations), or they can be related to changes required by any of the parties involved in the service provisioning.

It is worth noticing that the SLA signed between the SPECS Owner and the End-user is treated as an atomic contract: any change produced in any part of it, no matter how small it is, leads to the generation, renegotiation and signing of a new SLA.

Re-negotiation of an SLA can be triggered by either the End-user or the SPECS Enforcement module. The first case may occur when the EU wants to remove or add a new capability or an SLO to a running service or he/she wants to modify some specific value of some SLO (for example, the EU wants to change the encryption algorithm or wants to increase the frequency of the periodic backups). The second case, namely re-negotiation triggered by SPECS, may occur in presence of an unresolved violation or alert.

In both cases, the service being delivered is subject to changes (including termination) and may imply the update of the related supply chain.

Figure 11 illustrates the interactions in case of a re-negotiation triggered by the End-user. As shown, the End-user starts the re-negotiation process by accessing the SPECS Application. The SPECS Application retrieves the End-user's SLA from the SLA Platform, updates its state to *re-negotiating* and gets the related SLA Template from the Negotiation module. The SLA Template is updated with contents of the End-user's signed SLA (previously agreed capabilities, controls, and SLO values are set as default offers). At this point, the End-user can express his/her requirements as done during the Negotiation phase, by selecting desired capabilities, controls and SLOs.

A new set of supply chains is generated (note that they may overlap with the old supply chain associated with the SLA), which is used by the Negotiation module to build new SLA Offers, ranked and submitted to the End-user. If the End-user accepts one of these Offers, the old SLA is replaced with the new one.

Figure 12 shows what happens when re-negotiation is triggered after an alert or a violation by the CSP/SPECS Enforcement. The behaviour is very similar to the one just described for re-negotiation triggered by the End-user, except that the Application first updates the SLA Template according to needed changes (unavailability of certain SLO values, SLOs, or capabilities) and then notifies the End-user to let him/her start the re-negotiation.

Once the new SLA has been created, it has to be implemented. The steps performed to implement a re-negotiated SLA are shown in Figure 13. The Planning component updates the planning activity and retrieves the associated supply chain and implementation plan. The new SLA and the new supply chain are checked against the old implementation plan, in order to build a reaction plan that includes all needed actions to take to fulfil the new SLA. Basically, the re-negotiation is treated as an alert or a violation event: fake violation events are generated and handled as shown previously for remediation. For example, if for the re-negotiated SLA we need to acquire a new VM, we generate a violation that states unavailability of a VM. Such violation is remediated by acquiring a new VM. If, for example, an EU re-negotiated higher frequency of backups, we generate a violation stating that backup frequency is too low, and such violation is remediated by reconfiguring the frequency.

If remediation fails, then it means that the implementation of the re-negotiated SLA did not succeed. The End-user is hence notified.

SLA Termination can be seen as a particular type of re-negotiation. As shown in Figure 14, the SLA Application triggers the update of the implementation plan (either after a termination request from the End-user or after a violation), and a new plan is built which includes all recipes to terminate active services. Even in this case, fake violation events are generated that lead to the execution of remediation actions consisting in the termination of all services related to an SLA.

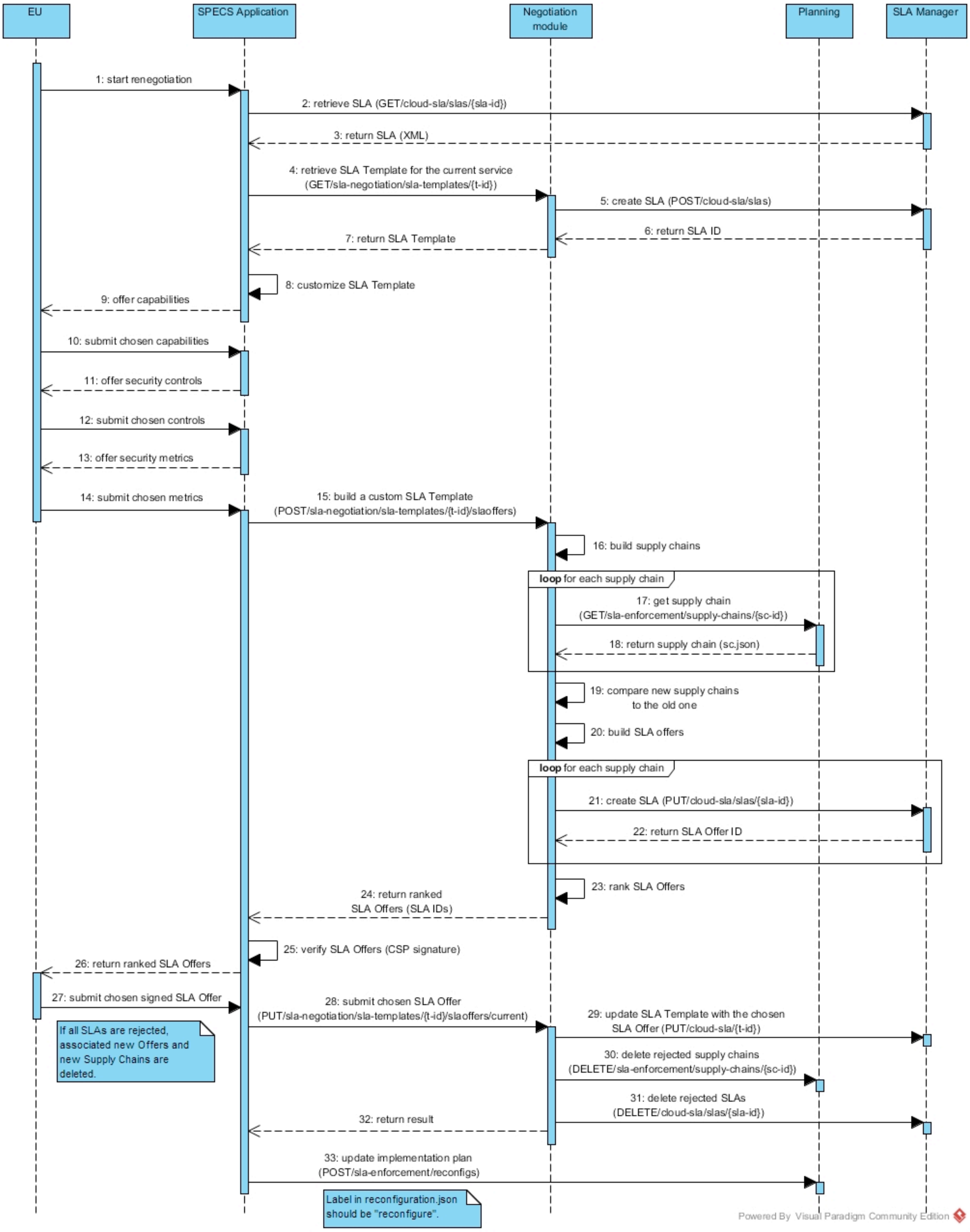
EU
SPECS Application
Negotiation module
Planning
SLA Manager
1: start renegotiation
2: retrieve SLA (GET/cloud-sla/slas/{sla-id})
3: return SLA (XML)
4: retrieve SLA Template for the current service (GET/sla-negotiation/sla-templates/{t-id})
5: create SLA (POST/cloud-sla/slas)
6: return SLA ID
7: return SLA Template
8: customize SLA Template
9: offer capabilities
10: submit chosen capabilities
11: offer security controls
12: submit chosen controls
13: offer security metrics
14: submit chosen metrics
15: build a custom SLA Template (POST/sla-negotiation/sla-templates/{t-id}/slaoffers)
16: build supply chains
loop for each supply chain
17: get supply chain (GET/sla-enforcement/supply-chains/{sc-id})
18: return supply chain (sc.json)
19: compare new supply chains to the old one
20: build SLA offers
loop for each supply chain
21: create SLA (PUT/cloud-sla/slas/{sla-id})
22: return SLA Offer ID
23: rank SLA Offers
24: return ranked SLA Offers (SLA IDs)
25: verify SLA Offers (CSP signature)
26: return ranked SLA Offers
27: submit chosen signed SLA Offer
If all SLAs are rejected, associated new Offers and new Supply Chains are deleted.
28: submit chosen SLA Offer (PUT/sla-negotiation/sla-templates/{t-id}/slaoffers/current)
29: update SLA Template with the chosen SLA Offer (PUT/cloud-sla/{t-id})
30: delete rejected supply chains (DELETE/sla-enforcement/supply-chains/{sc-id})
31: delete rejected SLAs (DELETE/cloud-sla/slas/{sla-id})
32: return result
33: update implementation plan (POST/sla-enforcement/reconfigs)
Label in reconfiguration.json should be "reconfigure".
Powered By Visual Paradigm Community Edition

Figure 11. Re-negotiation phase: re-negotiation started by the EU

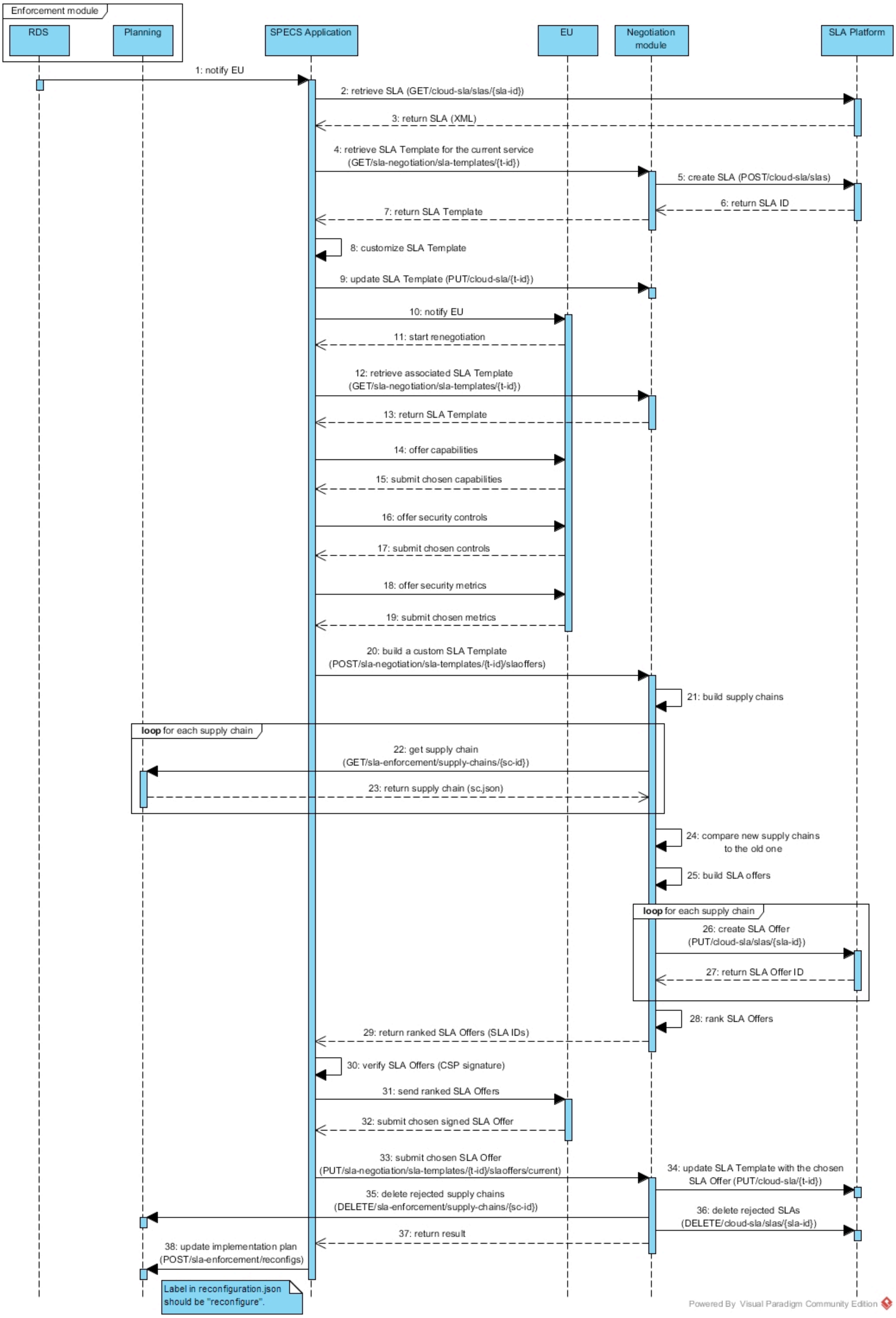
Enforcement module
RDS
Planning
SPECS Application
EU
Negotiation module
SLA Platform
1: notify EU
2: retrieve SLA (GET/cloud-sla/slas/{sla-id})
3: return SLA (XML)
4: retrieve SLA Template for the current service (GET/sla-negotiation/sla-templates/{t-id})
5: create SLA (POST/cloud-sla/slas)
6: return SLA ID
7: return SLA Template
8: customize SLA Template
9: update SLA Template (PUT/cloud-sla/{t-id})
10: notify EU
11: start renegotiation
12: retrieve associated SLA Template (GET/sla-negotiation/sla-templates/{t-id})
13: return SLA Template
14: offer capabilities
15: submit chosen capabilities
16: offer security controls
17: submit chosen controls
18: offer security metrics
19: submit chosen metrics
20: build a custom SLA Template (POST/sla-negotiation/sla-templates/{t-id}/slaoffers)
21: build supply chains
loop for each supply chain
22: get supply chain (GET/sla-enforcement/supply-chains/{sc-id})
23: return supply chain (sc.json)
24: compare new supply chains to the old one
25: build SLA offers
loop for each supply chain
26: create SLA Offer (PUT/cloud-sla/slas/{sla-id})
27: return SLA Offer ID
28: rank SLA Offers
29: return ranked SLA Offers (SLA IDs)
30: verify SLA Offers (CSP signature)
31: send ranked SLA Offers
32: submit chosen signed SLA Offer
33: submit chosen SLA Offer (PUT/sla-negotiation/sla-templates/{t-id}/slaoffers/current)
34: update SLA Template with the chosen SLA Offer (PUT/cloud-sla/{t-id})
35: delete rejected supply chains (DELETE/sla-enforcement/supply-chains/{sc-id})
36: delete rejected SLAs (DELETE/cloud-sla/slas/{sla-id})
37: return result
38: update implementation plan (POST/sla-enforcement/reconfigs)
Label in reconfiguration.json should be "reconfigure".
Powered By Visual Paradigm Community Edition

Figure 12. Re-negotiation phase: re-negotiation after an alert/violation

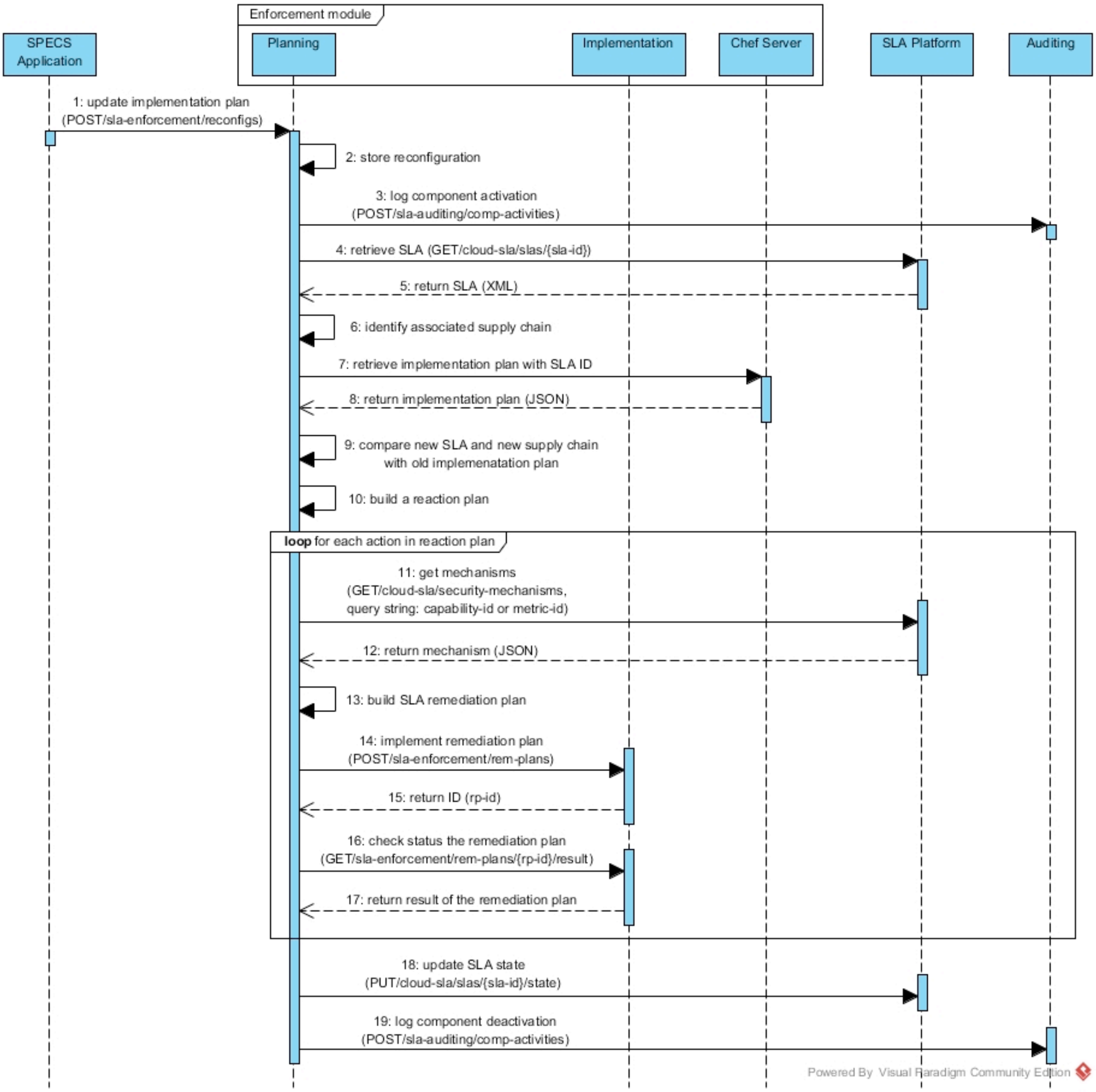
Enforcement module
SPECS Application
Planning
Implementation
Chef Server
SLA Platform
Auditing
1: update implementation plan (POST/sla-enforcement/reconfigs)
2: store reconfiguration
3: log component activation (POST/sla-auditing/comp-activities)
4: retrieve SLA (GET/cloud-sla/slas/{sla-id})
5: return SLA (XML)
6: identify associated supply chain
7: retrieve implementation plan with SLA ID
8: return implementation plan (JSON)
9: compare new SLA and new supply chain with old implemenatation plan
10: build a reaction plan
loop for each action in reaction plan
11: get mechanisms (GET/cloud-sla/security-mechanisms, query string: capability-id or metric-id)
12: return mechanism (JSON)
13: build SLA remediation plan
14: implement remediation plan (POST/sla-enforcement/rem-plans)
15: return ID (rp-id)
16: check status the remediation plan (GET/sla-enforcement/rem-plans/{rp-id}/result)
17: return result of the remediation plan
18: update SLA state (PUT/cloud-sla/slas/{sla-id}/state)
19: log component deactivation (POST/sla-auditing/comp-activities)
Powered By Visual Paradigm Community Edition

Figure 13. Re-negotiation: updating the implementation plan

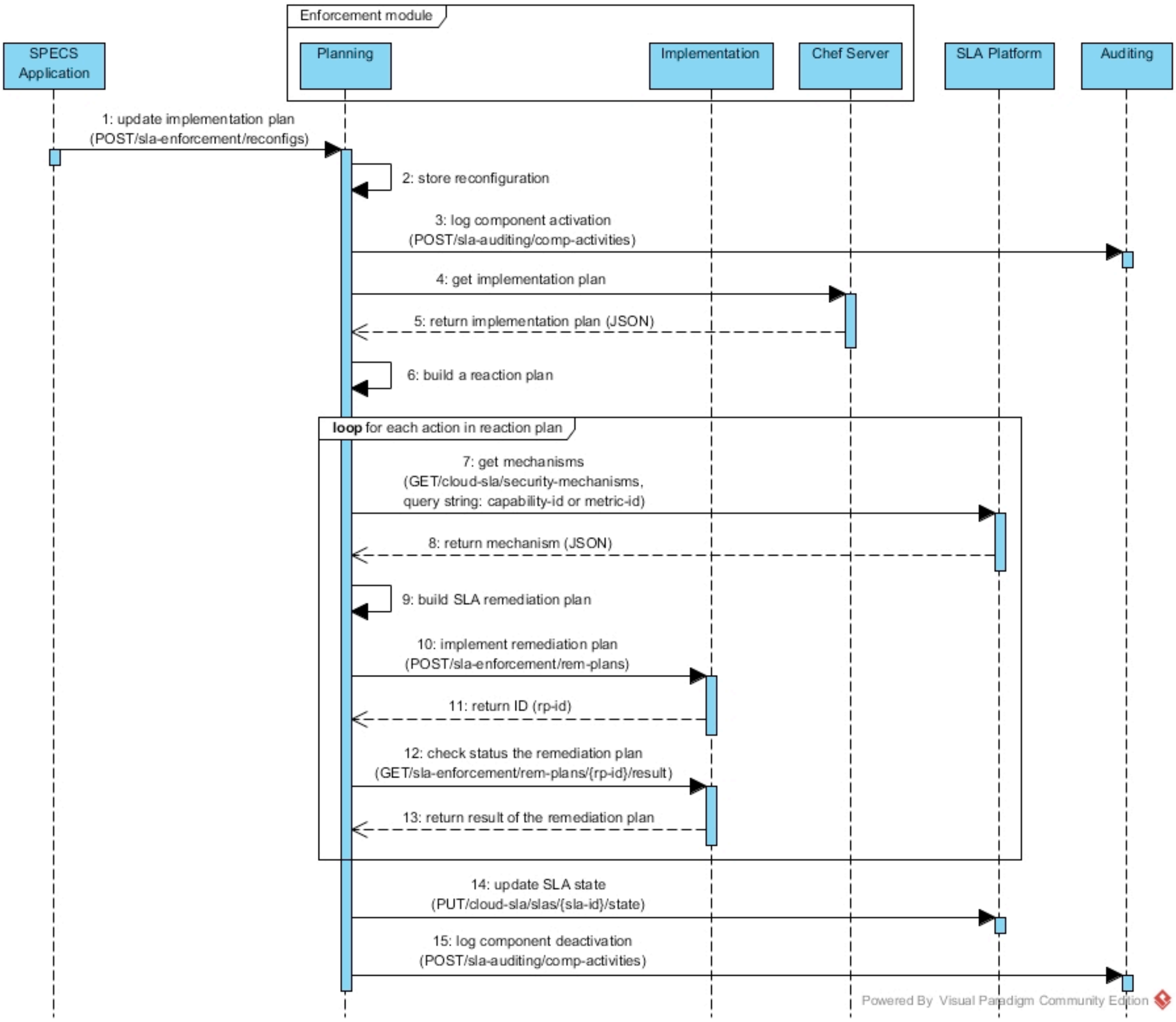
Enforcement module
SPECS Application
Planning
Implementation
Chef Server
SLA Platform
Auditing
1: update implementation plan (POST/sla-enforcement/reconfigs)
2: store reconfiguration
3: log component activation (POST/sla-auditing/comp-activities)
4: get implementation plan
5: return implementation plan (JSON)
6: build a reaction plan
loop for each action in reaction plan
7: get mechanisms (GET/cloud-sla/security-mechanisms, query string: capability-id or metric-id)
8: return mechanism (JSON)
9: build SLA remediation plan
10: implement remediation plan (POST/sla-enforcement/rem-plans)
11: return ID (rp-id)
12: check status the remediation plan (GET/sla-enforcement/rem-plans/{rp-id}/result)
13: return result of the remediation plan
14: update SLA state (PUT/cloud-sla/slas/{sla-id}/state)
15: log component deactivation (POST/sla-auditing/comp-activities)
Powered By Visual Paradigm Community Edition

Figure 14. Termination of an SLA

4. Module APIs

In order to identify the interaction protocols and to define the related detailed APIs, we analysed the framework's design reported in D1.1.2 and identified all modules' interfaces reported in the UML component diagrams drawn for each module in that deliverable. We analysed the exposed interfaces to elicit all resources and API calls needed to enable the communication among the modules according to the identified behaviour and to the requirements elicited in the previous deliverables.

This process resulted in the definition of a set of REST APIs, developed separately by the partners in compliance with precise guidelines, reported in Section 5.

In this section, we provide an overview of the APIs offered by the SPECS core modules and by the SPECS SLA Platform, and needed for their interoperation.

In particular, we illustrate the SLA, Services and Interoperability APIs offered by the Platform and the APIs for negotiation, monitoring and enforcement. Furthermore, the Log APIs and the public API for monitoring belonging to the Cloud Trust Protocol (CTP) are presented.

The complete API documentation is reported in the Annexes to this document.

4.1. SLA API

The SLA API, whose complete documentation is available in the Annex A, is offered by the SLA Manager component of the SLA Platform, and it is aimed at managing all the SLA life-cycle, by providing functionalities for:

1. **Creating new SLAs**. When the negotiation process starts, the SPECS Application retrieves available SLA Templates from the SLO Manager and shows to the End-user available offers. Meantime, a new SLA is created by the SLA Manager for the End-user: the SLA is created in the *pending* state, and will be assigned the *negotiating* state if no errors occur.
2. **Deleting SLAs**. If errors occur during the negotiation process, an SLA that has not been signed yet can be deleted.
3. **Updating existing SLAs.** When an SLA is in the *negotiating* or *re-negotiating* state, it can be updated to reflect ongoing changes. In particular, at the end of the negotiation process, the SLA is updated with the content of the SLA Offer selected by the End-user.
4. **Accessing and retrieving stored SLAs.** Accessing and retrieving stored SLAs is needed during all phases of the SLA life-cycle.
5. **Accessing and updating the state of a stored SLA.** The state of an SLA is updated during the different phases according to the state diagram reported in D1.1.1.
6. **Adding and updating annotations to a stored SLA.** In order to enable an easy processing of SLAs, it may be needed to annotate them with additional information.

4.2. Services API

The Services API, whose complete documentation is available in the Annex B, is offered by the Service Manager and the Metric Catalogue Manager components of the SLA Platform. It offers functionalities for:

1. **Adding new security mechanisms**. At development phase, a developer can create new security mechanisms and add them to the pool of available security mechanisms to offer through the SPECS Enforcement module. Note that security mechanisms include not only the actual mechanisms that enable to offer security capabilities, but also the related monitoring systems, used to monitor metrics associated with such security capabilities. A mechanism is represented, at the Service Manager, in terms of the covered capabilities, the enforced and monitored metrics, and a set of metadata. Metadata includes the list of software components implementing the mechanism, along with related deployment information and constraints.
2. **Updating/deleting a security mechanism.** At development phase, a developer can update a security mechanism by updating the related information stored at the Service Manager. Similarly, the developer can delete an existing mechanism.
3. **Retrieving available security mechanisms.** During the process of supply chains building, the Supply Chain Manager retrieves, from the Service Manager, the set of mechanisms associated with the metrics selected by the End-user during negotiation. The information
4. **Adding/updating/deleting mechanisms' metadata**. At development time, a developer can add/update/delete metadata associated with a security mechanism.
5. **Retrieving mechanisms' metadata**. During the process of supply chains building, metadata of selected mechanisms is retrieved and used to prepare the inputs to the Planning component's *analytical solver.* As mentioned before (see Section 3.1), the solver process is aimed at finding a solution to the planning problem, consisting in identifying the best allocation of needed software components (belonging to mechanisms) to available resources, while respecting deployment constraints set by the mechanisms' developers.
6. **Adding/updating/deleting security capabilities**. At development time, new capabilities can be added to those managed by the SPECS framework. They are expressed in terms of sets of security controls, as defined by NIST in [3]. We support both NIST security controls and CSA's CCM controls.
7. **Retrieving security capabilities**. At development time, existing capabilities are retrieved by the SPECS owner to build SLA Templates.
8. **Adding/updating/deleting security metrics**. At development time, a security expert can add new security metrics to the Metrics Catalogue. At current state, we support the RATAX standard for metrics representation [3]. Security metrics can be updated or deleted.
9. **Retrieving security metrics**. At development time, existing metrics are retrieved by the SPECS owner to build SLA Templates.

4.3. Interoperability API

The Interoperability API, whose complete documentation is available in the Annex C, is offered by the Interoperability layer and offers functionalities for enabling a transparent communication among different modules. In practice, it acts as a gateway by intercepting all REST calls and by redirecting them to the right component. This is accomplished by defining a proper *virtual interface* that associates a set of REST calls to a specific end point (i.e., a concrete URL address).

4.4. Negotiation API

The Negotiation API, whose complete documentation is available in Annex D, is offered by the SLO Manager component of the Negotiation module. It is the only component of the Negotiation module which is accessible to the other modules and components. Its main role is to offer to the SPECS Application an SLA Template representing the available offers, which will be used as a guideline for negotiation and which will be adopted as a basis to construct SLA Offers.

The Negotiation API provides the functionalities for

1. **Retrieving/updating/deleting Service Description Terms.** When negotiation/renegotiation starts, the SPECS Application must offer to the End-user the capabilities, controls and metrics associated to the offered security mechanisms. This information is stored internally in the SDT associated to a SLA Template. It is possible that certain information in the SDT is modified (certain supported controls or metrics change, some capabilities are not offered anymore, etc.), case in which the STD is updated or deleted.
2. **Retrieving/updating/deleting Service Level Objectives.** The SDT does not contain information related to the SLOs related to some metric. Methods related to SLOs allow manipulating SLOs.
3. **Creating new SLA Templates.** In order to set-up the SPECS Application, the SPECS Owner builds one or more SLA Templates, specifying all available capabilities, controls, metrics and SLOs in order to give the End-user the opportunity to construct himself/herself dynamically the SLA Offer during negotiation.
4. **Retrieving/updating/deleting SLA Templates.** The negotiation/renegotiation process is based on a SLA Template containing all the security features that can be offered. If some of the attributes of a SLA Template change (default SLOs for a metric change, some capabilities are not offered anymore) the SLA Template could be updated. In case the SLA Template is not updated but a new SLA Template is created, it might be the case that after some time the old SLA Template is not useful anymore and can be deleted.
5. **Retrieving/updating/deleting SLA Offers.** After the End-user specifies his/her security requirements, their fulfilment is checked by the Supply Chain Manager component and ranked based on their security level by the Security Reasoner, obtaining in the end a list of SLA Offers from which one will be chosen by the end-user to be signed. The remaining SLA Offers are deleted from the SLA Platform, where they are stored.

4.5. Enforcement API

The Enforcement API, whose complete documentation is available in the Annex E, is offered by all core components of the Enforcement module (i.e., Planning, Implementation, Diagnosis, RDS) and offers functionalities for

1. **Creating/retrieving a Supply Chain Activity**. In SLA negotiation phase at least one implementable supply chain for an SLA Template has to be generated. When all security parameters are chosen by the EU, the Negotiation module triggers generation of supply chains. All information related to the generation are stored and maintained by a Supply Chain Activity. This includes available resources, chosen capabilities, SLOs, and IDs of generated supply chains.
2. **Retrieving the status of a Supply Chain Activity**. When the Supply Chain Activity for an SLA Template is generated, its status is set to *Created*. While supply chains are being built, the status is set to *Running*. When all possible supply chains for an SLA template are built, the status is updated to *Completed*.
3. **Retrieving the list of Supply Chains generated for the Supply Chain Activity**. When the status of a Supply Chain Activity is *Completed*, the Negotiation module can retrieve a list of IDs of all associated supply chains.
4. **Retrieving/deleting Supply Chains**. Retrieving supply chains is needed during the SLA negotiation and SLA implementation phases.
5. **Creating/retrieving a Planning Activity**. In order to implement a signed SLA, the implementation plan has to be generated first. After SLA negotiation phase the Planning component is invoked to build an implementation plan according to the signed SLA. In the SLA remediation process some updates to the existing active implementation plan are needed (actually, plans are not updated; for every change a new plan is created which replaces the old one). Similarly, after SLA renegotiation phase, if after renegotiation the original SLA is still valid, but some changes in configuration of services are needed, a new plan is created which replaces the original one. All information needed for the initial generation of the implementation plan, a list of IDs and a number of all generated implementation plans for an SLA, and the ID of the last active implementation plan are stored in a Planning Activity.
6. **Retrieving the status of a Planning Activity**. When the Negotiation module invokes the Planning to generate an implementation plan, a Planning Activity is created with the status *Created*. When the plan generation starts, the status evolves to *Building*. After the plan has been built and its implementation is in progress, the status is updated to *Implementing*. When the plan has been successfully implemented, the status if associated Planning Activity evolves to *Active*.
7. **Retrieving the number of and a list of implementation plans associated to a Planning Activity**. When the status of the Planning Activity is *Active*, the number and the list of IDs of associated implementation plans can be retrieved.
8. **Retrieving the ID of the last active implementation plan associated to a Planning Activity**. When the status of the Planning Activity is *Active*, the ID of the last active associated implementation plan can be retrieved.
9. **Creating/retrieving an Implementation Activity**. In the SLA Implementation process the Planning invokes Implementation component to implement a plan associated to an SLA. All information related to all implementation plans associated to an SLA is stored in an Implementation Activity.
10. **Retrieving the status of an Implementation Activity**. When Implementation component is invoked to implement an initial implementation plan for a signed SLA, an Implementation Activity is created with status *Created*. During the implementation

process the status evolves to *Implementing*. When all services are up and running, the status is updated to *Complete*.

11. **Retrieving an Implementation Plan**. In SLA remediation phase and SLA implementation phase after remediation/renegotiation, access to the existing implementation plans for an SLA is needed. Implementation plans are maintained by the Chef Server.
12. **Adding/retrieving a Notification**. When the Monitoring module detects an event that might result in an alert or a violation, the occurrence is notified to the Diagnosis module. A Notification includes all information about the event and the infrastructure where the event has been detected.
13. **Creating/retrieving a Diagnosis Activity**. When the Monitoring module notifies the Diagnosis about a possible alert/violation, the Diagnosis component creates a Diagnosis Activity which gathers all information about the notified event and the results of the analysis.
14. **Retrieving the status of a Diagnosis Activity**. When a Diagnosis Activity is created, the status is set to *Received*. When the affected SLA has been identified, the status evolves to *SLAidentified*. After the analysis and classification process the status is updated to *Classified*.
15. **Retrieving the ID of the SLA associated to a Diagnosis Activity**. The access to the information about the SLA affected by a detected monitoring event might be needed.
16. **Retrieving a classification of an event associated to a Diagnosis Activity**. The access to the information about the classification of a detected monitoring event might be needed. Each event can be classified as a false positive, an alert or a violation.
17. **Creating/retrieving a Remediation Activity**. After a monitoring event has been classified as an alert or a violation, the RDS component has to prepare and implement a remediation plan. All information about the remediation process associated to an event/SLA is stored in Remediation Activity and maintained by the RDS component.
18. **Retrieving the status of a Remediation Activity**. Each Remediation Activity is created with status *Created*. During the remediation process the status evolves to *Remediating*. After remediation process is complete the status is updated to *Complete*.
19. **Creating/retrieving a Remediation Plan**. When the RDS prepares a remediation plan to mitigate the risk of having a violation or recovering from a violation, it is sent to the Implementation component which has to implement it.
20. **Retrieving the result of a Remediation Plan**. When the Implementation component receives a Remediation plan, the SLA remediation process starts. The process can either end with a successful mitigation/recovery (in this case the result is *Observe*) or we run out of possible actions to remediate the alert/violation. In this case the result is *Notify* which implies that the EU has to be notified about the occurrence and the consequences.
21. **Adding/retrieving a Reconfiguration**. In order to reconfigure running services after SLA renegotiation, a reconfiguration is triggered by the SPECS Application. In this case Reconfiguration (which consist of an SLA ID and a label) is labelled by *Reconfigure*. If an EU requests termination of an SLA before its expiration date (or decides to terminate an SLA after an alert/violation), the Reconfiguration is labelled by *Terminate*. In both cases the Planning component prepares a plan to reconfigure/terminate services and passes it to the Implementation component.

4.6. Log API

The Log API, whose complete documentation is available in the Annex F, is offered by the Auditing component (Enforcement module) and offers functionalities for

1. **Logging activations/deactivations of components**. After the signature of an SLA each activation/deactivation of components due to implementation or remediation activities for a signed SLA is logged for auditing purposes.
2. **Logging activation/deactivation of services**. In SLA implementation phase (either after SLA signature or SLA remediation/renegotiation) all activated/deactivated services are logged.
3. **Logging diagnosed monitoring events**. For each detected alert or violation, all the information gathered in the diagnosis process is logged. This includes affected SLOs and associated metrics, infrastructure on which the event occurred and the root cause, and the impact of the event on the affected SLA.
4. **Logging remediation results**. After the SLA remediation process the results of actions performed to mitigate the risk of a violation or to recover from a violation are logged.
5. **Logging notifications sent to the EU**. Each alert and violation that cannot be resolved automatically is notified to the EU. In this case all information related to the detected event and performed remediation actions are logged.

4.7. Monitoring API

The monitoring API, whose complete documentation is available in the Annex G, is offered by a set of components that ensure the entire life cycle of the monitoring flow where specialized components, called monitoring adapters, are sending monitoring raw data to a centralized communication router, called the event hub. The event hub will distribute the monitoring data to different monitoring core components (event aggregator, event archiver and monitoring policy filter) that are able to process the information and act according to the monitoring flow. The Monitoring API offers functionalities for:

1. **Collecting, publishing and routing the monitoring data**. Raw monitoring data is collected from the target services and sent to a monitoring router (the event hub), which publish the raw data for other specialized SPECS components to consume it. The processed raw monitoring data, called monitoring events, are also resent through the router to the designated consumers.
2. **Archiving the monitoring data and events.** All the monitoring information (raw data or events) are sent automatically the to event archiver for storing the data for later use.
3. **Evaluation, aggregation and decision-making.** The monitored data is sent to a specialized filter that, based on some customizable monitoring rules, is able to identify the event and to evaluate it. Multiple similar events are aggregated within a time frame based on aggregation rules. If the aggregation value breaks a threshold the decision making mechanism will trigger the corresponding component about the abnormal situation that needs to be diagnosticated (the case of a possible alert or violation).
4. **Format conversion.** In case of triggers the monitoring event format it translated into a metric compatible format and sent to the corresponding component for diagnosis.

4.8. Monitoring Public API

The SPECS platform will offer a public API to enable cloud customers to monitor the current security level of their cloud service. Customers will be able to get information about the current measurement of the security attributes of their cloud services, based on the same attributes that appear in the SLOs that are specified in the SLAs negotiated with the SPECS platform. To take an analogy with a thermometer, the monitoring service will allow the customer to consult the current temperature measured by the device, independently of the fact that the temperature may be considered good or bad (i.e. whether the temperature is within the margins defined in the SLO or not).

The API that customers will use to access monitoring data is CSA's Cloud Trust Protocol. This API is fully specified in an external document [4] and is therefore not repeated here in contrast with other API described previously. We call this the "CTP Public API".

The "CTP Public API" will be implemented by a CTP server that will act as an interface between the SPECS platform and the customer. In order to be able to present information to the customer with the "CTP Public API", the information produced by the SPECS platform must be translated and imported into the CTP server. In order to be able to import data in the CTP server, a new additional API has been developed for CTP in the context of SPECS: the CTP back office API. Whereas the "CTP Public API" enables customers to *read* information from a CTP server, the "CTP back office API" enables authorized users to *write* information to a CTP server. As such, the "CTP back office API" allows not only to update measurement results related to a security attribute but also to populate the CTP Server with descriptions of services, metrics, assets, attributes and measurements.

The following figure summarizes the positioning of each API in the context of providing a monitoring API to the customer.

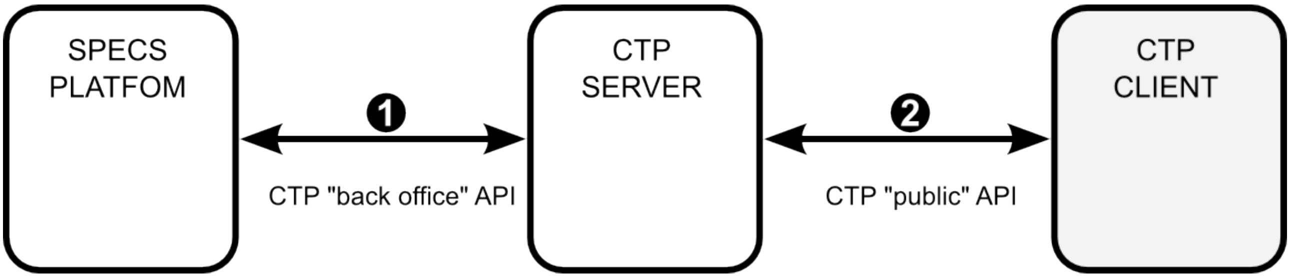

Figure 15. The "CTP Public API" versus the "CTP back office API".

While the "CTP back office API" will be detailed in Deliverable D3.4, we provide a high level view of the integration functionalities that will be implemented in order to provide a monitoring API to cloud customers.

1. **Adding customers:** When a new customer is added to the SPECS platform, it must contact the CTP server to execute the following actions:
   a) Create a "view" in CTP.
   b) Create an "access token" that is authorized to access the view in step (1.a).
   c) Return the created "key" from the access token: the customer can then use this key to authenticate with the CTP server.

2. **Adding an SLA for a customer**: SLAs negotiated in SPECS follow an extension of the WS-Agreement format (WSLA). These WSLAs should[1] be mapped to the CTP data model as follows:
   a) A WSLA metric is mapped to CTP metric.
   b) A WSLA Service is mapped to a CTP "service-unit".
   c) A default "asset" is created within the service-unit in CTP. CTP takes a more granular approach than SPECS and allows to define SLO and measurements that do not apply to a service as a whole but only to an asset within the service, so in SPECS we need to create a dummy asset that is equivalent to the service as a whole.
   d) WSLA variables that are defined within WSLA service properties are mapped to CTP attributes. These attributes are attached to the asset created in step (2.c).
   e) WSLA SLO are translated into measurements, where
      i) The SLO "metric-name" becomes the CTP measurement name.
      ii) The SLO "expression" becomes a measurement objective.
      iii) The create measurement is attached to the attribute created in step (2.d).

3. **Updating the measurements of security attributes**: When new relevant monitoring data is available from the SPECS event hub, it is used to update the CTP measurement results.
   a) Identifiers contained within the event data are used to identify the relevant corresponding measurement that needs to be updated in CTP.
   b) The measurement results are then updated in CTP.

4. **Deleting customers and/or SLAs**: When an SLA or a customer is removed from SPECS, it must be reflected in the CTP server:
   a) Removing an SLA means removing the corresponding "service unit" created in step (2.b). All other dependent resources are automatically deleted by CTP (e.g. assets, attributes, etc.).
   b) Removing a customer means removing a view and a token (those created in step .

[1] Based on the current structure of SLAs in SPECS and subject to future changes and clarifications.

5. API definition guidelines

In this section, we report the guidelines followed when developing SPECS' REST APIs. In particular, we discuss the conventions we adopted related to the use of HTTP response codes, the different mediatypes our APIs support, and the way resources are identified.

REST is *an architecture style* for designing networked applications that relies on a stateless, client-server, cacheable communication protocol. In most cases, REST is based on HTTP (i.e., HTTP methods are used to retrieve and send web content from/to remote servers) even if it is not bound to any protocol in particular. Recently, REST has gained widespread acceptance across the World Wide Web as a simpler alternative to SOAP and WSDL-based web services.

In his "Maturity Model" [1], Richardson classifies the APIs for services on the web (i.e., for software services built on top of the HTTP protocol) in three incremental maturity levels, according to the support offered for URIs, HTTP methods and for hypermedia respectively. The levels are defined as follows:

- **Level 0:** no support, there is only one endpoint, namely RPC over HTTP (WS-*, XML-RPC, ...);
- **Level 1:** support for resources (multiple endpoints), referenced through URIs;
- **Level 2:** support for HTTP methods: requests to endpoints shall follow the HTTP rules/semantic, i.e. they shall use standard methods (GET,POST,PUT,DELETE...);
- **Level 3:** support for RESTFul APIs, being self-descriptive in that a client can explore them by only knowing the basic interactions and how to interpret the responses.

According to Martin Fowler [2], Level 1 tackles the question of handling complexity by using divide and conquer, breaking a large service endpoint down into multiple resources. Level 2 introduces a standard set of verbs so that we handle similar situations in the same way, removing unnecessary variation. Level 3 introduces discoverability, providing a way of making a protocol more self-documenting.

SPECS' APIs sit at Level 2, but the support Level 3 is already planned in future versions.

5.1. Response Code Guidelines

In order to ensure a homogenous behaviour, some basic conventions about the use of response codes for all SPECS APIs are defined. The conventions are the following:

- When not specified differently, the response code for successful requests is always "200 OK", while the response entity body shall be specified into the API documentation.
- The 404 and 400 error codes are specified for all calls.
- If not otherwise specified in the API documentation, the response body in case of a client (4xx) or server (5xx) error code is empty.

The following response codes have fixed and shared semantics. They shall be supported by all implementations.

Response Code	Meaning
200 OK	The request has succeeded.
400 Bad Request	The request issued by client is malformed. This may imply a syntax error in the request headers or in the request entity body, or it may be due to a request entity body that contains a different mediatype compared to the one declared into "Content-Type" header.
404 Not Found	Resource not found.
405 Method Not Allowed	The server is refusing to service the request because the method is not applicable for the requested resource.
406 Not Acceptable	The resource identified by the request cannot be represented by the mediatype specified into the "Accept" header by the client.
415 Unsupported Media Type	The server is refusing to service the request because the mediatype of the request entity body, specified into "Content-Type" header, is not accepted for the requested pair resource/method.
500 Internal Server Error	A server-side error that prevents from fulfilling the request occurred. The client may retry the request later, with no guarantees that it will succeed.
503 Service Unavailable	A server-side temporary error occurred. Implementation should set a "Retry-After" header, while clients can retry the request after the specified time interval has elapsed.

Table 1. HTTP shared error codes

5.2. Mediatype support

The client should negotiate resources' representation, where applicable. The admissible media types are the following:

- text/plain (mostly for URL, simple strings, etc.)
- application/json (default)
- application/xml (optional)

The expected media type of both request and response body of an API call must be specified in the API documentation. Moreover, if a resource supports multiple representations, this must be declared in the API documentation, too. The JSON mediatype is used by default for the communication among internal components, while an XML description for those resources that can be accessed directly by users is also provided.

In case an application/json or an application/xml media type is involved, the corresponding data model (namely, the related JSON or XML schema) must be specified in the API documentation.
All data models are publicly available in a web repository, and the links to related schemas are reported in the bibliography section.

5.3. Resource Identification (URI)

Resource URIs must be used in a consistent way:

- Each URI identifies only one resource;
- A resource **can** be identified by more than one URI.
  For instance, “GET /triggers/” identifies all the triggers defined into system, while “/resources/r_X/triggers” identifies all triggers for resource “r_X”.
- A “query string” can be specified (mostly with GET methods) in order to filter results of an API call.

The APIs documentation shall follow the subsequent conventions:

- each resource is identified by an URI composed of a base-path and a resource identifier (e.g., /cloud-sla/slas identifies a resource, belonging to the cloud-sla base-path, representing the set of stored SLAs),
- parts of URLs surrounded by curly braces identify a variable string (e.g., /cloud-sla/slas/{sla-id} identifies the SLA with id sla-id).

5.4. Collections

Resources can be grouped into collections. Each collection contains an unordered set of resources belonging to one specific type and identified by means of their URIs. Collections objects are supported in XML/JSON formats and specify the following additional set of information:

Field	Meaning
resource	The type of resources included in the collection
total	The total number of items in the collection that are hosted on the system
members	The number of items of the collection involved in the call

2

In Table 2, we report the XML and JSON data models considered for collections, available at *[xml_1] and [json_1]* respectively.

Format	Data model

XML	`<?xml version="1.0" encoding="UTF-8"?>`
	
	`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >`
	
	`<xs:element name="collection" type="collectionType"/>`
	
	`<xs:complexType name="collectionType">`
	` <xs:sequence>`
	`  <xs:element name="item" minOccurs="0" maxOccurs="unbounded">`
	`   <xs:complexType>`
	`    <xs:simpleContent>`
	`     <xs:extension base="xs:anyURI">`
	`      <xs:attribute name="id" type="xs:string" use="optional"/>`
	`     </xs:extension>`
	`    </xs:simpleContent>`
	`   </xs:complexType>`
	`  </xs:element>`
	` </xs:sequence>`
	` <xs:attribute name="resource" type="xs:string" use="required"/>`
	` <xs:attribute name="total" type="xs:integer" use="optional"/>`
	` <xs:attribute name="members" type="xs:integer" use="optional"/>`

	</xs:complexType> </xs:schema>

JSON	{
	"type":"object",
	"properties":{
	"resource": {"type":"string"},
	"total":{"type":"string"},
	"members":{"type":"integer"},
	"item_list": {
	"type": "array",
	"items": {
	"type":"object",
	"properties":{
	"id":{"type":"string"},
	"item":{"type":"string"}
	}
	}
	}
	}
	}

Table 2. Collections data models

An example of instantiation of such models is as follows:

```
Media type: application/xml
<collection resource="annotation" total="100“ members=“10”>
   <item id="0">http://localhost/slas/67/annotations/1</item>
…
   <item id="9"> http://localhost/slas/67/annotations/87</item>
</collection>

Media type: application/json
{
  “resource”: ”annotation”,
  “total”:”100”,
  “members”=“10”,
  “item_list”:[
       {“id”:”0”,“item”:“http://localhost/slas/67/annotations/1”},
        “id”:”9”, “item”:“http://localhost/slas/67/annotations/87”}
    ]
}
```

The request URI for collections can include a query string, in order to limit the scope of the collection itself. An example is given in the following table.

URI	Interpretation
<base URL>/collection_resource?items=<total_items>	Returns a collection with items=<total_items>
<base URL>/ collection_resource?page=<value>&length=<value>	Returns a collection with at most <length> items where the page parameter specifies the first item to retrieve.

The well formed request for a collection always returns a HTTP 200 code. The collection object may be empty (i.e., “items=0”), in case of:

- an empty collection;
- a query string that results in an out of range error.

In case of a syntactically malformed query strings, a HTTP 400 code will be returned.

6. Conclusions

In this deliverable, the interaction protocols among the SPECS main modules have been presented. In particular, we analysed more in details the behaviour of the main SPECS modules during the phases of the SLA life cycle and elicited the functionalities needed to accomplish related tasks. Such functionalities were used to design the REST APIs exposed by each module, whose documentation has been reported in separated Annexes to this document. The REST APIs have been developed by following precise guidelines, which have been illustrated in the deliverable, too. Moreover, the API documentation includes the specification of all data models used by REST calls.

7. References

[1] J. Webber, S. Parastatidis, and I. Robinson, REST in Practice, Hypermedia and Systems Architecture. O'REILLY, 2010.
[2] M. Fowler, "Richardson maturity model: steps toward the glory of REST," 2010. [Online]. Available: http://martinfowler.com/articles/richardsonMaturityModel.html
[3] NIST, "NIST Special Publication 500-307 Draft: Cloud Computing Service Metrics Description," 2015.
[4]Cloud Security Alliance, "Cloud Trust Protocol", draft version 3.1, A. Pannetrat editor, July 2015.
[xml_1] http://www.specs-project.eu/resources/schemas/xml/collections.xsd
[json_1] http://www.specs-project.eu/resources/schemas/json/collections.json

Annex A – SLA API

Annex B – Services API

Annex C – Interoperability API

Annex D – Negotiation API

Annex E – Enforcement API

Annex F – Log API

Annex G – Monitoring API