D2.3

Proof-of-Concept Prototype of Secure Computation Infrastructure and SUPERCLOUD Security Services

Project number:	643964
Project acronym:	**SUPERCLOUD**
Project title:	User-centric management of security and dependability in clouds of clouds
Project Start Date:	1st February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Demonstrator
Reference Number:	ICT-643964-D2.3 / 1.0
Work Package:	WP2
Due Date:	June 2017 - M29
Actual Submission Date:	4th July, 2017
Responsible Organisation:	ORANGE
Editor:	Marc Lacoste
Dissemination Level:	PU
Revision:	1.0
Abstract:	This deliverable describes the software components that form the secure virtualization infrastructure and security services for computation. We give an overview of the structure of the security framework for computation, present the APIs of its main components, and provide information on how to access and use the software developed.
Keywords:	computation, virtualization, security self-management, hypervisor, orchestration, isolation, trust management, FPGA, authorization, monitoring, replication, SLA management

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 643964.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

Editor

Marc Lacoste (ORANGE)

Contributors (ordered according to beneficiary numbers)

Mario Münzer, Felix Stornig (TEC)
Marc Lacoste, Alex Palesandro, Denis Bourge, Charles Henrotte, Houssem Kanzari (ORANGE)
Marko Vukolić, Jagath Weerasinghe (IBM)
Reda Yaich, Nora Cuppens, Frédéric Cuppens (IMT)
Markus Miettinen, Ferdinand Brasser, Tommasso Frassetto (TUDA)
Daniel Pletea (PEN)

Disclaimer

Executive Summary

This deliverable presents the software components of the SUPERCLOUD distributed cloud infrastructure and security services for computation. We start by giving a general overview of the structure of the overall security framework for computation. We then present the software components of the framework organized into two separate infrastructures:

- a core virtualization infrastructure including components for virtualization and orchestration, system support for cross-layer security, hardware-based isolation and trust management using Intel SGX technology, and support for cloud FPGAs;
- and a self-management infrastructure including a security orchestrator and several security services covering authorization, security monitoring, geolocation-aware replication, SLA management, and software trust management.

The set of components should be viewed as a security toolbox to build customized U-Clouds hosting secure computations, selecting the needed security services to match user protection requirements. For each component, we describe its main APIs. We also provide information on how to access and use the developed software.

Contents

List of Figures

List of Tables

Chapter 1 Overview

1.1 Context

Previous steps of the project were focused on the design of the SUPERCLOUD computation infrastructure and on prototyping of a number of its components.

- Step 1 concerned **Architecture**. The design of the virtualization and self-management of VM security infrastructure was defined, and is reported in Deliverable D2.1 [19]. This included a *distributed virtualization infrastructure* providing virtualization, isolation, and trust management features for running U-Clouds, and a *self-management infrastructure* for automatically managing computing security with features including policy and SLA management, monitoring, authorization, and configuration compliance checking.

- Step 2 concerned **Implementation**. A number of Proof-of-Concept prototypes were developed, illustrating basic functionality of one or several components of the SUPERCLOUD computing framework. Results are reported in deliverable D2.2 [22].

  Prototypes were focused on: (1) *virtualization and orchestration* primitives for user-centric and cross-provider virtualization combining flexible security control and interoperability, with extension to cross-layer U-Clouds; (2) *isolation and trust* illustrating the use of Intel SGX as key VM isolation technology, trust management between enclaves, advances on trusted execution for large-size applications, and benefits of systems such as Cloud FPGAs (Field Programmable Gate Arrays) for hardware acceleration of security management primitives; and (3) *self-management* through prototypes for multi-cloud security policy modelling and enforcement with applications to network availability and negotiation of SLAs and geolocation-aware data replication.

1.2 Objective of the document

Step 3 is devoted to **Integration**. This deliverable describes the prototypical implementation of the distributed cloud infrastructure for computation (*computing security framework*) and related components for SUPERCLOUD security management, integrating the previous components. This deliverable is a *software* deliverable. The purpose of this document is thus to describe the structure of the computing framework and the APIs of its main components – their detailed design having been described in Deliverables D2.1 [19] and D2.2 [22].

1.3 Outline of the document

The rest of this document is organized as follows. In Chapter 2, we present the approach for specifying the framework and give an overview of its structure, that comprises two separate sub-infrastructures, the *virtualization infrastructure* and the *self-management infrastructure*. Chapters 3 and 4 then present in turn the two sub-infrastructures, in terms of general structure and APIs of their different components. Finally, we conclude in Chapter 5.

Chapter 2 The computing security framework

This Chapter introduces the computing security framework, in terms of design principles (Section 2.1), framework specification approach (Section 2.2), and high-level overview (Section 2.3).

2.1 Framework design principles

The purpose of the SUPERCLOUD computing security framework is to enable secure computing over multiple clouds. This means both providing a secure distributed computation infrastructure and automated protection of its VMs.
The computing security framework has three main objectives:

1. **Virtualization:** it shall reconcile security and interoperability within infrastructure layers and across cloud providers.
2. **Isolation and trust:** it shall provide primitives for secure multi-provider enclaves, and to manage platform integrity and trust.
3. **Security self-management:** it shall fight excessive complexities of protecting infrastructure and VMs through security automation across layers and providers, providing configuration compliance guarantees. It shall also enable user control over security to enforce security SLAs negotiated between user and provider.

These objectives are met through a two-level design:

- **Distributed virtualization infrastructure:** this sub-infrastructure tackles the first two objectives. A hybrid virtualization architecture combining nested virtualization, micro-hypervisor, and modular hypervisor system designs enables to reach interesting interoperability vs. security trade-offs.
- **A self-management of security infrastructure:** this sub-infrastructure tackles the third objective. It provides a unified policy model and management framework featuring: an expressive policy language with both cross-layer and cross-provider monitoring and enforcement, arbitration and SLA negotiation, and orchestration of security services.

The architectures of those infrastructures were described in Deliverable D2.1 [19]. The previous objectives were also explored through different prototypes in Deliverable D2.2 [22].

2.2 Framework specification approach

2.2.1 A component-based design

The SUPERCLOUD computing security framework is specified as a set of *components* that may be assembled depending on user requirements to build customized U-Clouds with needed security services. The *component-based approach* has largely proven its many benefits to design reconfigurable, adaptable, and customizable systems and infrastructures [32]. For instance, a component provides a simple

abstraction for encapsulating behaviors through *interfaces*. A component model also clearly identifies *system dependencies* through provided and required interfaces. It helps to *separate concerns* by distinguishing functional and non-functional interfaces for easier orchestration. Finally, *new features* may be included or enhanced to the system or infrastructure by adding or replacing components with other components. Recent evolutions of this paradigm notably include *micro-service architectures* for cloud and virtualized network infrastructures.
This design approach enables to build a SUPERCLOUD *service catalogue* that customers may use to build their own U-Clouds, picking the just-needed security services from the SUPERCLOUD framework to match security requirements of relevant cloud verticals (healthcare, cloud brokerage, etc.).
To specify the framework, a component-model and specification language is needed to describe in a uniform way components, interfaces, and APIs. We chose TOSCA [24] for this purpose, as this modeling language is one of the most widely used in the cloud domain for different types of applications. We give a short overview of its principles next.

2.2.2 TOSCA overview

Current cloud technologies suffer from a great heterogeneity in the description and management of cloud resources. Thus, OASIS released TOSCA (Topology and Orchestration Specification for Cloud Applications) [24, 25] that provides an XML-based modeling language aiming at portability of application description and management, automated application deployment and management, and interoperability of application components [9].
TOSCA represents applications in terms of *topology* and *life-cycle*. The structure of the application is captured by a typed topology graph that describes the components of the application (*nodes*) and the interaction between those components (*edges*). The application is also described through a set of *plans* capturing deployment and management tasks. The whole description is based on *templates*, or equivalent classes of nodes, relations, plans, etc. The connected set of template instances forms the concrete application. The key concepts of the TOSCA meta-model are summarized in Figure 2.1.

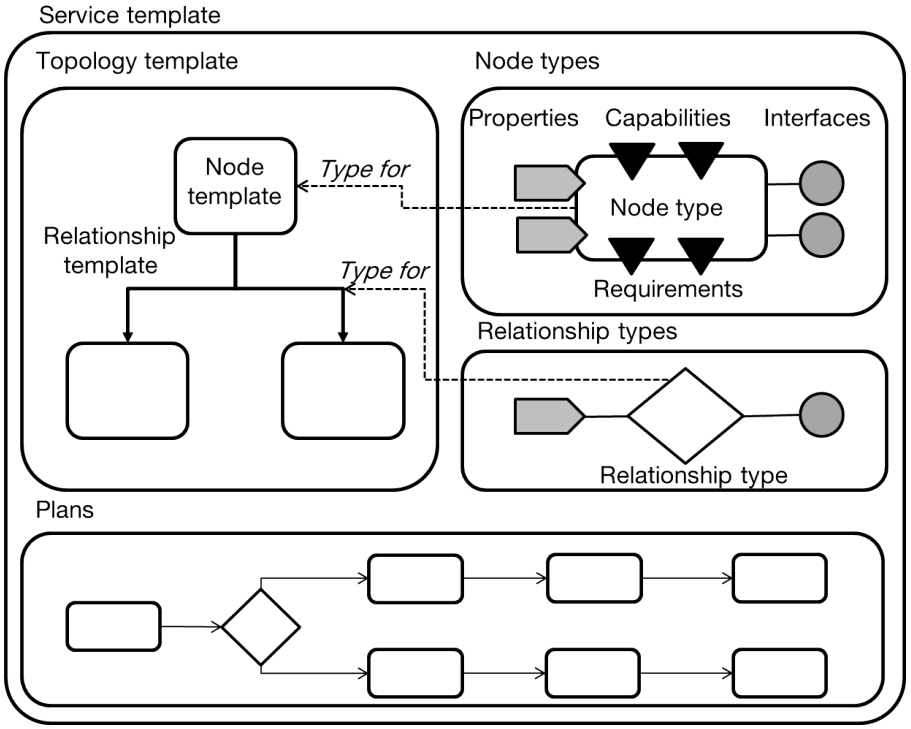

Figure 2.1: The TOSCA meta-model: a simplified view (adapted from [7, 9])

The application is modeled by a TOSCA `ServiceTemplate`, mainly composed of:

- a `TopologyTemplate` is a directed graph defining the application components (`NodeTemplates`) and their relations (`RelationshipTemplates`);
- `NodeTypes` define the structure of nodes of the graph [1];
- `RelationshipTypes` define the structure of edges of the graph, to capture custom links between application components.

`Plans` may also be included in the `ServiceTemplate` to describe as workflows how node operations should be executed to manage the component lifecycle [7].
An application component (TOSCA node) is described by a `NodeType` through the following attributes:

- observable *properties* (`PropertiesDefinitions`);
- *interfaces* and *operations* for managing the component (`Interface`, `Operation`);
- *capabilities* the component can provide to other components (`CapabilityTypes`);
- *requirements* needed for the component to run correctly (`RequirementTypes`).

An interface notably captures how to modify the resource dynamically at run-time through a number of operations, e.g., to manage the component life-cycle such as calling operations through an external orchestration engine.

2.3 Framework high-level overview

2.3.1 Relation to overall SUPERCLOUD framework

The computing security framework scope is a subset of the overall SUPERCLOUD framework, focusing on the protection of computing elements (VMs, containers...) and on their self-management. The framework structure and components are shown in increasing levels of detail in Figures 2.2 and 2.3. More details on the overall SUPERCLOUD framework may be found in [18] and in Deliverables D1.1 [21] and D1.2 [37].

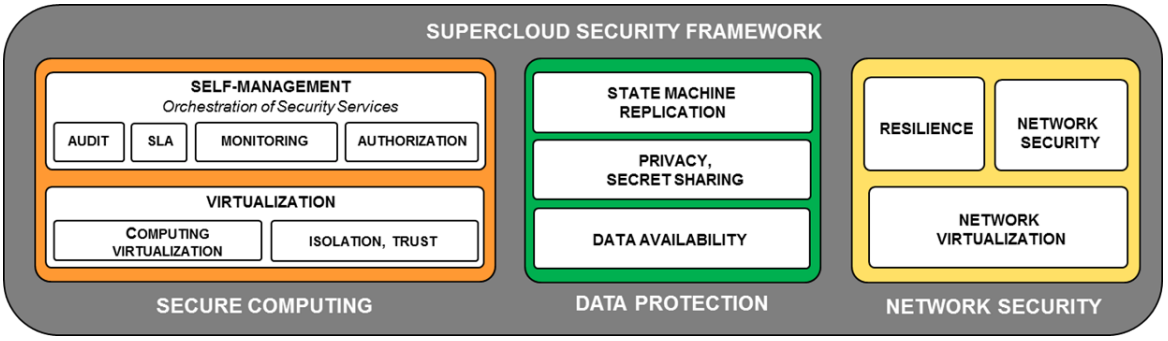

Figure 2.2: SUPERCLOUD framework: high-level overview

[1] In TOSCA, reuse is facilitated by both `NodeTypes` and `NodeTemplates` concepts. A `NodeType` enables to specify generic classes of components, such as their common properties and interfaces. A `NodeTemplate` describes an instance of a `NodeType`, as well as specific properties of the instance, in addition to those already described by the `NodeType`.

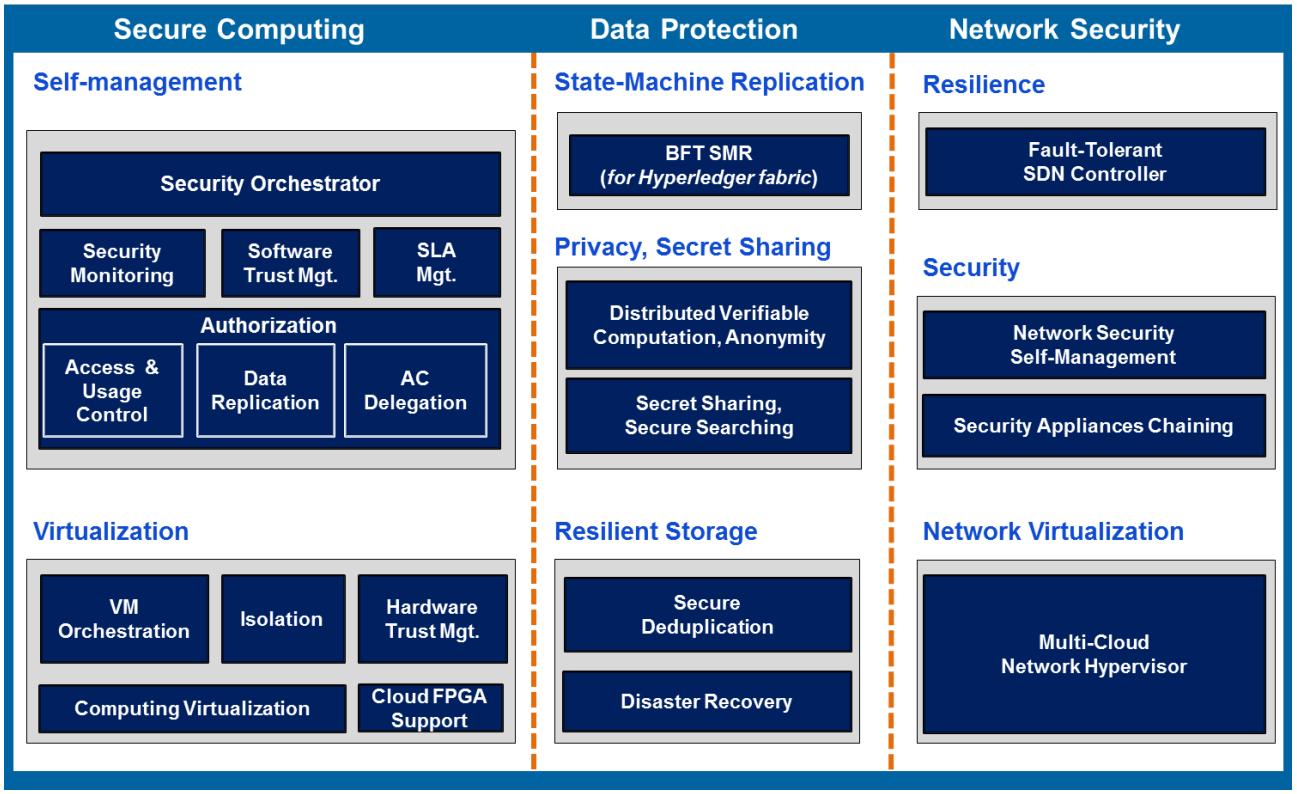

Figure 2.3: SUPERCLOUD framework: computing, data protection, and networking services

2.3.2 Framework structure

As shown in Figure 2.4, the computing security framework has a two-level structure:

- **The virtualization infrastructure** provides a distributed abstraction layer for computing resources spanning multiple cloud providers.
- **The self-management infrastructure** implements autonomic security management for the distributed cloud.

The corresponding framework components are shown in Figure 2.4.

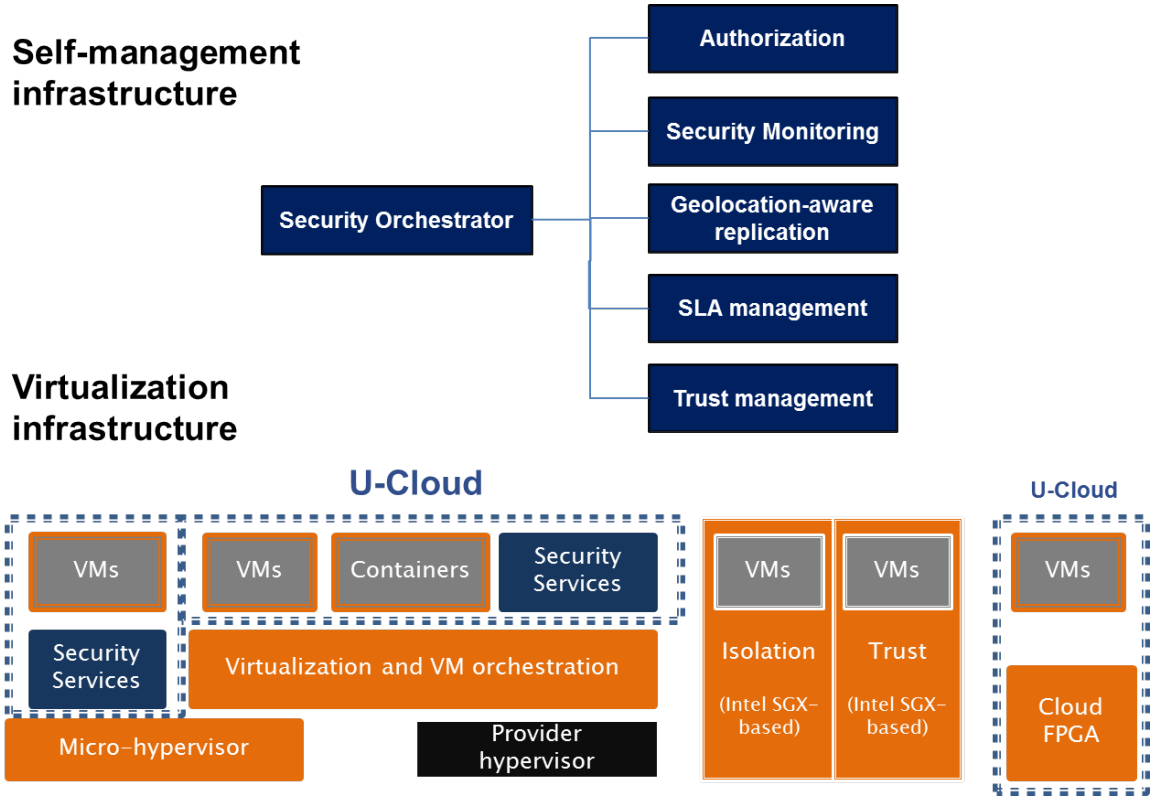

Figure 2.4: SUPERCLOUD computing security framework

2.3.2.1 Virtualization infrastructure

This infrastructure provides distributed virtualization primitives for running U-Clouds on computing resources of different cloud providers. It includes low-level infrastructure security services for: horizontal orchestration to achieve unified security management of computing elements regardless of providers; vertical orchestration, extending user-control over U-Cloud security into the lower layers of the provider infrastructure using a micro-hypervisor; and strong isolation and trust management between computing elements, relying on hardware security mechanisms (e.g., Intel SGX, FPGAs). This infrastructure will be described in more detail in Chapter 3.

2.3.2.2 Self-management infrastructure

This infrastructure provides a number of security services that may be orchestrated on-demand to guarantee (self-) protection of U-Clouds on top of the distributed virtualization infrastructure. Provided services include: authorization to perform resource access control at different infrastructure levels; security monitoring across infrastructure layers and providers; geolocation-aware data replication; management of security SLAs; and software-level trust management between users or providers. This infrastructure will be described in more detail in Chapter 4.

Chapter 3 Virtualization infrastructure

In this Chapter, we describe the virtualization infrastructure in terms of structure and components. We start by providing a general overview of the infrastructure (Section 3.1). We then present the different components grouped according to their main security goals, namely virtualization and orchestration (Section 3.2) and isolation and trust management (Section 3.3). A specific section describes the Cloud FPGA component (Section 3.4) and its benefits in terms of security and performance. For each component, we give a short overview of its functionality and design. We also describe the external component interface, either at high-level using the TOSCA specification language, or through more detailed API specifications. We also include information about how to obtain and use each component.

3.1 Infrastructure overview

The structure and components of the virtualization infrastructure are shown in Figure 3.1.

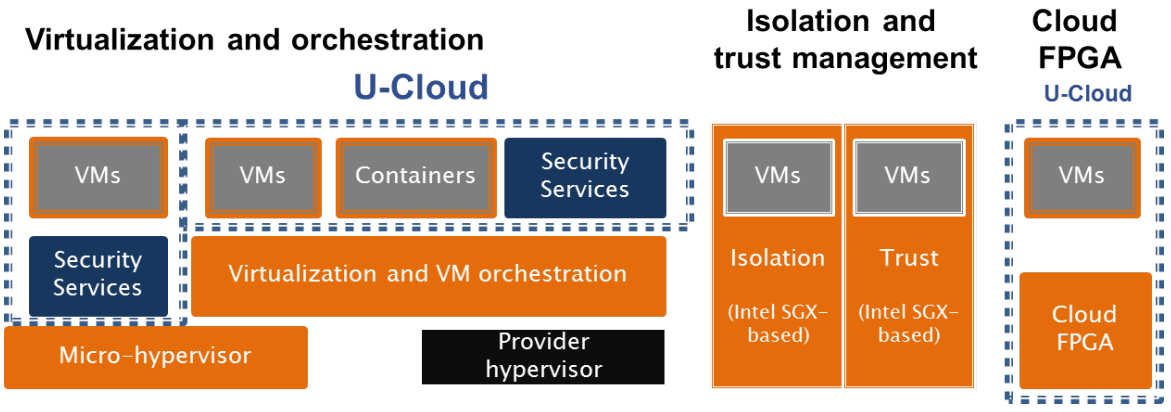

Figure 3.1: The virtualization infrastructure

Virtualization and orchestration. These components implement a distributed virtualization infrastructure for running U-Clouds on computing resources across different providers and extending user-control over security into low-level infrastructure layers. *Horizontal orchestration* achieves secure interoperability of computing elements regardless of providers. *Vertical orchestration* composes security mechanisms across infrastructure layers.

These dimensions are realized respectively by two components:

- A *virtualization and orchestration component:* This horizontal component provides a single point of control for managing U-Cloud security across providers and for composing different security services in a U-Cloud. This component is described in Section 3.2.1.
- A *micro-hypervisor:* This vertical component relies on a minimal and modular hypervisor to run U-Clouds on a single-provider with cross-layer control over security. This component is described in Section 3.2.2.

Isolation and trust management. These components provide strong isolation and trust management primitives for U-Clouds. Virtualization abstractions such as VMs, containers, or processes using the notion of *enclave* as trusted execution environment are notably considered.

- An *isolation component:* Leveraging Intel SGX hardware security technology, this component provides strongly isolated execution environments to guarantee security of applications despite an untrusted virtualization infrastructure. This component is described in Section 3.3.1.
- A *trust management component:* Building on the same hardware technology, this component allows to build and verify distributed *Chains of Trust* to provide evidence of trustworthiness of secure execution in a multi-cloud environment. This component is described in Section 3.3.2.

Cloud FPGA support. This component extends the SUPERCLOUD framework to the device side by adding the FPGA as hardware computing abstraction similarly to software VMs or containers. The *Cloud FPGA component* explores the benefits of FPGAs for accelerating compute workload processing or security management tasks. This component is described in Section 3.4.

3.2 Virtualization and orchestration

3.2.1 Virtualization and orchestration component

This component provides *horizontal* features for virtualization and orchestration. It enables: (1) *instantiation* and *deployment* of a distributed multi-cloud; and (2) selective *weaving* of security services in different parts of the architecture to build U-Clouds spanning multiple providers.

3.2.1.1 Objective

Despite mature cloud technologies, customers still struggle to cross Cloud Service Provider (CSP) barriers to benefit from fine-grained resource distribution, business independence from the provider, and cost savings. Most adopted IaaS inter-cloud solutions remain generally limited to specific public cloud providers or present maintainability issues. Remaining hurdles include complexity of management and operations of such infrastructures, in presence of per-customer customizations and provider configurations.
The *Infrastructure as Code (IaC)* paradigm[1] is emerging as key enabler for IaaS multi-clouds, to develop and manage infrastructure configurations. However, this approach is yet far from being fully practical due to the complexity of the infrastructure life-cycle, to the heterogeneity of composing resources and to user customizations.
To go one step beyond, we designed Orbits (ORchestration for Beyond InTer-cloud Security), a IaC-based overlay inter-cloud orchestration and virtualization framework providing simultaneously flexible application provisioning across multiple providers with a homogeneous service abstraction enforced at IaaS level. Orbits enables thus to instantiate distributed U-Clouds [28].
IaC-based deployment and management of a generic IaaS multi-cloud also requires the ability to flexibly inject or remove non-functional services, such as for security or reliability. To reach this goal, an *aspect-oriented approach* to IaC deployment and management was explored. We thus proposed Mantus, a IaC-based multi-cloud builder following the Orbits design [29]. Mantus features an aspect-oriented Domain-Specific Language called TOSCA Manipulation Language (TML) and a corresponding *aspect weaver* to inject flexibly non-functional services in TOSCA infrastructure templates.

[1] *Infrastructure as Code* may be defined as "the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools" [35]. Modelling cloud infrastructure (physical, virtualization layer, and VMs) as code managed by software engineering best practices present a number of benefits such as cost reduction, faster execution, and increased security and reliability.

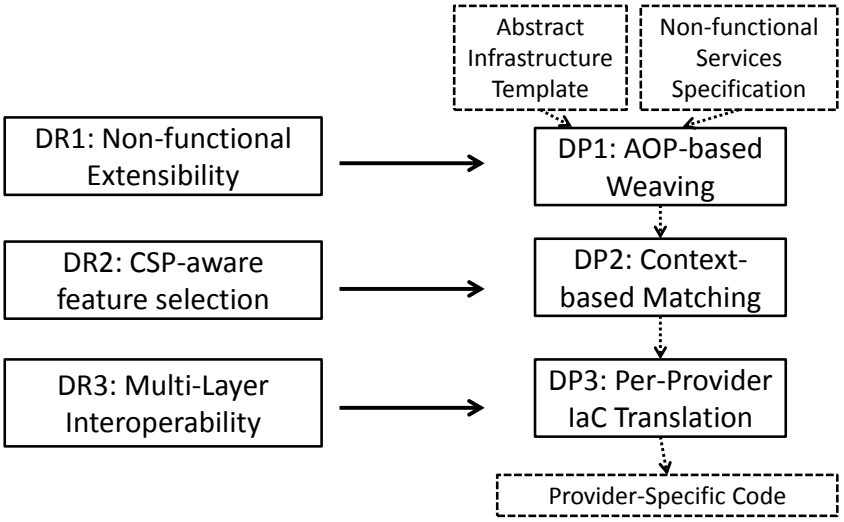

Figure 3.2: Mapping Design Principles to Design Requirements

3.2.1.2 Design

We started by defining a first overlay infrastructure completely managed through IaC templates [28]. This infrastructure extends the idea of a client-centric virtual infrastructure layer [12, 30] for enhanced resource control and multiple CSP support.
A full template-based description of software and hardware infrastructure resources is key to control the complete infrastructure life-cycle, which has proven difficult to manage even in single cloud provider settings.
A IaC-based virtual infrastructure layer should meet the following *Design Requirements (DRs)* (see Figure 3.2):

- **DR1: Non-functional extensibility**. Desired cloud infrastructures may greatly differ depending on the functional services deployed (e.g., Cloud Management System, SDN Controller). However, those basic infrastructure services should be manipulated to add complementary non-functional services (e.g., monitoring, auditing). This results in exploding complexity and does not allow to dynamically inject services inside basic templates.

- **DR2: CSP-aware feature selection**. To compete on the market, CSPs provide a number of differentiating services (e.g., DBMS-as-a-Service, Firewall-as-a-Service) additional to traditional resource provisioning (e.g., VMs, Object Storage). CSPs also propose differently-flavored resources (e.g., high I/O VM types, accelerated NICs) to better satisfy specific workloads. To effectively leverage CSP features, the interoperable virtual infrastructure should overcome simple "least common denominator" limitations to adapt instantiations to specific CSPs.

- **DR3: Multi-layer interoperability**. The multi-cloud should also meet the following goals to achieve the desired level of interoperability:

  1. *Interoperability*: The user should be able to describe its service specification (functional, non-functional, SLA) without knowing in advance on which CSP the multi-cloud will be deployed, and how it will be actually implemented. The virtual infrastructure layer should guarantee that the same abstract definition may be deployed on different CSPs, being "understood" by the different CSP orchestration engines.
  2. *Portability* of Execution Environments (EEs) where service components are executed on any CSP part of the multi-cloud. Deployed services should work similarly and expose the same APIs, requiring little adaptations from existing applications.

3. *Single point of orchestration* for services (e.g., Multi-Party Computation, high-availability Web Services) over the distributed multi-cloud infrastructure. This entry-point should be similar to widely-used frameworks (Mesos [20], Swarm [11], Kubernetes [3]) that integrate seamless extensions for multi-cloud enabled CSPs.

The deployment result is an *interoperable software layer* over a selected set of CSPs, coordinated by an *orchestration layer*, enabling multi-cloud awareness without handling heterogeneity.

The ORBITS architecture meets such requirements, overcoming limitations of traditional multi-cloud libraries in terms of application-specificity. ORBITS is based on the following *Design Principles (DPs)*:

- **DP1: AOP-based weaving**. To satisfy the non-functional extensibility requirement, we adopt an *Aspect-Oriented Programming (AOP)* [17] design approach. AOP is a programming paradigm that increases modularity by separating cross-cutting concerns, such as security, fault-tolerance, or persistence. Adding features does not require persistent modifications to the base code. Instead, additional behaviors (called *advices*) may be dynamically applied to the existing code at well-defined points. This approach is not only applicable to user-specified services but also to services aiming to optimize the infrastructure life-cycle management, such as load balancing or intrusion prevention.
- **DP2: Context-based matching**. To satisfy the CSP-aware feature selection requirement, we leverage a flexible form of matching derived from the TOSCA *Substitution Matching*. In TOSCA, the abstract definition of resources is separated from their actual implementations. According to inputs, resources may be matched with different equivalent implementations of the same service. The best implementation to fit a particular context (e.g., user-specified workload, selected CSP) should be chosen. We inflected this mechanism to our setting, introducing different forms of matching in TOSCA [8].
- **DP3: Per-provider IaC translation**. To satisfy the interoperability requirement, we defined a model of cloud infrastructure capturing the distributed aspect of multi-clouds, and the need of coordination between different services. This model was specified using TOSCA [26]. We also implemented a translator to transform a generic TOSCA description into code for a specific cloud provider.

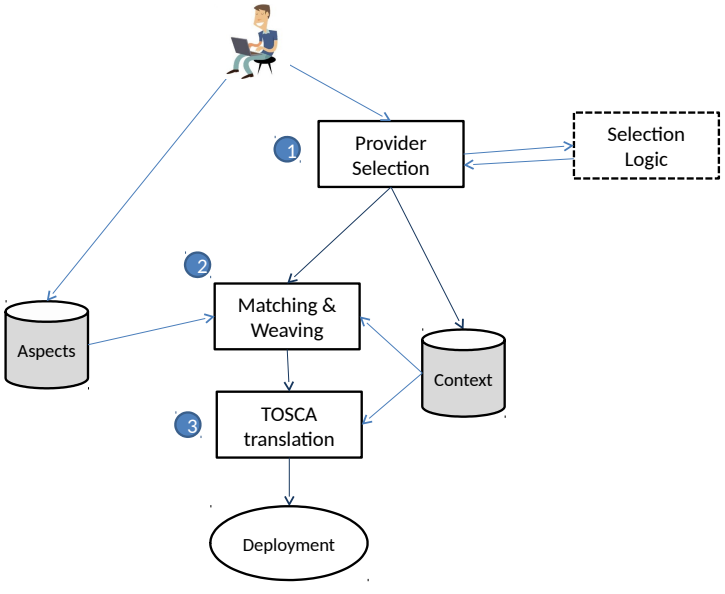

Figure 3.3: MANTUS workflow

Figure 3.2 maps DRs to steps of a simple workflow, showing how DPs handle different DRs. In the ORBITS and MANTUS implementation, those steps are refined into an implementation workflow shown in Figure 3.3 and including the following phases:

- *Matching:* A `ServiceTemplate` is created, composed of concrete nodes starting from an abstract specification. We adopted the standard TOSCA plug-in matching developed by Brogi et al. [8].
- *Fusion:* The matched file is generated, reconnecting different branches of matching.
- *Weaving:* Non-functional services are injected by applying TML scripts against the TOSCA `ServiceTemplate` as described in [29].
- *Translation:* Finally, the resulting TOSCA templates are translated to the Heat Orchestration Language (HOT) to be deployed on OpenStack as a target example of CSP.

This process enable to run U-Clouds over the multi-cloud infrastructure, choosing selectively the infrastructure elements and security services to deploy, as reported in the horizontal orchestration prototype presented in Deliverable D2.2 [22] We implemented the MANTUS TML weaver in Python. MANTUS first builds the expression abstract syntax tree for resources in the TOSCA graph using the pyPlus[2] LR-parser[3]. We used the tosca-parser[4] library to manipulate TOSCA `ServiceTemplates` for fully-compliant TOSCA generation. We also developed a driver-based small TOSCA translator supporting HOT and CloudFormation back-ends. The translator was developed from scratch to keep it minimal and to put emphasis on polymorphism for multi-CSP support.

3.2.1.3 APIs

MANTUS provides simple REST APIs shown in Table 3.1.

Request	Description	Parameters	Response
`POST URL/ucloud`	Create a new U-Cloud deployment	U-Cloud name SLA requirements Optimization criteria Security functions	U-Cloud id
`PUT URL/ucloud/id?`	Update a U-Cloud deployment	U-Cloud name SLA requirements Optimization criteria Security functions	U-Cloud id
`GET URL/ucloud/id?`	Get U-Cloud status information		U-Cloud id
`DELETE URL/ucloud/id`	Delete a U-Cloud deployment		U-Cloud id

Table 3.1: Virtualization and orchestration component API

3.2.1.4 Component access

The virtualization and orchestration component is being released as open source. The code is available on the SUPERCLOUD private repository[5]. Repository access may be granted by sending an email to marc.lacoste@orange.com. Instructions for installation and further documentation about the software are also distributed together with the code release.

[2] `https://github.com/erezsh/plyplus`
[3] Left-to-Right parser.
[4] `https://github.com/openstack/tosca-parser`
[5] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP2/virtualization/virtualization`

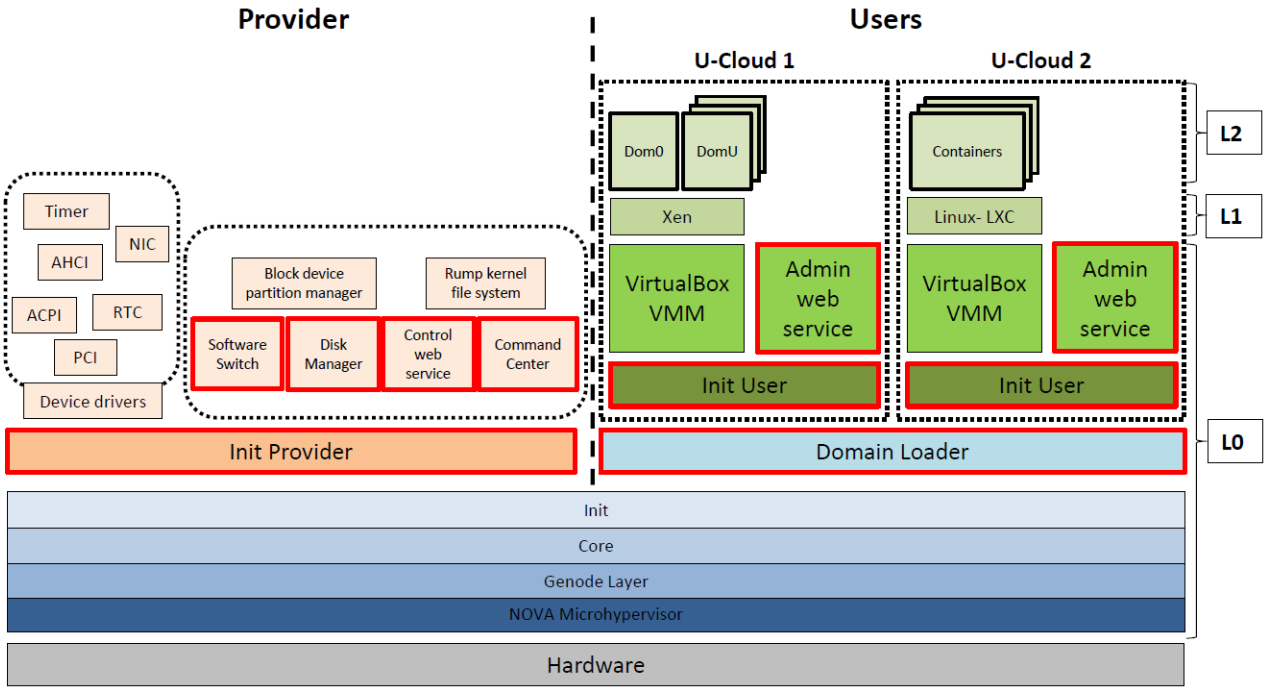

Figure 3.4: U-Clouds running on SUPERCLOUD-enabled private cloud

3.2.2 Micro-hypervisor

3.2.2.1 Objective

This component enables to instantiate U-Clouds on a single provider with cross-layer system support for U-Cloud security, deep in the virtualization infrastructure.

3.2.2.2 Design

U-Clouds should not only run on top of virtualization infrastructures under full provider control (e.g., public clouds running general-purpose hypervisors), but also on infrastructures where users can share control with the provider. This is typically the case of private clouds adopting the SUPERCLOUD virtualization architecture, combining nested virtualization and micro-hypervisor designs, and shown in Figure 3.4. This architecture was already presented in detail in Deliverable D2.1 [19]. We simply recall here its key features:

- The *lower layer* (L0) combines benefits of micro-hypervisors to minimize the virtualization layer and of modular hypervisors for disaggregation of control between several user domains. It is composed of a *micro-hypervisor* (NOVA [31] and its Genode [13] component library in Figure 3.4) and of a set of *user-level services* running on top of it. These services include: on the provider-side, device drivers and multiplexers, and administration tools allowing the provider to define the behavior of the architecture; and on the user-side, *user-centric services*, providing customers increased control over U-Clouds, of which they are fully part. In a multi-tenant scenario, such services are defined independently for each customer and should be isolated. Each user is thus in control of a user-level Virtual Machine Monitor (VMM) (VirtualBox VMM in Figure 3.4) handling non-privileged operations for VM management and of a set of user-centric services.

- The *upper layer* (L1) offers a provider-independent resource management plane aiming at interoperability between providers. This blanket layer addresses horizontal orchestration and is implemented by the virtualization and orchestration component described in Section 3.2.1. For

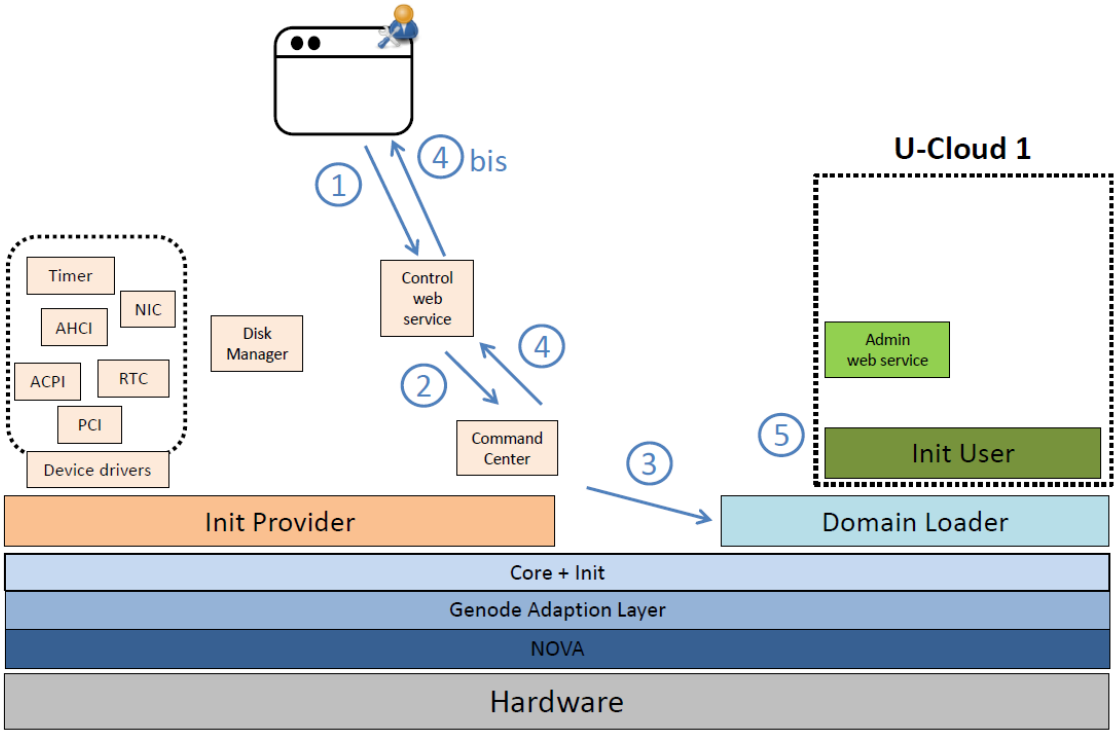

Figure 3.5: U-Cloud creation workflow

instance, it allows the creation of cross-provider U-Clouds. Key goals are interoperability and support for legacy cloud management technologies. For instance, it should support the widest possible range of configuration options to enable application-dependent trade-offs for users, notably for virtualization technologies, such as supporting both containers and VMs.

The cloud administrator only interacts with the provider-space user-level services, and cannot modify either the micro-hypervisor nor the U-Clouds. To maximize the user control benefits of the architecture, only system-wide concerns are handled in the provider-space. Every U-Cloud specific decision is left to the user. However, depending on the degree of control the provider desires to retain, more complex arbitration may be needed between the provider and user-controlled services.
The SUPERCLOUD micro-hypervisor component extends the NOVA/Genode micro-hypervisor [31]. A number of components were introduced (in red borders in Figure 3.4) to provide the basic features expected from a cloud node, and to overcome a number of limitations [6].
The micro-hypervisor component key features are:

1. Instantiating a U-Cloud;
2. Deploying a user-specific configuration;
3. Dynamically modifying this configuration without interrupting execution of the L2 VMs.

[6] For instance, basic Genode configuration of the execution environment did not support dynamic reconfiguration: the components launched, their allocated resources and the routing of the sessions had to be entirely defined in the configuration file given at boot time (`Init` component). Furthermore, VMs deployed had to be manually prepared on the hard drive in order to be launched by the VirtualBox VMM. Finally, the networking multiplexer was a software ARP proxy, which did not support multiple networking stacks operating on the same session. Because of that, nested VMs could not obtain connectivity, as they accessed the NIC driver through the session of their VirtualBox VMM.

Instantiating a U-Cloud by a client. The protocol is the following (see Figure 3.5):

1. The client performs a PUT request on the listening Control web service. The web service checks request conformity with the expected format, e.g., its header should specify an authentication token for the client. Ideally, the web service validates the token with an external entity.

2. If the request is appropriate, the web service notifies the Command Center by opening a temporary session (through RPC). The Command Center then validates the request (e.g., regarding availability of resources).

3. The Command Center instructs the Domain Loader component by RPC to build the U-Cloud, sending a minimal configuration to deploy, containing only an Admin web service and allocating an initial amount of resources. The static IP address of the Admin web service is selected (from an available address pool) by the Command Center and specified in the minimal configuration.

4. Upon acknowledgment by the Domain Loader, the Command Center informs the Control web service that the request has been processed and that instantiation is ongoing. The Control web service can thus reply to the client HTTP request with the appropriate code (202 Accepted). When the Control web service receives a specific GET request, the Command Center looks up the IP address assigned to the Admin web service, and returns it as the HTTP response body. The client can thus contact the Admin web service that has been created inside his U-Cloud, and use its management interface.

5. Meanwhile, the InitUser component is created by the Domain Loader, receiving the minimal configuration. It then deploys the Admin web service with the correct IP address.

Deploying services inside a U-Cloud. Once a U-Cloud has been instantiated, a client may deploy services inside it, such as a VMM:

1. After obtaining its IP address, the client can contact the Admin web service. The client can specify the desired configuration for the U-Cloud in XML through a PUT request. The Admin web service only parses the HTTP request, leaving configuration processing to the InitUser.

2. The Admin web service sends the XML configuration to the InitUser through the dedicated session. The InitUser validates its correctness. U-Cloud policies for the architecture are defined through the request processing infrastructure of the InitUser, e.g., available services, required and possible configuration information.

3. If the topology of services described in the XML is acceptable, the InitUser starts deploying it, negotiating with the Domain Loader the amount of resources. Every resource upgrade request is relayed to the Command Center for validation. In case of an invalid or deliberately improper configuration, only the InitUser (and corresponding U-Cloud) will be affected.

4. If the configuration contains a VirtualBox instance, the InitUser expects it to specify the type of VM that should be launched.

5. Once the configuration has been deployed completely, the InitUser notifies the Admin web service, which in turn replies to the client.

Updating a U-Cloud configuration at run-time. Because of their low complexity, most components do not properly handle run-time updates. Implementing such behaviors would have been a considerable task. While resource allocation can be propagated hierarchically, the parent-child interface is not enough for recursive reclaiming of resources[7]. A simple solution to limit resource consumption is to create a new updated U-Cloud and to destroy the previous one.

[7] This means that, while resources may be allocated by an InitUser component from the Command Center to launch a new service, they may not be reclaimed when a service exits in the U-Cloud.

This can be done seamlessly for L2 VMs as follows:

1. The client requests a new U-Cloud.
2. The request is transmitted to the Command Center which processes it.
3. The Command Center then transmits its instructions to the Domain Loader which deploys the U-Cloud.
4. The client can then deploy in the U-Cloud the new configuration. Modifications could range from simply modifying the L1 hypervisor (e.g., updating it to a more recent version) to deploying user services in the L0 layer (e.g., making all network access in the U-Cloud go through a firewall component).
5. Assuming both U-Clouds have been deployed with interoperable L1 hypervisors, the client can use their control interfaces to migrate the L2 VMs from the outdated one to the updated version. Most hypervisors support migration without significant impact on guest execution time.
6. When the migration is finished, the client can use the administration interface of the outdated U-Cloud to exit it. Only the updated version remains, now running the L2 guests.

3.2.2.3 External interface

A high-level TOSCA description of the micro-hypervisor component may be found in Figure 3.6.

```
node_templates:
  genode_box:
    type: orbits.boxes.Orbits
    properties:
      public_key: { get_input: key_name }
      public_net: { get_input: public_net }
      compute_image_url: { get_input: compute_image_url }
      compute_instance_type: { get_input: compute_instance_type}
      controller_image_url: { get_input: controller_image_url }
      controller_instance_type: { get_input: controller_instance_type }
    node_filter:
      properties:
        provider_name: { equal: Genode }
```

Figure 3.6: Micro-hypervisor component: TOSCA specification

3.2.2.4 APIs

The web services API is the run-time control interface of the component and provides an efficient summary of its features.

Control web service interface:

- **Creation of a new U-Cloud: PUT** request with an authentication token as header. If the request is accepted, the response contains an ID that can then be used to check the status of the U-Cloud. Otherwise, an error code dependent of the cause of failure is returned.
- **Checking the status of a U-Cloud: GET** request with an authentication token and an ID as headers. If the ID corresponds to an existing U-Cloud and the authentication token is valid, the response contains the IP address assigned to the Admin web server of the U-Cloud. Otherwise, an appropriate error code is returned.
- **Destroying a U-Cloud:** For the moment, this feature is only available through the Admin web service inside the U-Cloud.

Admin web service interface:

- **Launching a specific service in the U-Cloud** using a standard configuration: **POST** request with a service name and a label as headers. The label is used for further administration operations to target the correct service.
- **Deploying a configuration as a sub-system inside the U-Cloud: PUT** request with an XML configuration as body. The header `Clean-previous` can be set to `true` to request closing of all active services in the U-Cloud prior to deployment.
- **Obtaining an XML list of services currently running in the U-Cloud**: **GET** request.
- **Closing** either a specific service, all services, or the U-Cloud entirely: **DELETE** requests, depending on the URL specified. Closing one or more services does not reclaim allocated resources by the Command Center. It is thus generally preferable to close the entire U-Cloud and relaunch another with a different configuration.

Command Session interface:

The Command Session is the session opened by the Control web service on the Command Center to transmit requests. To reduce the node attack surface, it implements only two functions, symmetrically to the two types of requests that the web service handles:

- `create_subsystem()`: requests the creation of a U-Cloud and returns the ID if it succeeded.
- `subsystem_info(id)`: returns the IP address used by the Admin server of the specified U-Cloud; or an empty string if the U-Cloud does not exist.

Domain Loader Session interface:

The Domain Loader Session is the session through which the Command Center requests the launch of a U-Cloud to the Domain Loader. Each U-Cloud deployment is requested through a different instance of the Domain Loader Session. Thus, it is always possible for the Command Center to terminate one of them and regain its resources, without having to destroy all existing U-Clouds[8].

3.2.2.5 Component access

The micro-hypervisor component is being released as open source. The code is available on the SUPERCLOUD private repository[9]. Instructions for installation and further documentation about the software are also distributed together with the code release.

[8] To increase code modularity, each Domain Loader Session instance in the Command Center is encapsulated inside a handler class. Destroying a U-Cloud in the Command Center is thus as easy as deallocating its handler object.

[9] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP2/virtualization/virtualization`

3.3 Isolation and trust management

Trust, its management, and isolation are tightly coupled properties and need to go hand in hand when used in the context of SUPERCLOUD. Trust in computer systems requires reliable information about all components that can influence the system's behavior. For complex systems, obtaining such reliable information may not be feasible, which makes the management of systems with regards to trust challenging and often even impractical. Isolation is, however, an important building block that can help to reduce the complexity of systems. With isolation, only a defined (sub-)set of components can influence a system's behavior. This means that only a subset of components needs to be trusted, making management of trust more straightforward. At the same time, reliable isolation requires that the isolation mechanism and its correct instantiation can be trusted. Hence, without trust management for isolation mechanisms, the overall trust in the system cannot be narrowed down.
In this Section, the SUPERCLOUD isolation and trust management components are described, starting with the isolation component, followed by the trust management component.

3.3.1 Isolation component

The goal of the *isolation component* is to provide an execution environment which allows applications to run either in a globally trusted environment, like a private cloud, or in a trusted execution environment, like an Intel SGX enclave. The application should be able to run unmodified in both scenarios, while still facilitating the functionalities of the different environments. For instance, it should enable secure channel establishment from an SGX enclave with the help of the remote attestation feature of SGX.
To achieve this goal, the execution environment is abstracted through the use of an *interpreter*. In particular, a Python interpreter is modified, such that it can execute arbitrary Python scripts inside an SGX enclave. Thus the scripts can be run in both environments, a private cloud and an SGX enclave.
The Python interpreter behaves mostly like a normal Python interpreter. In principle, any Python code and module can be executed. So far, we have tested it with code dealing with files and network connections.

3.3.1.1 Trust

Normally, the Python interpreter loads additional code at run-time: Python code is read directly from the source files, while native code for Python modules having a C component (as opposed to pure-Python modules) is loaded from dynamic libraries.
To trust a Python enclave, one must trust all code available in the enclave, including Python code. Moreover, the processor creates a *measurement* of the enclave (`MRENCLAVE` hardware instruction) when the enclave is initialized and before it runs, which contains a cryptographic hash of the code of the enclave which is tied to the individual processor. Hence, it is desirable to load all native and Python code at initialization time, to get a complete measurement of all the enclave code. Thus, we link the native components statically when the interpreter itself is built, so that no dynamic libraries are needed. Moreover, all Python code necessary for the core and the modules is embedded as text in the enclave memory.

3.3.1.2 Prerequisites

Our enclave has been implemented on Linux. No special libraries are required, apart from the Intel SGX SDK.

3.3.1.3 External interface

A high-level TOSCA description of the isolation component may be found in Figure 3.7.

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template for Python SGX enclave.
topology_template:
  inputs: # omitted for brevity
node_templates:
  pyenclave:
    type : supercloud.computing.Component
    requirements:
      - sgxplatform: supercloud.computing.sgxcpu
    interfaces:
      script_execution:
        pysgx_open_debug_console: # omitted for brevity
        pysgx_run_script:
```

Figure 3.7: Isolation component: TOSCA specification

The Python enclave works like a normal Python interpreter. There are two modes of operation:

- `pysgx_open_debug_console`: when started without parameters, the interpreter opens an interactive shell, which can be used for debugging purposes – the interactive shell processes input coming from the untrusted OS, so it should be disabled in release mode.
- `pysgx_run_script`: when given a Python file as a parameter, the interpreter will execute the script and then exit.

As mentioned above, a number of Python source files are embedded in the enclave in order to protect their integrity. The user script can either be embedded in the enclave at compile-time, or it can be retrieved from the normal file system – this requires trusting the OS and the file system, since the integrity of the script is not checked by the enclave.
One option to overcome the necessity to provide the script at compile-time is to prepare a bootstrap script that retrieves the actual script in a secure way from an external untrusted source. As an example, the bootstrap script could retrieve the code from the network or a file, check its signature against a hard-coded list of trusted certificates, and finally execute the code if the signature is correct. This bootstrap script can be embedded statically in the enclave while still allowing arbitrary dynamic computation.

Example

```
> /path/to/starter /path/to/python-enclave.signed.so
>>> def fib(n):
...     if n < 2:
...         return n
...     return fib(n-2) + fib(n-1)
...
>>> list((i, fib(i)) for i in range(1,9))
[(1, 1), (2, 1), (3, 2), (4, 3), (5, 5), (6, 8), (7, 13), (8, 21)]
```

3.3.1.4 Deployment

Conceptually, the enclave could be deployed by distributing only two files (called `starter` and `python-enclave.signed.so` in the example above. In our demonstrator, we prove the feasibility of this approach by embedding all required Python code. Including other required data, like configuration files and file metadata, can be done using the same approach.

3.3.2 Trust management component

3.3.2.1 Objective

This component aims to manage trust between computing abstractions at hardware level, through building and verification of *Chains of Trust (CoT)*.
We consider more specifically the Intel SGX model to build and verify trust relationships between SGX enclaves. Intel's Software Guard Extensions (SGX) is an extension to Intel architecture to generate protected software containers, referred to as *enclaves*. Inside an enclave, software code, data, and stack are protected by hardware-enforced access control policies that prevent attacks against the enclave content.
To establish confidence between enclaves, a assertion is needed as evidence to demonstrate that the enclave has been properly instantiated on the SGX platform, locally or remotely. This process is known as *attestation*. The SGX run-time generates evidence structures called *reports*, cryptographically bound to the hardware, that may be presented for verification to third parties, so that they may make decision on the ability of the enclave to operate in a certain state.
To perform remote attestation, SGX enables a special enclave, the *quoting enclave*, to be remotely created. This enclave verifies reports from other enclaves on the remote platform using intra-platform attestation. It then replaces the MAC in those reports with a new MAC computed with the private key of the verifier enclave using a dedicated asymmetric cryptographic scheme. The output of this process is called a *quote*.
While a number of SDKs have been defined to support enclaves, either natively [2] or emulating enclave behavior at system-level through the Open SGX virtualized architecture [14], our aim is to identify the minimal key interfaces to support such CoTs. We describe that interface starting from the wider API of Open SGX - a first basic component implementation being provided by the corresponding subset of the Open SGX code base [1].

3.3.2.2 External interface

The trust management component has three main interfaces for key management, communication channels, and attestation. A TOSCA description is shown in Figure 3.8.

- **Key management:** To guarantee attestation, SGX allows creating cryptographic keys (`EGETKEY`) and cryptographic reports (`EREPORT`) to check the integrity of an enclave during exchanges with other enclaves. An interface is thus needed to create keys, get reports from SGX, and check integrity of reports by comparing the computed report with a received one.
- **Communication channels:** Interactions between elements of a CoT need also to be managed such as handling network connections between enclaves, and reading/writing to/from enclaves.
- **Attestation:** This user-facing API implements the CoT attestation protocols described in [16], leveraging the previous helper interfaces. Several methods are distinguished depending on the type of SGX platform (local or remote) and on the attestation role (challenger or target).

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template for trust management component.
topology_template:
  inputs: # omitted for brevity
node_templates:
  trustmanager:
    type : supercloud.computing.Component
    requirements:
      - sgxplatform: supercloud.computing.sgxcpu
    interfaces:
      key_management:
        cot_getkey: # omitted for brevity
        cot_getreport:
        cot_check:
      communication:
        cot_make_server: # omitted for brevity
        cot_connect_server:
        cot_read:
        cot_write:
        cot_make_quote:
      attestation:
        cot_intra_attest_challenger: # omitted for brevity
        cot_intra_attest_target:
        cot_remote_attest_challenger:
        cot_remote_attest_target:
        cot_remote_attest_quote:
```

Figure 3.8: Trust management component: TOSCA specification

3.3.2.3 APIs

The **key management interface** enables to create keys, to sign reports, and to check report integrity. Its key methods are the following:

- `cot_getkey` This method invoked by an enclave allows to create a cryptographic key needed for signing a report.
- `cot_getreport` This method takes a generated SGX key, and enables to create, and then to sign a report.
- `cot_check` This method takes a Report Key, and allows to compute a MAC and to compare it with the MAC contained in the received report to check its integrity.

The **communication channels interface** manages local and network connections between enclave processes, as well as quote operations. Its key methods are the following:

- `cot_make_server` This method starts a server socket, and waits for connections from entities wishing to attest to the CoT.
- `cot_connect_server` This method connects to an already started server to attest to a CoT.
- `cot_read/cot_write` Those methods enable enclave programs to read/write to communication channels during attestation protocols.
- `cot_make_quote` This method realizes a *quote* operation used for attestation between remote SGX platforms: it performs a secure hash of a report, generates an RSA key for future communications between remotely attested elements, and signs the report with the RSA key.

The **attestation interface** enables to perform both *intra-attestation* and *remote attestation*, which correspond respectively to attestation between enclaves running on the same SGX platform, or on remote SGX platforms. This interface implements the protocols specified in [16]. Its key methods are the following:

- `cot_intra_attest_challenger` This method is called by a challenger enclave to perform intra-platform attestation, i.e., to get an attestation from a target enclave on the same SGX platform.

- `cot_intra_attest_target` This method is called by the target enclave to perform intra-platform attestation, i.e., to implement for the target enclave to respond to an attestation request on the same SGX platform.
- `cot_remote_attest_challenger` This method is called by the challenger enclave to perform remote attestation to a remote target enclave (e.g., identified by its IP address).
- `cot_remote_attest_target` This method is called by the target enclave to perform remote attestation on the target side. It provides a response to an attestation requested by a remote challenger enclave.
- `cot_remote_attest_quote` This method is called by the quoting enclave co-located with the target enclave in remote attestation. It computes a secure hash of the received report from the target enclave, and generates an RSA key to send back to the target.

3.4 Cloud FPGA support

This part of the infrastructure is implemented by a single component, the Cloud FPGA (Field Programmable Gate Array) framework.
FPGAs are increasingly used in data centers to offload and accelerate compute-intensive operations. But these FPGAs are not yet available to general cloud users who want to get their own workload processing accelerated. This puts the cloud deployment of compute-intensive workloads at a disadvantage compared with on-site infrastructure installations, where the performance and energy efficiency of FPGAs are increasingly being exploited.
The *Cloud FPGA framework* solves this issue by offering FPGAs to the cloud users as an IaaS resource. Using the Cloud FPGA framework, cloud users can rent FPGAs, similarly to renting VMs in the cloud, and get their workload processing accelerated. Cloud FPGA is available in the OpenStack-based public cloud hosted at IBM Research Zurich lab. Instances of Cloud FPGA are accessible over the Internet and can be used by following the steps explained in Section 3.4.2.

3.4.1 Objective

Cloud FPGA is used to off-load CPUs from intensive computations, such as computations related to cryptography or security in the SUPERCLOUD infrastructure. Cloud FPGAs are accessed from VMs over the data center network. A distributed application is split into software (SW), running in the VM, and hardware (HW), running in the Cloud FPGA. They communicate with each other over standard TCP/IP for offloading tasks from the CPUs.

3.4.2 External interface

This Section explains the infrastructure-level interface for accessing Cloud FPGA instances. The steps to use Cloud FPGA instances, which are shown in Figure 3.9 and Figure 3.10, are as follows:

(1) Connect to the VPN server at IBM Research Zurich lab.

(2) Connect to the Linux container (CT) over SSH.

(3) Make a connection from the CT to the Cloud FPGA over TCP/IP.

(4) Send data to the HW application. The HW application operates on the sent data or commands and returns a response. Steps 4.1 to 4.4 in Figure 3.10 show the interaction between the SW part of the application (running in the VM) and the HW part of the application (running in the Cloud FPGA) over TCP/IP with string-based messages.

(5) Close the connection from the CT to the Cloud FPGA.

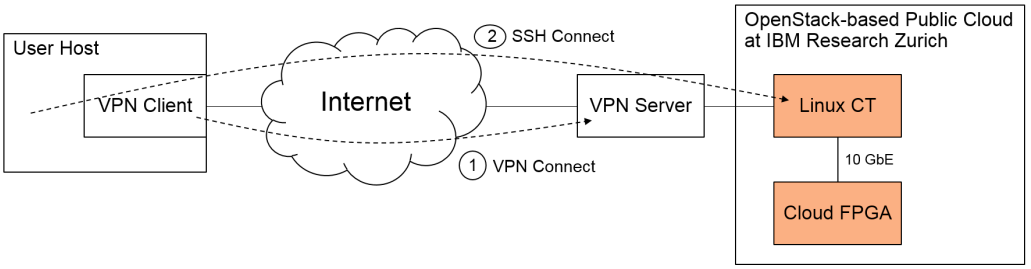

Figure 3.9: Access to Linux CT over VPN and SSH

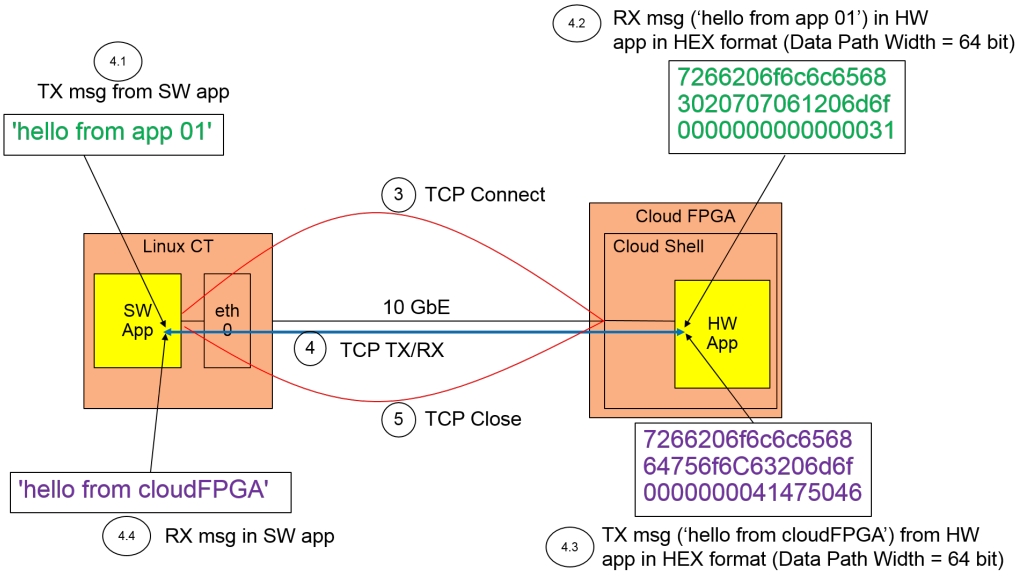

Figure 3.10: Access to Cloud FPGA from Linux CT

Chapter 4 Self-management infrastructure

In this Chapter, we describe the self-management infrastructure in terms of structure and components. We start by providing a general overview of the infrastructure (Section 4.1). We then present the different components, separating the orchestrator of protection services (Section 4.2), from the security services themselves (Section 4.3). For each component, we give a short overview of its functionality and external interface, with for most components, a more detailed API description. We also include information about how to obtain and use each component.

4.1 Infrastructure overview

The structure and components of the self-management infrastructure are shown in Figure 4.1.

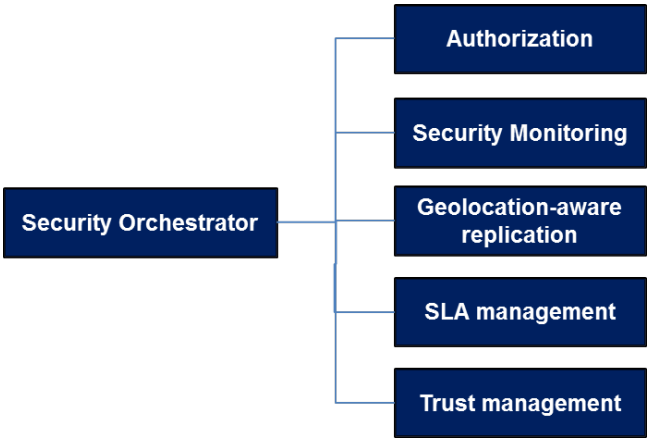

Figure 4.1: The self-management infrastructure

4.1.1 Security orchestration

This part of the infrastructure coordinates security services. This task is implemented by a single component, the *Security Service Orchestrator*. This component is described in Section 4.2.

4.1.2 Security services

A number of *security services* are also provided that the customer can select to build a secure U-Cloud matching his security requirements. Security services are inserted into the virtualization infrastructure, either horizontally, or vertically and guarantee protection of the U-Cloud. The provided services are the following:

- *Authorization:* This component enables to perform resource access control at different levels of the infrastructure. This component is described in Section 4.3.1.

- *Security monitoring:* This component enables to detect and to react to threats on the U-Cloud, taking into account multi-layer and multi-provider aspects of the virtualization infrastructure. This component is described in Section 4.3.2.
- *Geolocation-aware replication:* This component that is deployed in each VM of a U-Cloud handles data replication separately when this functionality is needed for data only in allowed locations. It notably integrates the authenticated discovery protocol defined in Deliverable D2.2 [22]. This component is described in Section 4.3.3.
- *SLA management:* This component implements monitoring and arbitration of Security SLAs (SSLA), in close liaison with the SLA management infrastructure defined in Deliverable D1.4. This component is described in Section 4.3.4.
- *Trust management:* This component aims to assess trust that a customer can put in a given multi-cloud, i.e., before making a decision to choose a given provider for a specific service requested by a customer, it will compute a trust value based on a trust model, past experience and observable provider behavior. This component is described in Section 4.3.5.

4.2 Security orchestration

In this Section, we provide a brief presentation of the implementation of security orchestration. This task is implemented by a single component, the *Security Service Orchestrator*.

4.2.1 Security orchestrator

4.2.1.1 Objective

The main objective of the Security Orchestrator is to automate the instantiation, configuration and coordination of security services developed within the SUPERCLOUD project. This component also ensures the automation of inter-services interactions[1].

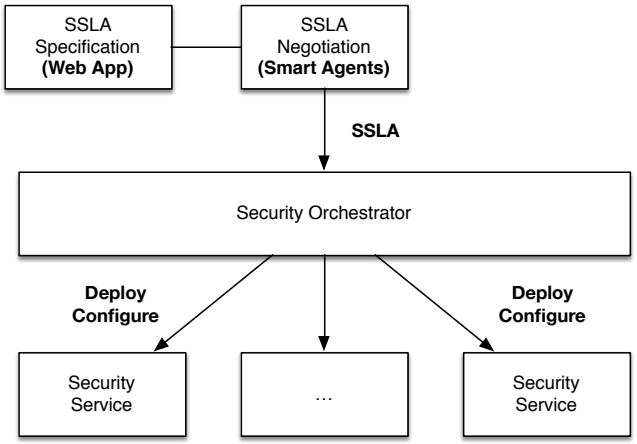

Figure 4.2: Security Orchestrator overview

As illustrated in Figure 4.2, the Orchestrator processes the active SSLA to identify the security services to be deployed and the required configuration parameters. Mimicking the micro-services[2] approach, the Orchestrator generates a composite security services deployment and configuration file. This file is then processed by a deployment engine that will build, configure and run each security service. This process is described in detail in the next Section.

[1] A more detailed description of the Security Orchestrator can be found in Deliverable D1.4.

[2] *Micro-services* is a new application development paradigm that shares many similarities with Service-Oriented Architecture (SOA) as it tries to structure monolithic applications into a set of loosely-coupled services.

4.2.1.2 Security orchestration approach

The SUPERCLOUD project partners agreed on using Docker[3] containers to encapsulate self-management of security services. The choice of containers allows to build agile software delivery pipelines to ship new features faster, more securely for any operating system.
While the Docker Engine works well for packaging simple and single-container application and services, it is known to be less suitable for defining and deploying complex applications that consist of numerous dependent and independent services, which is the case for self-management of security.

To address this issue, we build on the top of the Docker Compose tool [4] to implement deployment and configuration features of the Security Orchestrator. The overall orchestration process is split into three phases, each operated by a dedicated engine that we illustrate in Figure 4.3.

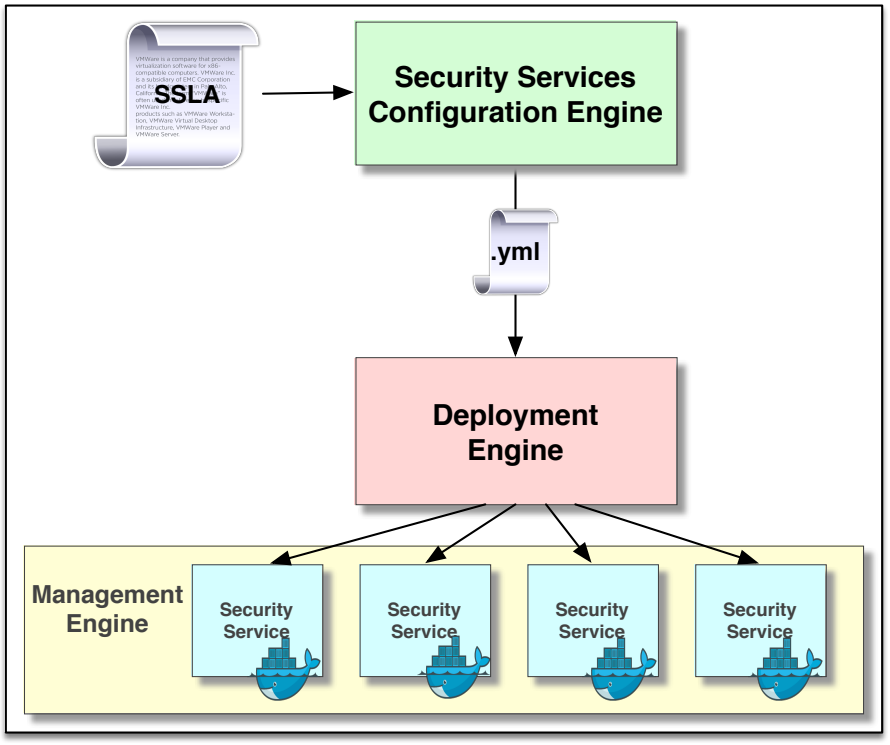

Figure 4.3: Security Orchestrator architecture

First, based on the SSLA specified by the Cloud Service Customer, we identify the security services to deploy and prepare a configuration file to specify the parameters of each service. Second, the generated configuration file is pushed to the deployment engine that will instantiate and configure the services. Finally, in the management phase, we monitor service execution and trigger auto-healing mechanisms in faulty settings. In what follows, we present each of the three steps of the security orchestration workflow.

1) Configuration
The configuration of self-management of security services is achieved through a single configuration file. This file specifies how services can interact with each other in a coherent and consistent way. It defines the order in which services are deployed and the conditions under which a service may be invoked or not. The file defines parameters that are necessary for the execution of each service such as network ports to be opened, volumes to be mounted or dependencies with respect to other services. We provide in Listing 4.1 an example of what a deployment configuration file looks like.

[3] `https://www.docker.com/what-docker`

```
1  services:
2
3  authorization:
4  build: .
5  ports:
6  - "8080:8080"
7  volumes:
8  - /policies:/policies
9  db:
10 image: mysql
11 ports:
12 - "3306:3306"
13 environment:
14 MYSQL_ROOT_PASSWORD: 123456
15 MYSQL_USER: supercloud
16 MYSQL_PASSWORD: 123456
17 MYSQL_DATABASE: selfmanagement
```

Listing 4.1: Example of a deployment file generated by the Orchestrator

In this example, the Security Orchestrator will deploy two services, an *authorization service* and a *storage service* containing persistent data such as monitoring information. The authorization service is built using a Docker file placed in the current directory.
The instructions specify that the authorization service needs to expose port `8080` and map it to port `8080` outside the container. The policies used for authorization are copied from the mounted volume `/policies`. For the storage service, a default `MySQL` images is used (cf. line 10) and port `3306` is mapped to `3306`. Additional configuration information (e.g., root password, database name) is also specified (cf. lines 13-17).

2) Deployment
The current version of the Orchestrator relies on the Docker Compose tool for the deployment of security services (i.e., containers and links between them). Deployment is done in two steps.

First, the Docker images are built based on the instructions provided within the individual `Dockerfile` files. Listing 4.2 provides an example of the `Dockerfile` used for the authorization service.

```
1  FROM ubuntu
2
3  MAINTAINER Reda Yaich <reda.yaich@imt-atlantique.fr>
4
5  # Update the base ubuntu image with dependencies needed
6  RUN apt-get update && \
7  apt-get install -y openjdk-8-jdk && \
8  apt-get install -y ant && \
9  apt-get clean;
10 RUN apt-get update && \
11 apt-get install ca-certificates-java && \
12 apt-get clean && \
13 update-ca-certificates -f;
14
15 # Setup JAVA_HOME, this is useful for docker commandline
16 ENV JAVA_HOME /usr/lib/jvm/java-8-openjdk-amd64/
17 RUN export JAVA_HOME
18
19 # Expose the required ports
20 EXPOSE 8080:8080
21
22 #copy files to the created container volumes
23 COPY /Authorization_Service /Authorization_Service
24 COPY /policies /Authorization_Service/bin/policies
25
26 #set the working directory
27 WORKDIR /Authorization_Service
28
29 #Specifies the entrypoint to the docker
30 ENTRYPOINT ["/Authorization_Service/start.sh"]
```

Listing 4.2: Authorization service Dockerfile

The command `docker-compose build` will process the `docker-compose.yml` file present in the current folder and create the image of each security service.

Then, using the command `docker-compose run`, the security services are executed as follows:

- A self-management dedicated virtual network is created.
- Volumes are mounted for security services requiring such operation.
- Images of each service are pulled by Docker.
- Creation of services is ordered based on dependencies.

At this stage, the services are deployed as shown in Figure 4.4.

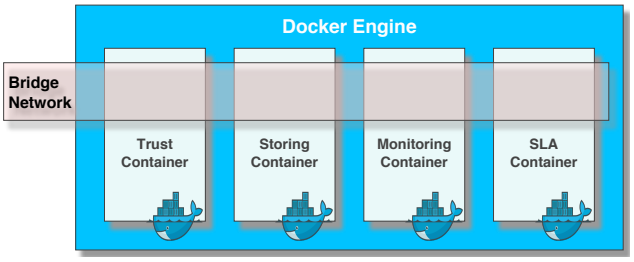

Figure 4.4: Illustration of security services deployment

We show in Figure 4.5 the corresponding process displayed by the Docker machine.

```
CREATED             STATUS              PORTS                           NAMES
4 minutes ago       Up 4 minutes        80/tcp, 0.0.0.0:1982->8080/tcp  orchestrator_trust_1
4 minutes ago       Up 4 minutes        80/tcp, 0.0.0.0:8080->8080/tcp  orchestrator_authorization_1
4 minutes ago       Up 4 minutes        80/tcp, 0.0.0.0:1981->8080/tcp  orchestrator_ssla_1
4 minutes ago       Up 4 minutes        0.0.0.0:3306->3306/tcp          orchestrator_storage_1
```

Figure 4.5: Deployed security services

3) Management
In the deployment phase, security services are deployed on the host successively to avoid dependency conflicts. After that, the Orchestrator enters the management phase to prevent the self-management of security framework from running in an unhealthy state (e.g., due to faulty services). Two main states are handled by the Security Orchestrator:

1. **Overloaded services**. This situation can occur when the user multi-cloud scales-up in response to client demand. Consequently, some security services (e.g., monitoring, authorization) need to adapt to such change. To address this issue, the Orchestrator makes use of the `docker-compose scale SERVICE=X` command, provided by `Docker-Compose` and `Docker Swarm`, to launch `X` instances of the considered service.

2. **Faulty services**. For some reasons, security services may stop. As a self-managed security service, the Orchestrator needs to re-run without human intervention security services. This is achieved using command `docker-compose up --no-recreate` that allows the Orchestrator to re-launch the same service without re-building its image, hence reducing recovery time.

The automation of restarting faulty security services is important due to the critical nature of their objectives. However, in some settings, continually restarting faulty containers embodying these services may block the overall self-management process. This is mainly due to the restart process

loop that will fill up the physical host disk space. This is relatively common when handling stateful services such as MySQL. This is particularly true for the storage service in our architecture. To address this issue, we make use of a more sophisticated management scheme that relies on the **Swarm** mode [5] of the Docker engine. Within Swarm, the Docker engine makes use of an explicit restart policy that needs to be specified within the `docker-compose.yml` configuration file.

Listing 4.3 shows an example of `Docker-Swarm` specific restart instructions. The `max_attempts` parameter makes the deployment process safer by fixing a limit to container restarts. The `replicas` statement enables to specify the initial number of instances for each security service.

```
...
deploy:
replicas: 3
restart_policy:
condition: on-failure
delay: 30s
max_attempts: 3
window: 60s
...
```

Listing 4.3: Restart policy configuration in Swarm

4.2.2 Component access

The orchestrator is a core component in SUPERCLOUD self-management of security, split into three sub-modules, each responsible for a specific phase of the process presented earlier (i.e., configuration, deployment and management). We describe next how the deployment module can be downloaded and tested. The other modules will be released within Deliverable D1.4 as part of the overall self-management of security implementation.
The component is currently provided as a `Java JAR` application. The objective of this component is to deploy the configuration of security services that have been derived based on the SSLA. We list in what follows the steps needed to download and run the Orchestrator.

- Download the security service deployment module from the SUPERCLOUD repository[4].
- The component needs to be placed in the root directory containing security services resources as follows:

```
/
├── SSLA2Config.jar ..... Converts SSLA to services configurations
├── Deployer.jar ..... Deploys the configuration
├── docker-compose.yml ..... Generated file
├── /ServiceName ..... Service folder
│   ├── service.cfg ..... Service-specific configuration
│   ├── Dockerfile
│   └── /Resources ..... Other resources
└── /ServiceName ..... Other security service
    ├── service.cfg
    ├── Dockerfile
    └── /Resources
```

- Execute the `SSLA2Config.jar` to generate the configuration file.
- Execute `java -jar Deploy.jar` to deploy security services on the local machine.
- An execution trace will be displayed to verify that services are running.

[4] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP2/selfmanagement/sorchestrator`

4.3 Security services

4.3.1 Authorization

In this Section, we present the SUPERCLOUD authorization service. This service represents the Policy Decision Point, called OrBAC-PDP in Deliverables D1.2 [37] and D2.2 [22].

4.3.1.1 Objectives

The service derives decisions (i.e., permission, prohibitions, obligations) from OrBAC policies. The service ensures both Access and Usage Control functions within the SUPERCLOUD framework. The authorization service is invoked by various applications (i.e., Philips Imaging Platform, geo-replication service) that represent the Policy Enforcement Points (PEPs) in standard Usage Control settings (see Deliverable D2.1 [19]). It notifies these services in usage control mode.
As illustrated in Figure 4.6, the SUPERCLOUD authorization service works in a classical client-server mode wherein the application (here the Philips Imaging Platform) plays the role of the client and the SUPERCLOUD authorization service the role of server.

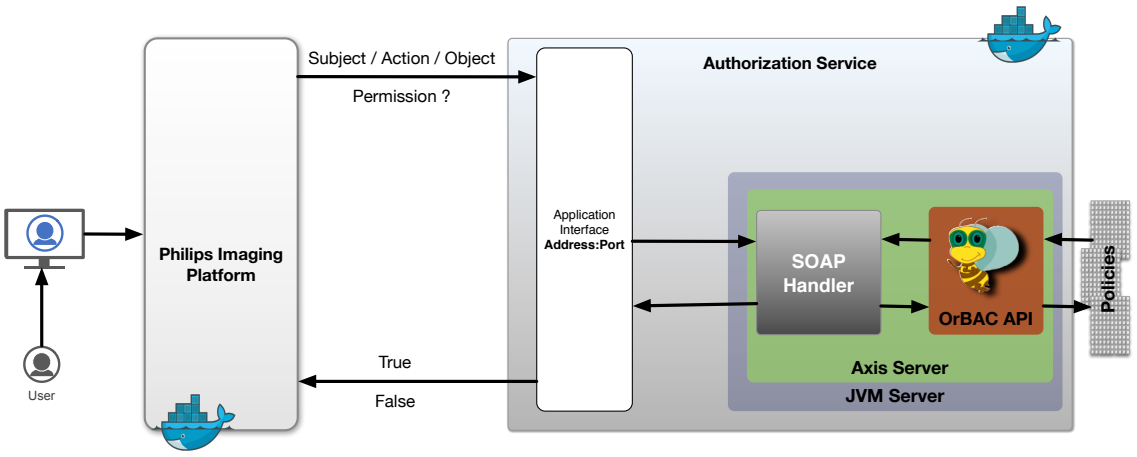

Figure 4.6: Illustration of the Integration of the Authorization Service with the Philips Imaging Platform (Use Case)

The Philips Imaging Platform performs standard `HTTP` requests. Authentication is achieved by a dedicated service that needs to be specified. Hence, we assume that the SUPERCLOUD authorization service considers all identities as valid.

4.3.1.2 External interface

The SUPERCLOUD authorization service REST API can be accessed in two ways:

- The standard and public REST API only responds to GET request and runs at `http://IP:PORT/services/SuperCloudAuthorizationService/` endpoint. This API does not require any authentication.
- The admin-only REST API runs at the `SuperCloudAuthorizationService/admin/` endpoint and responds to **GET, POST, PUT** and **DELETE** requests[5].

[5] This first version of the REST API specification does not detail the administration aspect of the SUPERCLOUD authorization service.

Responses. Depending on the nature of the request received by the REST API, the responses computed by the SUPERCLOUD authorization service may be decision(s), subject(s), objects(s) or action(s). In this Section, we provide some examples of responses provided by the REST API.
Decisions can be of three types; ***permissions***, ***prohibitions*** and ***obligations***. A decision is made with respect to a triple `(subject, object, action)`. For example, to verify if a subject `s` is authorized to perform an action `a` on an object `o`, the syntax of the request should be as follows:

o `/IsPermitted?subject=s&action=a&object=o`

With respect to the Philips Imaging Platform, one can request if a scope, that is mapped to view in the OrBACModel, is active for a certain subject or not. This can be achieved by invoking the `GetAssignedViews` method.
All requests represent a conjunction of conditions in which the above-mentioned triples can be presented in any order. The SUPERCLOUD authorization service standard reply to such decision requests is a Boolean value.
The SUPERCLOUD authorization service REST API is shown in Table 4.1.

Request	Description	Parameters	Response
`GET URL/IsPermitted?`	Checks if an action is permitted on a subject	`subject=s&action=a&object=o`	Boolean
`GET URL/IsProhibited?`	Checks if a scope is prohibited	`subject=s&action=a&object=o`	Boolean
`GET URL/IsObliged?`	Checks if a scope is obliged for a user	`subject=s&action=a&object=o`	Boolean
`GET URL/Actions?`	Retrieves all active actions in the policy	None	List of actions
`GET URL/Objects?`	Retrieves all active objects	None	List of objects
`GET URL/Subjects?`	Retrieves all active subjects	None	List of subjects

Table 4.1: Authorization service API

Response status codes. Response codes are handled in classical settings with respect to protocols such as HTTP. Listing 4.4 provides an example of a request and the associated response code.

```
Request :
> HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.49.1
> Accept: */*
> http://localhost:8080/services/SupercloudAuthorizationService?wsdl

Reply :
< HTTP/1.1 200 OK <--- Returned Code
< Date: Sat, 21 Jan 2017 11:18:44 GMT
< Server: Simple-Server/1.1
< Transfer-Encoding: chunked
< Content-Type: application/xml; charset=UTF-8
```

Listing 4.4: Authorization service request and reply

Here the code is 200, meaning that the request was successfully processed. Similarly, the REST API can reply with the following codes:

- **200** A response code of 200 means the request was successful and details about the response can be found in the body of the response.
- **201** The requested POST operation was successful and an object was created in the system.
- **204** The requested operation was successful and there is no response body.
- **400** The request was improperly formatted. The user should verify that the request conforms to this specification and re-issue the request in a properly formatted manner.

- **401** UNAUTHORIZED request code.
- **404** The requested resource does not exist.
- **403** The request was not allowed because the request did not pass authentication or the proper access rights to the target have not been granted.
- **500** The SUPERCLOUD authorization service failed to process the request because of an error inside the SUPERCLOUD authorization service. These responses should be reported to the SUPERCLOUD authorization service as they always represent an internal bug.
- **501** The user requested an action on a resource that does not support that action.
- **503** The SUPERCLOUD authorization service is temporarily unavailable for API queries.

Response entities. All GET methods respond with the JSON or XML of the resource(s) being requested. HEAD methods have no response entity. POST methods may respond with a 201 CREATED or 202 ACCEPTED response code depending on whether the creation completed immediately or is an asynchronous operation. PUT and DELETE methods generally respond with 204 NO CONTENT unless the operation is a long-lived operation. In those scenarios, the PUT method will respond with a 202 ACCEPTED response code and include a Job resource in the response entity.

Response format. While forming the request, one may specify an "Accept" header to define whether we wish to receive responses as XML or JSON[6]. The values that can be specified for "Accept" are thus `application/xml` or `application/json`.

4.3.1.3 Component access

In this Section, we describe the procedure to retrieve and deploy the authorization service. As illustrated in Figure 4.6, the service is available as a REST application deployed inside a Docker container. We provide a deployment-ready OrBAC authorization service Docker image.
We list hereafter the procedure to follow to run and test the service. The following deployment process has been tested in Linux and Mac OS.

- Download the tar compressed file containing the service Docker image[7]
- The archive contains :
  - The authorization service repository that wraps the OrBAC REST API service running using Apache Axis 2.
  - A Docker file that specifies the way the image should be built.
  - A (`start.sh`) shell script that runs the service.
- Launch the `./start.sh` script to start the authorization service.
- Test if the service is running.
  - Go to the URL address: `http://localhost:8080/axis2/services/SupercloudAuthorizationService?wsdl`
  - Type the command-line: `curl -get` with the same URL above.

[6] This feature is under development at this stage but will be provided within the next month.
[7] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP2/selfmanagement/sservices/authorization`

4.3.2 Security monitoring

4.3.2.1 Objective

The security monitoring component implements self-protection of U-Cloud resources, to detect and react to threats to the computing infrastructure in an autonomous manner. Two aspects of self-protection should be considered: *cross-layer defense* (vertical orchestration) and *cross-provider defense* (horizontal orchestration). We present here a preliminary version of the monitoring component focusing on cross-layer self-protection. A more extensive version will be described in deliverable D2.4.

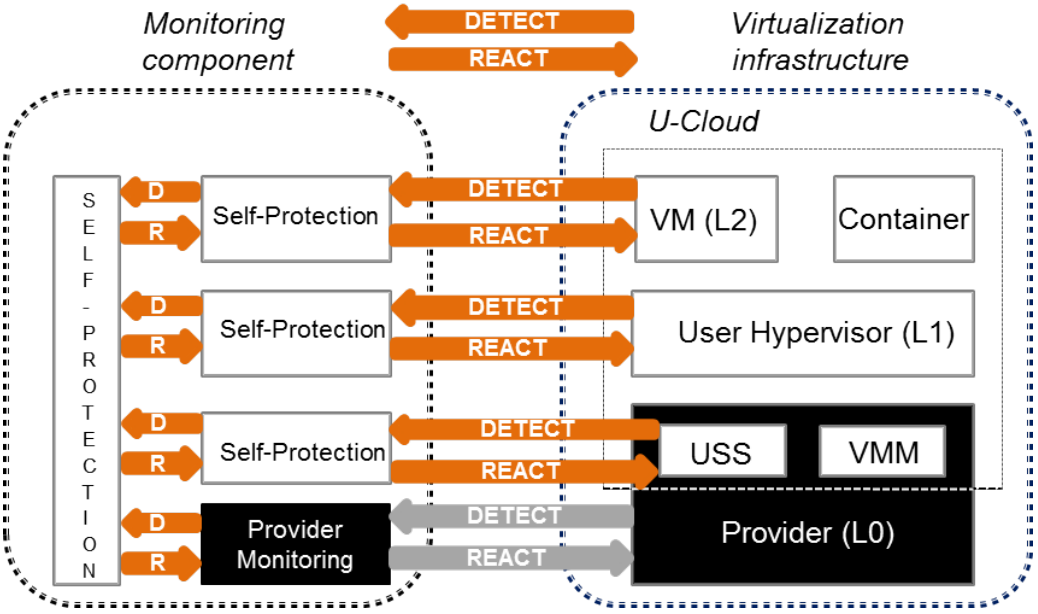

Figure 4.7: Security monitoring: approach

The security monitoring approach was presented in Deliverable D2.1 [19] and is summarized in Figure 4.7. In terms of design, the monitoring component is well-separated from the virtualization infrastructure. Its different components may then be flexibly embedded in the virtualization infrastructure at deployment time. Monitoring relies on orchestration of two hierarchical autonomic security loops.

- The first level manages *intra-layer security monitoring* in user- or provider-controlled parts of the virtualized infrastructure.
- The second level manages *cross-layer security monitoring*, also integrating monitoring information and counter-measures from the cloud provider.

The general design of the component is based on the VESPA framework [33] for the system model and on the OpenStack Watcher [27] framework for external APIs. We chose VESPA as it already implements a first two-level autonomic security monitoring model, but with a very basic API. We selected Watcher, as it is already integrated with OpenStack and provides a very rich monitoring API.

4.3.2.2 Design

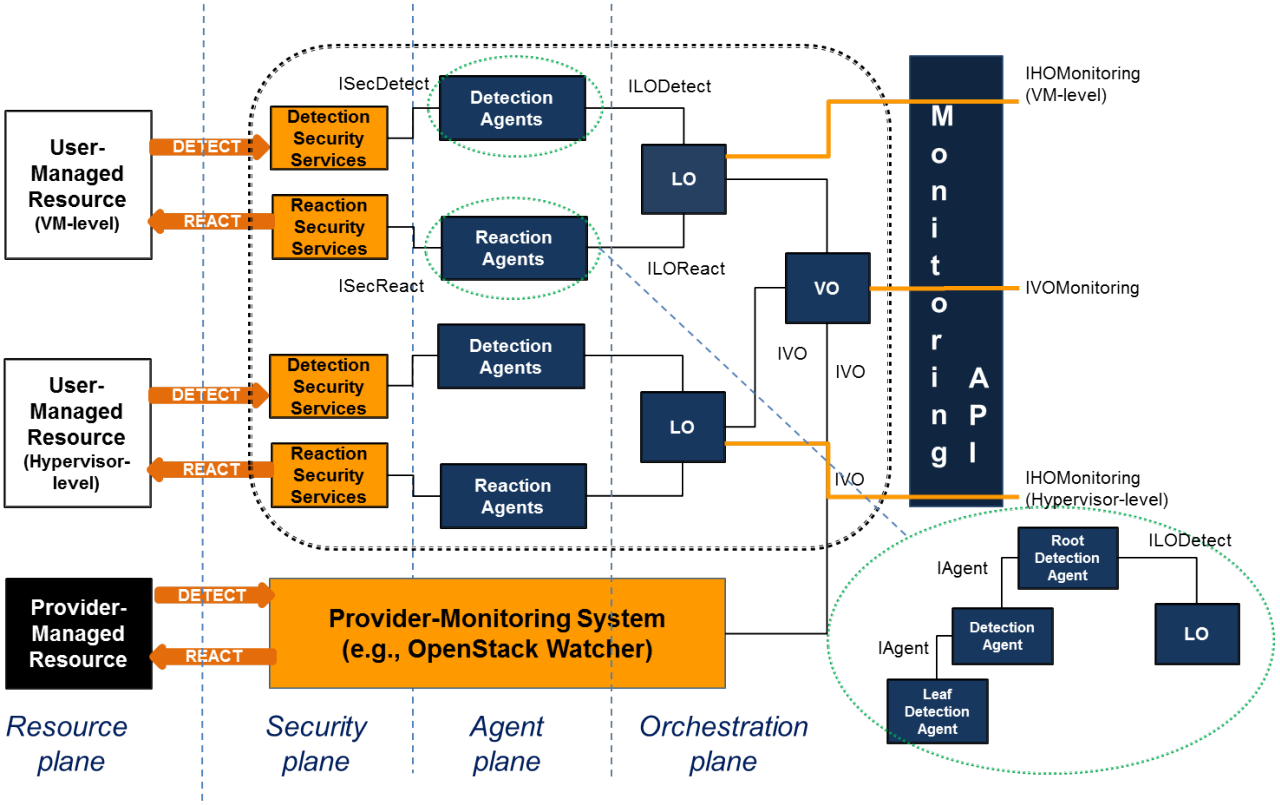

Figure 4.8: Security monitoring: system model

The system model of the monitoring component is shown in Figure 4.8. This design is based on several planes to make the monitoring component more modular, clearly separating resources, detection and reaction mechanisms that may be off-the-shelf, and the decision-making logic:

- The *Resource Plane* contains managed resources to be monitored and protected. Such resources may be located in VMs, containers, or in one or several nested hypervisors.
- The *Security Plane* contains security mechanisms for detection and reaction, which may be arbitrarily added due to the openness of the design.
- The *Agent Plane* contains a mediation layer between security mechanisms and the decision logic through two hierarchies of agents for detection and reaction that may be deployed very flexibly throughout the multi-cloud infrastructure.
- The *Orchestration Plane* contains the decision logic to trigger reactions based on detected threats, implementing the two previous nested autonomic loops. A *Layer Orchestrator (LO)* is an autonomic security manager supervising layer-level monitoring, e.g., at VM or hypervisor levels. The *Vertical Orchestrator (VO)* autonomic security manager oversees cross-layer monitoring, also interfacing with the provider-level monitoring system. This plane also contains an API component to connect the monitoring system with the outside world. More information on such a system design may be found in [34].

4.3.2.3 Entities and internal interfaces

The main entities in the system model are the following.

Security mechanisms. These detection and reaction mechanisms respectively detect unusual behaviors over managed resources (*DSS: Detection Security Services*) or enable to apply security counter-measures (*RSS: Reaction Security Services*). A DSS may be security-related (e.g., IDS, anti-virus, etc.) or exploit external data sources (Ceilometer [10], Monasca [23]) for security monitoring. A RSS counter-measure may be applied at VM-level (e.g., anti-virus), hypervisor-level (e.g., VM migration), or hardware level. The corresponding interfaces (`ISecDetec`, `ISecReact`) are security mechanism-specific. They contain methods to collect events or to apply security rules over a managed resource respectively.

Leaf Agents. These agents are the interface of the mediation layer with the security plane, both for detection (*LDA: Leaf Detection Agent*) and for reaction (*LRA: Leaf Reaction Agent*). Those agents translate security-specific interfaces to higher-level interfaces to collect alarms or to enforce reactions to resources. Such agents implement the interfaces to the Security Plane (`ISecDetec`, `ISecReact`). They also include a standard agent interface `IAgent` to push alarms to upper-level agents or to enforce reaction rules into lower-level agents.

Agents. These intermediate agents correlate alerts for detection, and refine chosen security reactions closer to the infrastructure. They implement the standard agent interface based on an event-based messaging system to push received alerts to upper-levels and send messages back down.

Root Agents. Those agents form the root of agent hierarchies for detection (*RDA: Root Detection Agent*) and for reaction (*RRA: Root Reaction Agent*). They respectively capture the overall security context and the chosen reaction to trigger the mitigation of an attack. As all agents, they implement the standard agent interface, with additional methods when interfacing towards decision-making components (LO), e.g., an `alert` method that contains the decision-making behavior to trigger in the Orchestration Plane.

LO. Those autonomic managers are in charge of layer-level decision-making to choose a reaction plan depending on the gathered security context. This plan may be chosen depending on a security strategy (`Strategy`) or to reach a security goal (`Goal`). The reaction plan contains a number of actions and is then passed down to the RRA for enforcement down to the infrastructure. This component may also delegate actions to the upper-level (VO) for further analysis and decision-making. LOs implement the standard agent interface towards lower agents, with additional methods towards the VO (`alert` method). This autonomic manager also provides an external REST API to provide part of the security context to other components, to get attributes regarding the VM state, the hypervisor state, to enforce a given reaction policy, to set security goals, etc.

VO. This autonomic manager is in charge of cross-layer decision-making. It aggregates events from LOs, or from the provider monitoring infrastructure, and takes decisions for enforcement, which are then applied into layers by LOs. The decision-making logic is similar to that of LOs in terms of security goals, actions, etc. The VO has thus similar interfaces to those of an LO. It also includes a REST API that forms the overall cross-layer monitoring API for the infrastructure.

4.3.2.4 External interface

The REST API of the monitoring component is shown in Table 4.2. It captures the overall monitoring interfaces, either at LO-level (VM-level monitoring or hypervisor-level monitoring) or VO-level (cross-layer level monitoring). Examples inspired from the Watcher API [27] are provided in the table below. Most of those requests may be called either at VO level (`IVOMonitoring`) or LO level (`ILOMonitoring-VM`, `ILOMonitoring-Hypervisor`) to retrieve cross-layer, or VM/hypervisor-layer security context information respectively. The corresponding API component is under implementation.

Request	Description	Parameters	Response
`GET URL/SecurityContext?`	Retrieves information on a given attribute of the monitored security context.	`attribute=a`	Attribute value
`GET URL/SecurityStrategy?`	Retrieves in the security strategy action plans mapped to a given security context.	`sec-context=c`	List of action plans
`PATCH URL/UpdateSecurityStrategy?`	Updates entries in the security strategy, adding a security context-to-action plan mapping. Similar APIs may be included to update information in action plans,actions, or security goals.	`sec-strategy-uuid=uuid& sec-context=sc& action-plan=ap`	Security strategy
`GET URL/ComputeSecurityStrategy?`	Computes a security strategy given a security goal and additional parameters.	`sec-goal=sg& params=p`	Security strategy
`GET URL/TriggerActionPlan?`	Forces execution of the reaction planned by the current security strategy given a provided security context. Forcing a reaction may be used to interface with third-party security monitoring/reaction systems.	`sec-strategy=ss& sec-context=sc`	Boolean

Table 4.2: Monitoring component API

Cross-domain monitoring. The monitoring component may be integrated with the ORBITS framework to achieve horizontal orchestration, independently from the provider, reconciling different VOs, according to different patterns (P2P, hierarchical, hybrid), as discussed in D2.1. More details may be found in [29]. A more detailed design will be described in D2.4.

4.3.2.5 Component access

The monitoring component is accessible at `https://github.com/Orange-OpenSource/vespa-core`.

4.3.3 Geolocation-aware replication

In this Section, we present the SUPERCLOUD geolocation data replication component. The functionality of this service was introduced in Deliverable D2.2 [22].

4.3.3.1 Objective

The main objective of the geolocation-aware data replication component is to replicate data only in allowed locations and therefore address different geolocation directives, regulations or requirements that limit the options for the places where the data can be replicated. This situation is further complicated by relying on virtual machines (VMs) provided locally or by multiple cloud providers. Deployment of new services and requirements like availability or backup procedures cause the configuration of these VMs to be dynamic over time.

The component is using a cryptographic protocol for discovering VMs that satisfy the existent geolocation requirements. The geolocation requirements are enforced by authenticating the VMs using geolocation attributes. The component uses a single transmission multicast of encrypted data after a single pass discovery broadcast. This solution provides an efficient, adaptable and decentralized discovery of replication VMs that satisfy existent geolocation constraints. Furthermore, the component is platform independent, allows easy integration across heterogeneous systems, and lowers the needed amount of trust in the resource/VMs providers (e.g. cloud).

4.3.3.2 Entities and internal interface

The geolocation-aware data replication component is implemented in Java and instances of this component are deployed in VMs where data are stored. These geolocation instances communicate between each other using the TCP/IP protocol by writing in and reading from connecting sockets. This communication is mapped on the protocol designed in Deliverable D2.2 [22], as depicted in Figure 4.9. Given that this component is a `war` file, it can be exposed via a Docker container for easy integration. A load balancer would decide when replication is needed and would ask for a list of available locations for replication (e.g. using the "replCandidates" request). Internally the protocol is executed in a straightforward manner according to the following three steps:

1. The origin broadcasts the message: $Policy, E_{Policy}(sessionKey)$
2. The targets that are able to decrypt the broadcast message will answer the challenge with the message $(src, dst, Policy)$ encrypted with the $sessionKey$: $E_{sessionKey}(src, dst, Policy)$
3. The origin selects one or more VMs where the data will be replicated. Next, the origin sends the sensitive data, encrypted with the session key proposed in the beginning. Therefore, the message sent to the replication VMs is: $E_{sessionKey}(DATA)$

The policies are received from the "Authorization security service".

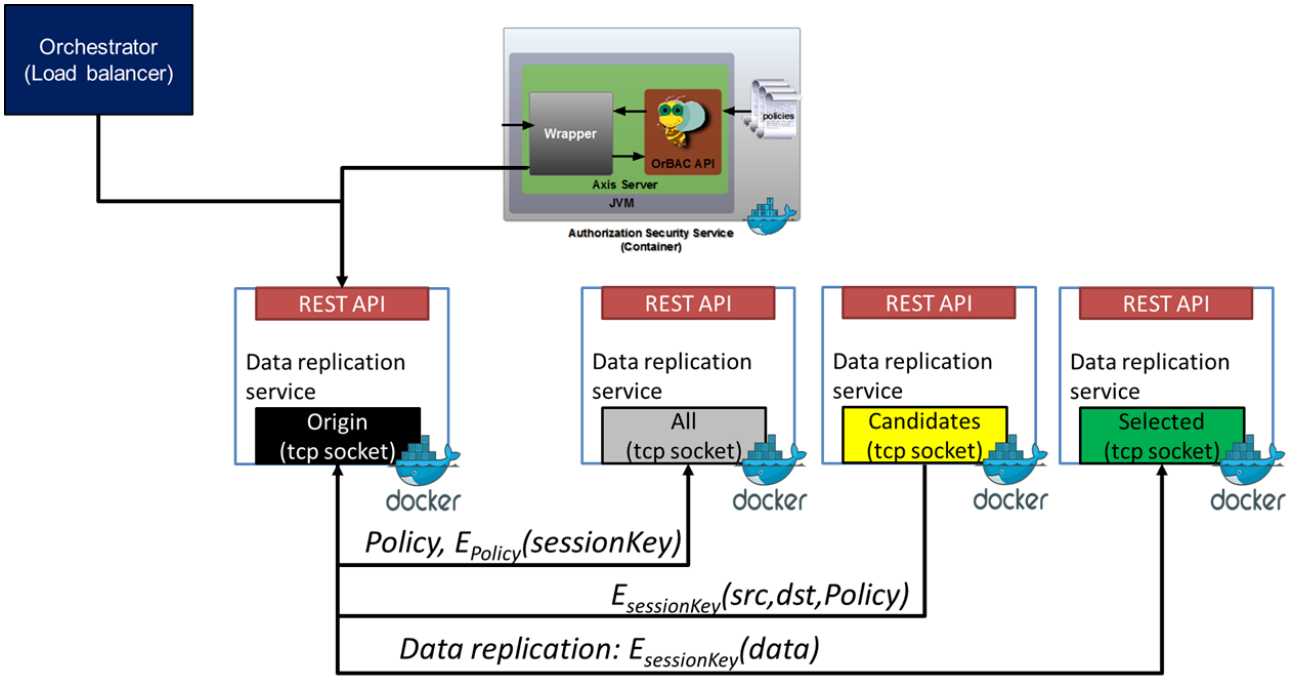

Figure 4.9: Communication between geolocation instances

4.3.3.3 External interface

The geolocation data replication component can be called from the outside using the REST APIs shown in Table 4.3.

Request	Description	Parameters	Response
`GET URL/replCandidates?`	Requests replication of data, where the identifier of the data block is given as parameter.	`dataId=did`	list of candidates
`GET URL/repl`	Requests the data to be replicated in one or more of the locations selected from the list received as response in the "replCandidates" request.	`dataId=did`	Boolean
`POST URL/upload`	Request used for uploading data to the VM where this component is deployed. This can be done internally, within the VM where the owner of the data stores it. This can also be done externally, this being needed when the data is initially uploaded to the cloud.	data content	Boolean
`POST URL/setPolicy`	Request that uploads a policy to be attached to a portion of the data. The portion of the data is given as ID. The data is replicated in one of the locations selected from the list received as response in the "replCandidates" request.	`dataId=did`	Boolean

Table 4.3: Geolocation data replication component API

4.3.3.4 Component access

The geolocation data replication component will be available on the SUPERCLOUD private repository[8] once the way this component can be shared is concluded. Instructions for installation and further documentation about the software are also distributed together with the code release. The deployment process has been tested in Linux.

4.3.4 SLA management

Security Service Level Objectives (SSLAs) outline a commitment between Cloud Service Customers (CSC) and Cloud Service Providers (CSP). As discussed in Deliverable D1.2 [37], the management of SSLAs involve five main phases: specification, negotiation, enforcement, monitoring and arbitration. In this Section, we make a focus on the monitoring and arbitration phases. The implementation of the overall SSLA life-cycle is described in detail in Deliverable D1.4.

4.3.4.1 Objective

The objective of this service is to provide the Cloud Service Customer (CSC) with mechanisms to supervise the execution of the active SSLA. Thus the service will process the SSLA upheld by the CSC and the CSP and extract the metrics to be monitored (Performance and Security Level Objectives). As illustrated in Figure 4.10, the SSLA Enforcement Service (SES) translates and maps low-level raw resource metrics measured by monitoring services to high-level SSLA objectives. In our implementation, we assume that raw data is stored by the monitoring service within a dedicated database provided by the storage service. Table 4.4 provides examples of raw metrics that the monitoring service can collect from the provider infrastructure.
The above metrics are subsequently processed to be mapped to concrete Service Level Objectives as specified in SSLAs. For instance, `upTime` and `downTime` are mapped to availability for both Compute, Storage and Network as follows:

$$Availability = 1 - \frac{\texttt{upTime}}{\texttt{downTime}} \tag{4.1}$$

[8] `https://github.com/H2020-SUPERCLOUD/`

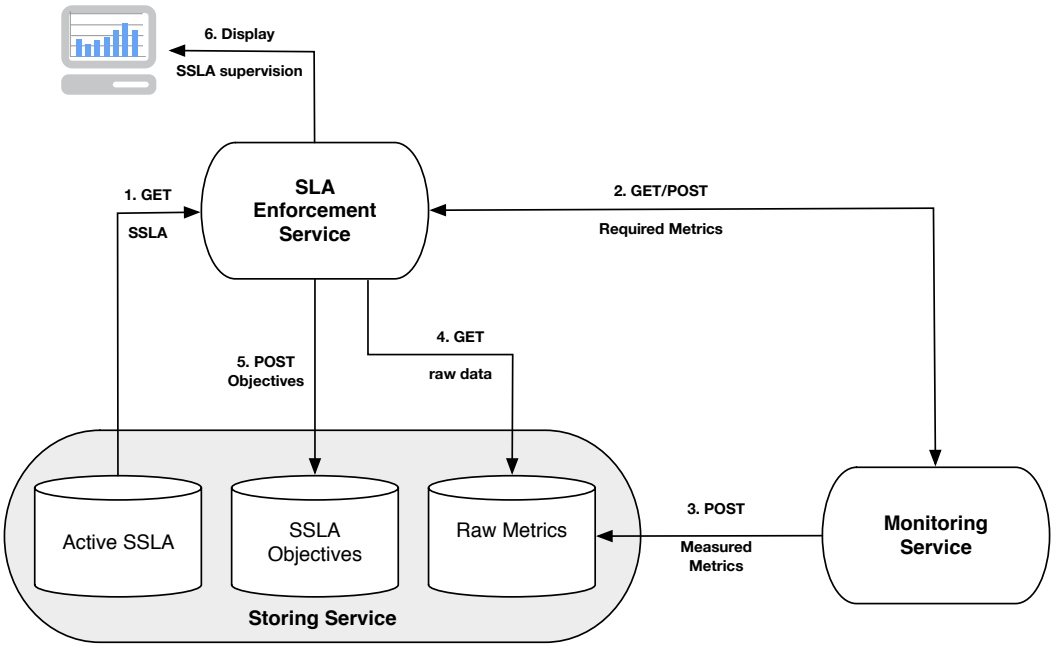

Figure 4.10: SSLA Enforcement Service

Layer	Metrics	Frequency (sec)
Compute	`memTotal`, `memUsed`, `memFree`, `cpuIdle`, `cpuUse`, `upTime`, `DownTime`	±10
Storage	`diskTotal`, `diskFree`, `diskUsed`, `Encrytion`, `Location`, `upTime`, `DownTime`	±60
Network	`netPacketsIn`, `netPacketsOut`, `netBytesIn`, `netBytesOut`, `upTime`, `DownTime`	±30

Table 4.4: Example of monitoring metrics

Here, `downTime` refers to the time required to bring a service (Compute, Storage or Network) to work after a failure, while `upTime` represents the sum of time without failure. Following this schema, the SSLA Enforcement Service extracts, periodically, relevant information to compute statistics that reflect the fulfillment of each Protection/Security Level Objective. This information is then displayed graphically using charts as shown in Figure 4.11.

4.3.4.2 External interface

As presented in Section 4.2.1, SLA enforcement, storage and monitoring services are provided as Docker images that are deployed and configured by the Security Orchestrator. Consequently, the coordination of their execution, and most specifically their interaction is achieved using standard REST API methods GET and POST as illustrated in Figure 4.10. The complete SSLA Service API is part of the Self-Management Security Architecture and will be presented in Deliverable D1.4.

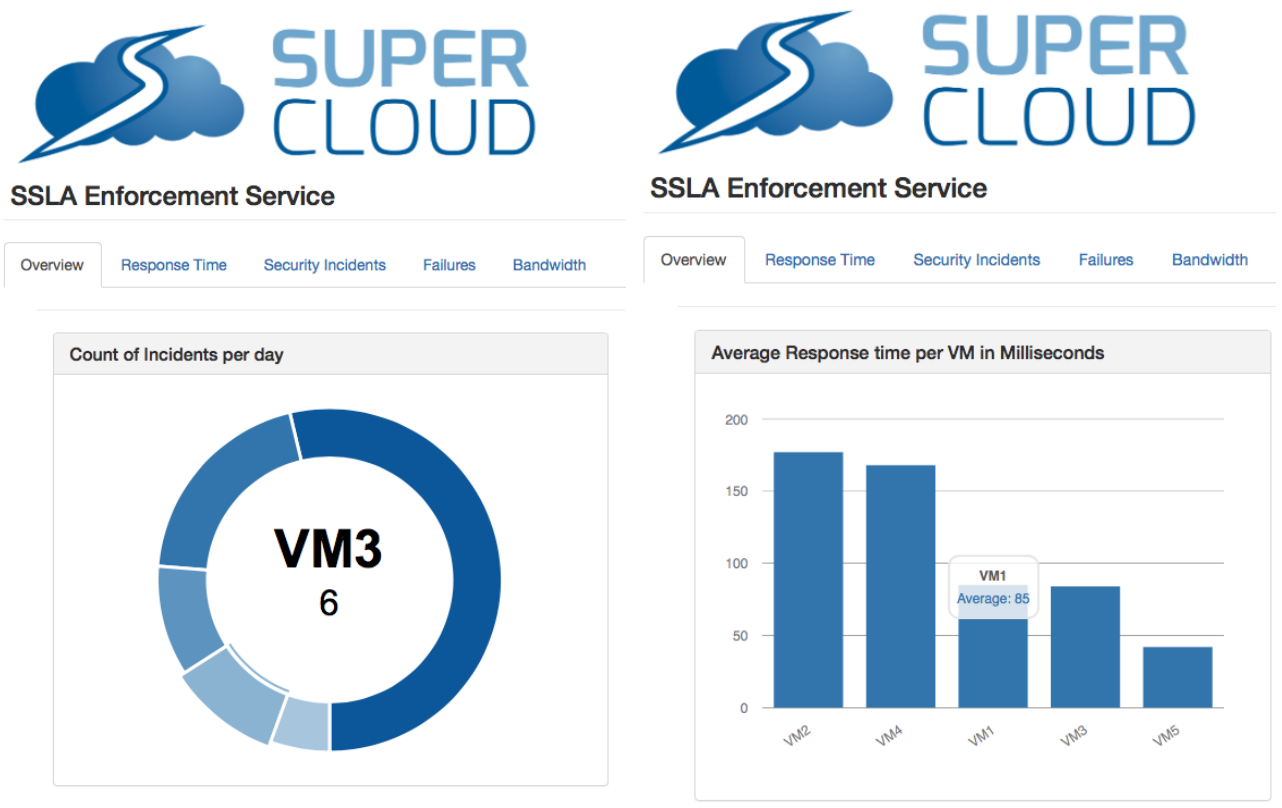

Figure 4.11: Captures from the SSLA reporting dashboard

4.3.4.3 Component access

In this Deliverable, we provide the SSLA enforcement component that extracts SSLA objectives to be presented to the Cloud Customer. This component is part of the SSLA orchestrator that manages requirements specification, negotiation and enforcement. In what follows, we describe the procedure to retrieve and deploy the SSLA Enforcement Web Service. As stated before, the service is available as a deploy-ready Docker image. It contains the web application responsible for displaying the information collected by the monitoring service and aggregated by the SSLA engine.
We present hereafter the procedure to download, deploy and test the services on the user's machine[9].

- Download the tar compressed file containing the service Docker image[10].
- Launch the `./start.sh` script to start the SSLA Enforcement Service.
- To test if the service is running, go to the URL address : `http://localhost:80`
- This component displays the fulfillment of SSLA based on metrics from the storage service. The engine that computes these values and stores them into the database is part of the SSLA life-cycle presented in Deliverable D1.4.

[9] The following deployment process has been tested in Linux and Mac OS.
[10] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP2/selfmanagement/sservices/sla`

4.3.5 Software trust

In the cloud ecosystem, Security Service Level Agreements (SSLAs) are considered as a good trust enabler as they represent a legal document that certifies the provider's willingness to meet the customer's expected Quality-of-Service (QoS) and Quality-of-Protection (QoP) [18]. From the cloud provider perspective, QoS or QoP metrics respectively provide a good indicator of the infrastructure capacities, while the same metrics from the customer perspective testify about the performance experienced by the cloud customer.
The processing of QoS and QoP metrics usually includes a customers × providers experiences matrix, as shown hereafter. Each matrix represents the experiences that all customers of the system ($\forall c \in \mathcal{C}$) had with cloud providers (i.e., $\forall p \in \mathcal{P}$) for a particular cloud service $s \in \mathcal{S}$. Rows represent experiences issued by a certain customer, while columns reflect the experiences expressed with respect to a particular provider.

$$\mathcal{E}^s_{\mathcal{C}:\mathcal{P}} = \begin{array}{c} \\ c_1 \\ c_2 \\ \vdots \\ c_y \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & \cdots & p_n \end{array} \\ \begin{pmatrix} \mathcal{E}^s_{c_1:p_1} & \mathcal{E}^s_{c_1:p_2} & \cdots & \mathcal{E}^s_{c_1:p_y} \\ \mathcal{E}^s_{c_2:p_1} & \mathcal{E}^s_{c_2:p_2} & \cdots & \mathcal{E}^s_{c_2:p_y} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{E}^s_{c_x:p_1} & \mathcal{E}^s_{c_x:p_2} & \cdots & \mathcal{E}^s_{c_x:p_y} \end{pmatrix} \end{array} \tag{4.2}$$

In this Section, we present the *Software Trust Service (STS)* that generates Cloud Customers Experiences based on monitored information.

4.3.5.1 Objective

The objective of the Software Trust Service is to assist cloud customers and providers selecting the best interacting partners within the Cloud Market Place (CMP).
A CMP System Model can be defined at a time t by:

$$CMP = \langle \mathcal{C}, \mathcal{Q}, \mathcal{P}, \mathcal{O}, \mathcal{S}, \mathcal{M}, \mathcal{A}, \mathcal{E} \rangle^t \tag{4.3}$$

where $\mathcal{C} = \{c1, c_2, \ldots\}$ is the set of customers, $\mathcal{Q} = \{q_1, q_2, \ldots\}$ is the set of queries, $\mathcal{P} = \{p_1, p_2, \ldots, p_l\}$ is the set of providers, $\mathcal{O} = \{o_1, o_2, \ldots\}$ is the set of offers, $\mathcal{S} = \{s_1, s_2, \ldots\}$ is the set of cloud services, $\mathcal{M} = \{m_1, m_2, \ldots\}$ is the set of multi-clouds, $\mathcal{A} = \{a_1, a_2, \ldots\}$ is the set of agreements, $\mathcal{E} = \{e_1, e_2, \ldots\}$ is the set of customer experiences. In what follows, we make use of c, q, p, o, s, m, a, e to refer to, respectively, an arbitrary customer, query, provider, offer, service, multi-cloud, agreement and experience.

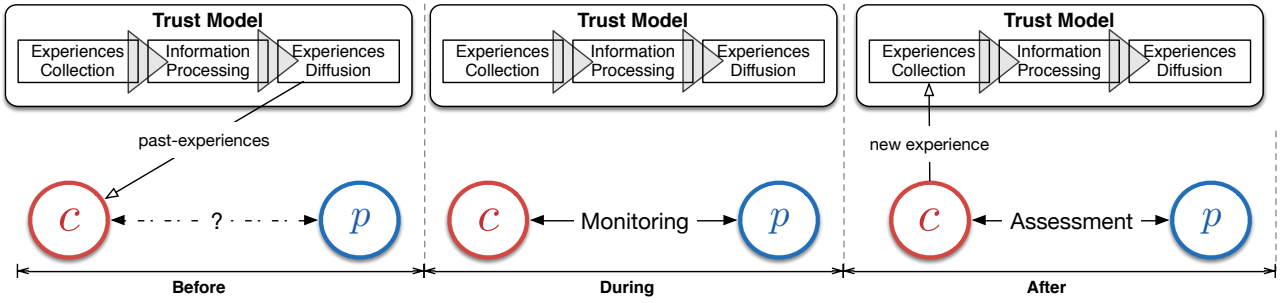

Figure 4.12: Classical feedback-based trust assessment

The trust that a cloud customer c is willing to put into a cloud provider p depends on the experience derived from past experiences. The experiences constitute customers' and providers' feedback and reflect their level of satisfaction with respect to the expected quality of service and protection. This experience information is thus processed by the Trust Service for assessment of the trust that a customer can put in the candidate provider. As illustrated in Figure 4.12, before making a decision about the provider to engage with, for a specific service s, the requesting customer c will make use of the SUPERCLOUD Trust Management Service (see Deliverable D1.4) to derive a trust value based on past-experiences. Then, during the transaction (i.e., Cloud Service Delivery), the CSC c will make use of monitoring mechanisms to observe the behavior of the provider. We make the reasonable assumption that all Service Level Objectives (SLOs) conveyed in an SSLA agreement (i.e., $a \in \mathcal{A}$) can be monitored and that monitoring information is reliable and cannot be tampered with.

Experiences representation. We denote $\mathcal{E}^s_{c:p} \subseteq \mathcal{E}$ the chronologically ordered set of experiences issued by the customer c towards the provider p for a specific delivered service s. Each experience $e_i^{\langle c,p,s\rangle} \in \mathcal{E}^s_{c:p}$ is stored in the system as a quintuplet [11]:

$$e_i^{\langle c,p,s\rangle} = \langle c, p, s, \upsilon, t\rangle \tag{4.4}$$

$e_i^{\langle c,p,s\rangle}$ maps the services provided by p towards a customer c at a time t to a fulfillment level υ. For simplicity, we make use of a normalized rating scale of $[0, 1]$. For instance, in the following experience example $\langle c, p, availability, 0.9995, t\rangle$, the monitored value $99,95\%$ (corresponds to the QoS Level Objective that represents availability of cloud services) is mapped to the normalized 0.9995 value. We assume that categorical values are mapped to $true$ if the service level is met and $false$ if not. These values are then converted to, respectively, 0 and 1.

Experiences aggregation. The aggregation of individual and collective experiences constitute a reputation value. Computing the reputation of a set of providers (i.e., multi-cloud) is known in the literature as *group reputation* [6]. Few works tried to address it and no real consensus exists about how to obtain it. We were inspired from the Simple Additive Weighting approach [15] and propose to proceed in two steps:

1. First, we compute the individual reputation of each provider. The schema used is identical for both customers and providers.
2. Then the computed individual reputation values are combined to compute a collective reputation for the candidate multi-cloud.

The general formula used to aggregate experiences into reputation values is defined as follows::

$$R^s_{c:p} = \frac{\sum_{i=1}^{\lambda} \left[(e_i^{\langle c,p,s\rangle}).\upsilon\right]}{\lambda} \tag{4.5}$$

where : $\forall e_i, e_j \in \mathcal{E}^s_{c:p} \mid ij \implies e_i.t \geq e_j.t$

In Equation 4.5, the reputation built based on the experiences of a customer c towards a provider p for a service s is the weighted sum of the fulfillment levels υ. We make use of the constant λ to express how fast the reputation value of the provider changes after each experience. The larger the value of λ, the longer the memory of the system is. In other words, the constant λ reflects the willingness of a customer to forgive past negative experiences [38]. It avoids that providers suffer too much from their initial poor behavior which may sentence all the system, as very few interactions would be possible. Thus only the λ last experiences witnessed are used to compute the reputation.

[11] Experiences are issued by both providers and customers. The same format applies, indistinguishably to both types.

The benefits of the Software Trust Service were demonstrated experimentally and its results presented in a conference paper [36]. These results will be summarized in the Appendix of Deliverable D1.4.

4.3.5.2 External interface

In this Section, we present the integration of the software trust model within the SUPERCLOUD computing framework, and more specifically within the self-management of security . The Software Trust Service builds on the top of the SSLA enforcement and monitoring service to make trust assessments (see Figure 4.13). The trust computation component mainly integrates the algorithm that processes the SSLA objectives and derives trustworthiness values as described in [36]. These values are made available via a standard REST API interface.

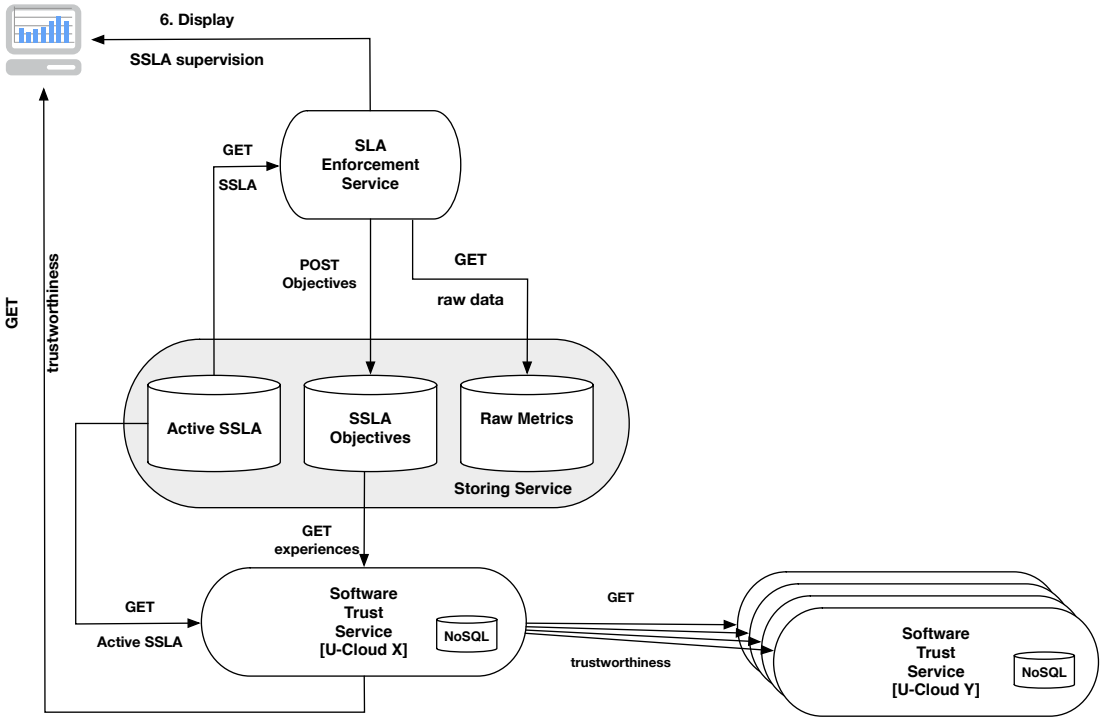

Figure 4.13: SSLA Enforcement Service

4.3.5.3 APIs

The Software Trust Service REST API interface is split into two parts, internal and external.

- The *internal interface* interacts with other SUPERCLOUD services. These interactions are orchestrated by the Security Orchestrator as described in Section 4.2.1.
- The *external interface* is used to communicate with other Software Trust Services to exchange experiences and trust values. This interface is mandatory when computing Cloud Marketplace level Trust Metrics (i.e., reputation).

Table 4.5 summarizes the methods and parameters of the service interfaces.

Request	Description	Parameters	Response
`GET URL/trust`	Retrieves all trust values	`none`	A list of [Provider,trust]
`GET URL/getTrust`	Retrieves the trustworthiness a provider	`provider=id`	$\in [0, 1]$
`GET URL/getMCTrust`	Retrieves the trustworthiness a multi-cloud	`provider=id1&...`	$\in [0, 1]$
`GET URL/getReputation`	Retrieves the reputation of a provider	`provider=id`	$\in [0, 1]$

Table 4.5: Software Trust service external interfaces

4.3.5.4 Component access

This component is part of a complex trust management framework implemented in the context of Deliverable D1.4. Consequently, the component will be released as part of this framework. Nevertheless, we present here after the procedure to retrieve and deploy the Software Trust Service[12]. Like other self-management of security services, the Trust Service is available as a deploy-ready Docker image.

- Download the tar compressed file containing the service Docker image[13].
- Launch the `./startSTS.sh` script to start the Software Trust Service.
- Once the trust service starts, it can be accessed thought standard REST calls. For instance, `curl -XGET ://192.168.99.100:8000/trust` will retrieve the trust values from the service.

[12] The following deployment process has been tested on Mac OS, but tests have been conducted to verify that the services would run similarly on any other operating system.

[13] `https://github.com/H2020-SUPERCLOUD/SUPERCLOUD-FW/tree/master/WP2/selfmanagement/sservices/trust/`

Chapter 5 Conclusions

This document is part of the deliverable that presents the proof-of-concept implementation of the distributed cloud infrastructure for computation and components for SUPERCLOUD computing security management. Our purpose was to describe the structure of the computing framework, the APIs of its main components, and to provide information on how to access and run the software developed.
We first presented the approach for specifying the framework, giving an overview of its structure, composed of two separate sub-infrastructures, the *virtualization infrastructure* and the *self-management infrastructure*.
We then presented the two sub-infrastructures, in terms of structure and APIs of their components. Namely:

- For the virtualization infrastructure: a virtualization and orchestration component and a micro-hypervisor, isolation and trust management components and support for Cloud FPGAs;
- For the self-management infrastructure: a security orchestrator, and a number of security services covering authorization, security monitoring, geolocation-aware replication, SLA management, and software trust management.

The next steps will be devoted to: (1) refining the structure of the self-management infrastructure (WP2) specified in Deliverable D2.4; integrating those components into the overall SUPERCLOUD frameworks (WP1), specified in Deliverables D1.3 and D1.4, in connection with data management (WP3) and networking (WP4) components, and deploying those components in the demonstrated use cases of the project (WP5).

List of Abbreviations

AC	Access Control
ACPI	Advanced Configuration and Power Interface
AHCI	Advanced Host Controller Interface
AOP	Aspect-Oriented Programming
API	Application Programming Interface
ARP	Address Resolution Protocol
BFT	Byzantine Fault Tolerance
CMP	Cloud Market Place
CoT	Chain of Trust
CPU	Central Processing Unit
CSC	Cloud Service Customer
CSP	Cloud Service Provider
DBMS	Database Management System
DP	Design Principle
DR	Design Requirement
DSS	Detection Security Services
EE	Execution Environment
FPGA	Field-Programmable Gate Array
HOT	Heat Orchestration Language
HTTP	Hypertext Transfer Protocol
HW	Hardware
IaaS	Infrastructure-as-a-Service
IaC	Infrastructure as Code
I/O	Input / Output
IP	Internet Protocol
JSON	JavaScript Object Notation
LDA	Leaf Detection Agent
LO	Layer Orchestrator
LR	Left-to-Right
LRA	Leaf Reaction Agent

MAC	Message Authentication Code
NIC	Network Interface Card
ORBITS	ORchestration for Beyond InTer-cloud Security
OS	Operating System
PCI	Peripheral Component Interconnect
PDP	Policy Decision Point
PEP	Policy Enforcement Point
QoP	Quality of Protection
QoS	Quality of Service
REST	Representational State Transfer
RPC	Remote Procedure Call
RDA	Root Detection Agent
RRA	Root Reaction Agent
RSA	Rivest-Shamir-Adleman
RSS	Reaction Security Services
RTC	Real-Time Clock
SDK	Software Development Kit
SDN	Software-Defined Networking
SES	SSLA Enforcement Service
SGX	Software Guard eXtensions
SLA	Service Level Agreement
SLO	Service Level Objective
SMR	State Machine Replication
SOA	Service Oriented Architecture
SSH	Secure Shell
SSLA	Security Service Level Agreement
STS	Software Trust Service
SW	Software
TCP	Transmission Control Protocol
TML	TOSCA Manipulation Language
TOSCA	Topology and Orchestration Specification for Cloud Applications
U-Cloud	User Cloud
URL	Uniform Resource Locator
USS	User-Centric Security Service
VM	Virtual Machine
VO	Vertical Orchestrator
VMM	Virtual Machine Monitor
VPN	Virtual Private Network
XML	Extensible Markup Language

Bibliography

[1] Intel SGX repository. URL: https://github.com/sslab-gatech/opensgx.

[2] Intel Software Guard Extensions SDK. URL: https://software.intel.com/en-us/sgx-sdk.

[3] Kubernetes. URL: http://kubernetes.io/.

[4] Docker compose tool. 2016. URL: https://docs.docker.com/compose/.

[5] Docker engine swarm mode. 2017. URL: https://docs.docker.com/engine/swarm/.

[6] B. Baranski, T. Bartz-Beielstein, R. Ehlers, T. Kajendran, B. Kosslers, J. Mehnen, T. Polaszek, R. Reimholz, J. Schmidt, K. Schmitt, D. Seis, R. Slodzinski, S. Steeg, N. Wiemann, and M. Zimmermann. The impact of group reputation in multiagent environments. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1224–1231, July 2006. doi:10.1109/CEC.2006.1688449.

[7] T. Binz, G. Breiter, F. Leyman, and T. Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing*, 16(3):80–85, May 2012.

[8] Antonio Brogi and Jacopo Soldani. Matching Cloud Services with TOSCA. In *European Conference on Service-Oriented and Cloud Computing (ESOCC) Workshops*, 2013.

[9] Antonio Brogi, Jacopo Soldani, and PengWei Wang. TOSCA in a Nutshell: Promises and Perspectives. In *Third European Conference on Service-Oriented and Cloud Computing (ESOCC)*, 2014.

[10] Ceilometer. URL: https://github.com/openstack/ceilometer.

[11] Docker Swarm. URL: https://docs.docker.com/swarm/.

[12] Alex Fishman, Mike Rapoport, Evgeny Budilovsky, and Izik Eidus. HVX: Virtualizing the Cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2013.

[13] Genode Operating System Framework. URL: https://genode.org/.

[14] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. OpenSGX: An Open Platform for SGX Research. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[15] I. Kaliszewski and D. Podkopaev. Simple Additive Weighting – A Metamodel for Multiple Criteria Decision Analysis Methods. *Expert Syst. Appl.*, 54(C):155–161, July 2016. URL: http://dx.doi.org/10.1016/j.eswa.2016.01.042, doi:10.1016/j.eswa.2016.01.042.

[16] Houssem Kanzari and Marc Lacoste. Towards Management of Chains of Trust for Multi-Clouds with Intel SGX. In *Second ComPAS Workshop on Security in Clouds (SEC2)*, 2016.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[18] Marc Lacoste, Markus Miettinen, Nuno Neves, Fernando M. V. Ramos, Marko Vukolic, Fabien Charmet, Reda Yaich, Krzysztof Oborzynski, Gitesh Vernekar, and Paulo Sousa. User-Centric Security and Dependability in the Clouds-of-Clouds. *IEEE Cloud Computing*, 3(5):64–75, 2016.

[19] Marc Lacoste, Benjamin Walterscheid, Alex Palesandro, Aurélien Wailly, Ruan He, Yvan Rafflé, Jean-Philippe Wary, Yanhuang Li, Sören Bleikertz, Alysson Bessani, Reda Yaich, Sabir Idrees, Nora Cuppens, Frédéric Cuppens, Ferdinand Brasser, Jialin Huang, Majid Sobhani, Krzysztof Oborzynski, Gitesh Vernekar, Meilof Veeningen, and Paulo Sousa. D2.1 - Architecture for Secure Computation Infrastructure and Self-Management of VM Security. *SUPERCLOUD*, 2015. URL: `https://supercloud-project.eu/downloads/SC-D2.1-Secure-Computation-Infrastructure-PU-M09.pdf`.

[20] Mesos. URL: `https://mesos.apache.org/`.

[21] Markus Miettinen, Ferdinand Brasser, Ahmad-Reza Sadeghi, Marc Lacoste, Nizar Kheir, Marko Vukolic, Alysson Bessani, Fernando Ramos, Nuno Neves, and Majid Sobhani. D1.1 - SUPERCLOUD Architecture Specification. *SUPERCLOUD*, 2015. URL: `https://supercloud-project.eu/downloads/SC-D1.1-Architecture-Specification-PU-M10.pdf`.

[22] Markus Miettinen, Mario Münzer, Felix Stornig, Marc Lacoste, Alex Palesandro, Denis Bourge, Charles Henrotte, Houssem Kanzari, Ruan He, Marko Vucolic, Jagath Weerasinghe, Sabir Idrees, Reda Yaich, Nora Cuppens, Frédéric Cuppens, Ferdinand Brasser, Raad Bahmani, Tommaso Frassetto, David Gens, Daniel Pletea, and Peter van Liesdonk. D2.2 - Secure Computation Infrastructure and Self-Management of VM Security. *SUPERCLOUD*, 2016. URL: `https://supercloud-project.eu/downloads/SC-D2.2-Secure-Computation-Infrastructure-and-Self-Management-of-VM-Security-PU-M21.pdf`.

[23] Monasca. URL: `http://monasca.io/`.

[24] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0. URL: `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`.

[25] OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. URL: `http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf`.

[26] OASIS TOSCA. URL: `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`.

[27] OpenStack Watcher. URL: `https://github.com/openstack/watcher`.

[28] Alex Palesandro, Chirine Ghedira Guegan, Marc Lacoste, and Nadia Bennani. Overcoming barriers for ubiquitous user-centric healthcare services. *IEEE Cloud Computing*, 3(6):64–74, 2016. URL: `https://doi.org/10.1109/MCC.2016.131`, `doi:10.1109/MCC.2016.131`.

[29] Alex Palesandro, Marc Lacoste, Nadia Bennani, Chirine Ghedira Guegan, and Denis Bourge. Putting Aspects to Work for Flexible Multi-Cloud Deployment. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[30] K. Razavi, A. Ion, G. Tato, K. Jeong, R. Figueiredo, G. Pierre, and T. Kielmann. Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure. In *IEEE International Conference on Cloud Engineering*, 2015.

[31] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. EuroSys'10.

[32] Clemens Szyperski. *Component Software: Beyond Object-oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[33] VESPA: Virtual Environment Self-Protecting Architecture. URL: `https://github.com/Orange-OpenSource/vespa-core`.

[34] Aurélien Wailly, Marc Lacoste, and Hervé Debar. VESPA: Multi-layered Self-Protection for Cloud Resources. In *International Conference on Autonomic Computing (ICAC)*, 2012.

[35] Andreas Wittig and Michael Wittig. *Amazon Web Services in Action*. Manning Press, 2016.

[36] R. Yaich, N. Cuppens, and F. Cuppens. Enabling Trust Assessment In Clouds-of-Clouds: A Similarity-Based Approach. *In Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2017. `doi:10.1145/3098954.3098970`.

[37] Reda Yaich, Sabir Idrees, Nora Cuppens, Frédéric Cuppens, Marc Lacoste, Nizar Kheir, Ruan He, Khalifa Toumi, Krzysztof Oborzynski, Meilof Veeningen, and Paulo Sousa. D1.2 - SUPERCLOUD Self-Management of Security Specification. *SUPERCLOUD*, 2015. URL: `https://supercloud-project.eu/downloads/SC-D1.2-Self-Management_Security_Specification-PU-M09.pdf`.

[38] Giorgos Zacharia, Alexandros Moukas, and Pattie Maes. Collaborative reputation mechanisms for electronic marketplaces. *Decision Support Systems*, 29(4):371–388, 2000. `doi:10.1016/S0167-9236(00)00084-1`.