DECENTER

Decentralised technologies
for orchestrated Cloud-to-Edge intelligence

D3.3

Second release of the fog computing platform

04/12/2020

Revision history

Administration and summary		
Project acronym:	DECENTER	
Document identifier:	D3.3: Second release of the fog computing platform [M24]	
Leading partner:	ATOS	
Report version:	1.1	
Report preparation date:	04.12.2020	
Classification:	PU (Public)	
Nature:	R (Report)	
Author(s) and contributors:	Ismael Cuadrado Cordero (ATOS) Alejandro García (ATOS) Roi Sucasas Font (ATOS) Alejandro García Bedoya (ATOS) Roberto Doriguzzi-Corin (FBK) Silvio Cretti (FBK) Raffaele Giaffreda (FBK) Seungwoo Kum (KETI) Uroš Paščinski (UL) Petar Kochovski (UL) Vlado Stankovski (UL) Christophe Munilla (KENT) Soonho Lee (DW)	
Status:	-	Plan
	-	Draft
	X	Final

The DECENTER Consortium has addressed all comments received, making changes as necessary. Changes to this document are detailed in the change log table below.

Date	Edited by	Status	Changes made
04.03.2020	Ismael Cuadrado-Cordero	Plan	ToC
18.03.2020	Ismael Cuadrado-Cordero	Draft	First version gathered from MS9
11.04.2020	Klaus Kim	Draft	Section on Monitoring System (6.3)
30.04.2020	Silvio Cretti	Draft	Sections on platform implementation advances (Section 3 and Annex A and B)
15.05.2020	Roberto Doriguzzi-Corin	Draft	Sections on platform implementation advances (Section 3)
18.05.2020	Seungwoo Kum	Draft	Sections on AI support advances (Section 4)
20.05.2020	Raffaele Giaffreda	Draft	Section on Resource Brokering advances (Section 7)
02.06.2020	Alejandro Garcia Bedoya	Draft	Section on Security (Section 8)
03.06.2020	Ismael Cuadrado-Cordero	Draft	Integration of work
05.06.2020	Ismael Cuadrado-Cordero	Draft	Initial overall review
15.06.2020	Uroš Paščinski	Draft	Added the second part of section 9
16.06.2020	Petar Kochovski	Draft	Section 9
18.06.2020	Christophe Munilla	Draft	Improvements Sections 2 & 3.1.3
19.06.2020	Ismael Cuadrado-Cordero	Draft	Overall review
23.06.2020	Soonho Lee	Draft	Section 3.1.3.5
24.06.2020	Vlado Stankovski	Draft	Section 9

26.06.2020	Christophe Munilla	Draft	Improvements section 2
03.07.2020	Ismael Cuadrado-Cordero	Draft	Executive Summary, Sections 1 and 10
15.07.2020	Roberto Doriguzzi-Corin, Silvio Cretti, Domenico Siracusa	Draft	Internal review
04.08.2020	Ismael Cuadrado-Cordero	Final	Final version
04.12.2020	Roi Sucasas Font Alejandro García Bedoya	Final	Reviewers comments addressed

Notice that other documents may supersede this document. A list of latest *public* DECENTER deliverables can be found at the DECENTER Web page at https://www.decenter-project.eu/.

Copyright

Acknowledgements

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 815141 (DECENTER: Decentralised technologies for orchestrated Cloud-to-Edge intelligence)

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT/IITP) (No. 1711075689, Decentralised cloud technologies for edge/IoT integration in support of AI applications).

The partners in the project are FONDAZIONE BRUNO KESSLER (FBK), ATOS (ATOS), COMMISSARIAT À L'ENERGIE ATOMIQUE ET AUX ENERGIES ALTERNATIVES (CEA), COMUNE DI TRENTO (TN), ROBOTNIK (ROB), UNIVERZA V LJUBLJANI (UL), KOREA ELECTRONICS TECHNOLOGY INSTITUTE (KETI), GLUESYS (GLSYS), DALIWORKS (DW), LG U+ (LGUP), SEOUL NATIONAL UNIVERSITY (SNU).

The content of this document is the result of extensive discussions within the DECENTER © Consortium as a whole.

More information

Public DECENTER reports and other information pertaining to the project are available through DECENTER public Web site under http://www.decenter-project.eu.

Contents

List of figures

List of tables

Executive Summary

This deliverable focuses on the implementation activities of the computing platform during the second year of the DECENTER project, carried out in tasks T3.1 (Design and implementation of the fog computing platform), T3.2 (Resource exchange broker and blockchains to enable cross-border edge federation), T3.3 (QoS-aware resource orchestration) and T3.4 (Infrastructure security and robustness). These activities have led to the release of the updated version of the DECENTER PaaS, and its implementation from the two different aspects (IaaS and tools for AI).

These activities have focused on different aspects: (i) improving the support for federation of edge/Cloud resources across multiple administrative regions; (ii) improving the resource advertising and leasing through the network; (iii) deploying a monitoring and SLA ensuring system to increase trust in the system; (iv) provide an AI-focused platform for microservice applications; and (v) ensure the security and trust on the system. All these steps have been taken based on existing, state-of-the-art, open-source technologies, and in this deliverable, we provide different levels of detail on these advances, always focusing on the readability of the document. With this goal in mind, we provide a summary of the main requirements gathered from WP2 and a generic scenario, which the reader can use throughout the document to understand the advances on the system.

Building on top of D3.1, in this document we continue the work on SLA and monitoring (left as future work on D3.1), improve the brokering platform shown in the previous year and ensure the support for AI applications from the design. Additionally, we also show our investigations on robustness and security for the fog, which we plan to finish by year 3. Finally, we draw on an initial evaluation of the work done up until year 2, with reproducible implementations for the user to corroborate our findings. Overall, this deliverable shows the advances on year 2 and sets the ground for the last contributions to the platform, to be presented on the final release of the architecture (D3.4, M30), on which the last contributions will be presented.

1 Introduction

The DECENTER project built a Fog computing platform that allows the deployment of AI applications in the Edge. The DECENTER platform is designed for service providers that own their infrastructure, but at the same time need to outsource resources. As required by the Use Cases defined in the deliverables D2.1 and D2.2, this platform dynamically federates the client's infrastructural resources to Cloud and/or Edge ones, in a vertical (edge-to-Cloud) to a horizontal (Cloud-to-Edge) federation, automatically discovering and leasing resources, while ensuring SLAs between the resource providers. This federation occurs across multiple administrative domains, while ensuring a trustful environment and adapting this approach to modern blockchain technologies.

As shown in D3.1, this platform is a technological breakthrough in the existing literature in the Cloud, because it is designed to host AI applications in the Edge. With this goal in mind, the architecture presented in the deliverable D2.2 includes a set of libraries and tools to support these deployments. This deliverable reports on the implementation activities carried out in the Fog PaaS of DECENTER during the second year of the project. In this deliverable we aim to describe the main advances with respect to D3.1, giving a high-level view of the platform without going into deep, code-centred, descriptions. For the reader avid for more detailed, we provide 3 annexes to this deliverable with code examples and more details on the specifics of different components.

The remainder of the document is as follows: Section 2 summarizes the main requirements presented in D2.1 and D2.2 and shows a generalist scenario to ease the reader through the innovations of the project. Section 3 focus on the advances in the implementation of the Fog computing and brokerage platforms. Specifically Section 3.1 deepens on the updates which have been done to the platform in the application composing and container orchestration, while Section 3.2 in the advances to specifically support the AI applications, and the tools and libraries that have been deployed/implemented to support those during this year. Then, Sections 3.3 & 3.4 drives on the work done for ensuring of SLAs, both from the SLA management and monitoring perspectives, and how these have been implemented/evolved during the second year of the project, and its integration with the different parts of the architecture, including the blockchain. Section 3.5 shows the advances in security and robustness of the platform, which tasks started by year 2; and Section 3.6 is dedicated to the discovery and leasing of resources, which is done through a blockchain brokerage platform. Finally, Section 4 shows an initial evaluation of two scenarios in DECENTER and Section 5 wraps up all the information of this deliverable in the form of conclusions and main lessons learnt. The remainder of Sections are dedicated to extra information, including annexes on the details of main architectural components.

2 Platform Requirements & Scenario

This Section recalls on the identified requirements and a generic scenario for the usage of DECENTER, to draw the picture upon which the innovations of WP3 are based. As it was done in the previous version of this deliverable, this section reports the result of this work to enlighten the role of the components described later in the document, and to help at understanding how these requirements led architectural decisions.

2.1 Identified Requirements

The requirements reported in Table 1 have been identified starting from the investigation on the use cases, refined and updated in the context of the WP2. Particularly, the table provides a summary of the common UCs requirements by architectural components and maps them to the platform-related requirements (as described in D2.1 [10]), as reported in the previous version of this deliverable.

Table 1: Common Use Cases Requirements

Architectural component	Category	Platform-related requirement	Common UCs requirement
Fog Platform / Brokerage Platform	Interoperability	FR-01, FR-02, FR-03, FR-04, FR-05	Ensure the correct relation between providers using smart contracts FR_UC2_006, NFR_UC3_001, NFR_UC3_002, NFR_UC3_004
Fog Platform	Reliability and Autonomy	FR-06, FR-07, FR-08, FR-09	Ability to respond even if specific nodes are not operating and recover from failure NFR_UC2_004, NFR_UC3_004
	IoT function Support	FR-38, FR-39	Collect / store data from heterogeneous sources and trigger alerts FR_UC1_007, FR_UC1_009-FR_UC1_012, FR_UC1_014, FR_UC3_003, FR_UC3_007, FR_UC4_006 - FR_UC4_008
Fog Platform / Management	Hierarchy and Scalability	FR-10, FR-11, FR-12	Addition or reduction of computing resources SR_UC1_001,SR_UC2_003,NFR_UC3_001
	Orchestration (Inter-fog and Infra-fog)	FR-25, FR-26, FR-27	Selection of appropriate AI models and response in a reasonable time NFR_UC1_006, NFR_UC1_009, NFR_UC3_003, NFR_UC4_004, NFR_UC4_005
	Virtualization and Containerization Support	FR-17, FR-18	The processing unit should have virtualization capabilities and be optimized for container virtualization : SR_UC1_005, SR_UC2_006
	AI function Support	FR-36, FR-37	Use, customize and distribute pre-trained AI models NFR_UC1_009, FR_UC3_001, FR_UC3_002, NFR_UC3_003, FR_UC3_004, FR_UC4_001, FR_UC4_003 - FR_UC4_005, NFR_UC4_005
	System Management	FR-19, FR-20, FR-21, FR-22, FR-23, FR-24	Ability to deploy applications on the fog and stable internet connection NFR_UC2_002, NFR_UC2_003, NFR_UC4_005, NFR_UC4_006,

	(Provisioning, Deployment, Alerting)		SR_UC1_002, SR_UC2_004, SR_UC3_007, SR_UC4_004
Management	Application and Service Management (Application Catalogue, Repository, Service)	FR-28, FR-29, FR-30	Repository of available application services / applications (e.g., Digital Twin, AI models) with easy installation and use FR_UC3_001, FR_UC3_002, SR_UC4_001, SR_UC4_002
	Resource Management (resource allocation, orchestration, sharing, REB)	FR-31, FR-32, FR-33, FR-34	Allocate and share resources SR_UC1_001, NFR_UC2_002, NFR_UC2_003, SR_UC2_003, NFR_UC3_001, SR_UC3_001, NFR_UC4_006
	User Management	FR-35	Provide authorized users with access to the management system NFR_UC1_005, NFR_UC1_011, FR_UC2_002, FR_UC2_003
Management /Authentication	Security	FR-13, FR-14, FR-15, FR-16	Safe data processing and storage, confidentiality, authentication, and access control NFR_UC1_010, NFR_UC2_005, NFR_UC1_007, NFR_UC2_001, NFR_UC3_002

2.2 Scenario

A generic workflow of the utilization of DECENTER is shown in Figure 1, similar to the one presented in D3.1 [15]. This sample workflow[1] allows us to add information about components added to the workflow in the second year of the project, such as the monitoring system and the Service-Level Agreement (SLA) manager. As a reminder, this workflow involves the modules and interfaces here described and tackles two closely related scenarios:

- The basic one (in line with the initial deployments of fog computing infrastructures and control planes), which we call here “single-fog” (SF) for simplicity, where resources are owned by a single Infrastructure Provider and microservice-oriented AI applications are deployed over a cloud-to-edge infrastructure already set up;
- The advanced one (more “futuristic”), which we call here “multi-fog” (MF), in which resource sharing across two Infrastructure Providers is needed to deploy a microservice-oriented AI application, because e.g. a provider does not have enough fog computing resources in a specific location. It contains all the steps from resource advertisement on the Resource Exchange Broker to the application deployment.

In practice, the single-fog (SF) scenario is a strict subset of the multi-fog (MF) one. Therefore, the two scenarios are described hereunder in just one flow, highlighting each step with (MF) or (SF, MF) depending on the scenario they belong to. In the remainder of the deliverable, we will not use this distinction, since it is mostly presented to show that different flows are possible depending on the needs of the use cases supported by DECENTER.

[1] Due to the complexity of the involved processes and technologies, the workflow mentions only the main components and provides a simplified representation of their operations and interactions.

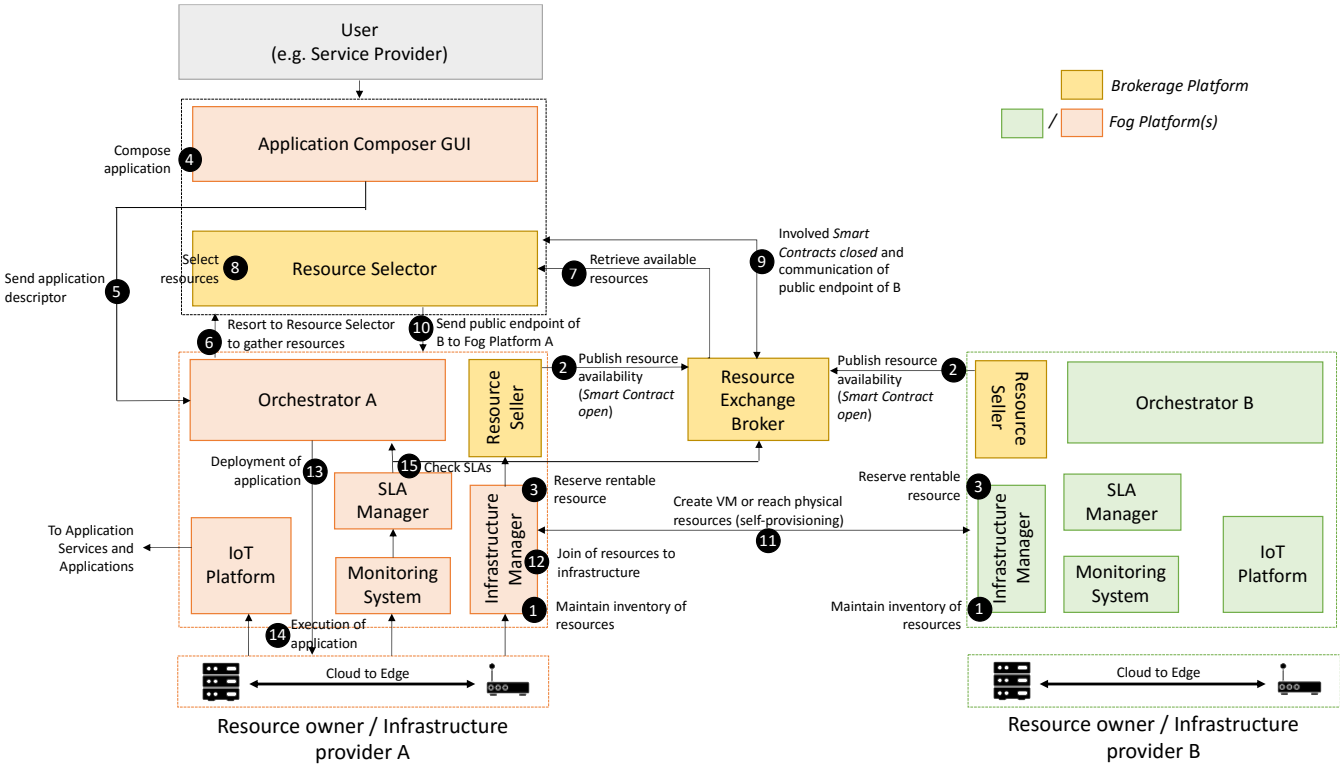

Figure 1: Sample workflow scenario

Our initial assumptions for the multi-fog scenario are identical to those specified in D3.1:

- There are only two Infrastructure Providers sharing their resources: Infrastructure Provider A and Infrastructure Provider B;
- The Service Provider needs resources from both Infrastructure Provider A and Infrastructure Provider B;
- The Service Provider chooses an Infrastructure Provider (e.g. Infrastructure Provider A) as the *main provider*. The GUI and Resource Selector used are those from the main provider and the Service Provider prefers using the resources from the main provider;
- The Infrastructure Providers adopt an IaaS-based approach for sharing resources.

In this context, the workflow is the following:

1. (SF, MF) *Maintain inventory of resources*: the Infrastructure Manager of the DECENTER platform continuously monitor the existing resources that can be used to deploy cloud-native applications or shared with third parties through the Brokerage platform.

2. *(MF) Publish resource availability*: both infrastructures use the Resource Exchange Broker APIs to specify their resources availability. The exchanged data structure contains the type and amount of resources to be shared, as well as guaranteed Service Level Objectives (SLOs). Once a resource is declared to be available to the Resource Exchange Broker, the latter opens a Smart Contract with all the information.

3. *(MF) Reserve rentable resources:* the Infrastructure Providers are responsible for ensuring that committed resources are available to rent. Therefore, upon publication in the Resource Exchange Broker, the Infrastructure Provider will revoke these resources' availability.

4. *(SF, MF) Compose application*: the Service Provider designs their application and specifies its requirements by using the Application Composer GUI (front end of the Fog Platform). During the last year, the GUI has been improved in combination with the Orchestrator to offer efficient application composition and deployment, in a distributed microservice-oriented AI application context.

5. *(SF, MF) Send application descriptor and check available resources*: the Application Composer GUI generates a description file, which contains all the relevant information of the application. If the resources requested by the application are already available (case SF), the workflow continues from step 13.

6. (MF) *Resort to Resource Selector to gather resources*: if the resources requested in the application descriptor are not already present in the infrastructure (case MF), the Resource Selector is involved to match user's resources needs to third party providers' offer.

7. *(MF) Retrieve available resources*: upon request, the Resource Exchange Broker (REB) forwards to the Resource Selector the description of the available resources that have an open Smart Contract.

8. *(MF) Select resources:* the Resource Selector selects the resource offer that better suits the needs of the application.

9. *(MF) Involved Smart Contracts closed and communication of public endpoint of B*: once the resources are chosen, the Resource Selector calls the Resource Exchange Broker APIs to create a Smart Contract, which is recorded to the REB's blockchain together with a corresponding set of SLOs. Resource Exchange Broker exchanges the *public endpoint* used by Infrastructure Provider A to attach the selected resources from Infrastructure Provider B.

10. *(MF) Send public endpoint of B to Fog Platform A*: The Resource Selector from Infrastructure Provider A communicates the public endpoint for rented resources and their amount to the Fog Platform A APIs. Then, the Infrastructure Provider A starts the procedure to federate the resources shared by Infrastructure Provider B in an IaaS model.

11. *(MF) Create VM or reach physical resources (self-provisioning)*: The Infrastructure Provider A uses the Provisioner component for self-provisioning of resources over the Infrastructure Provider B.

12. *(MF) Join resources to the infrastructure:* The Provisioner A module performs a *join* of virtual (or physical) resources from Infrastructure Provider B into the pool of resources (i.e., Kubernetes cluster) of Infrastructure Provider A. This way, the infrastructure of Infrastructure Provider A is *augmented* with the resources shared by Infrastructure

Provider B and federation of resources is performed. At this point, repeat step 5b, that is, the application descriptor is sent to the orchestrator, and go to step 12.

13. *(SF, MF) Deployment of application:* Infrastructure Provider A uses the Orchestrator to apply the (Re)deployment Algorithms, which, in turn, leverage the Monitoring data, in order to find the best real-time placement for the different microservices of the AI application and to deploy them over the infrastructure spanning from the cloud to the edge, while meeting the applications requirements. During year 2 the entire deployment process (and the eventual re-deployment, in the case in which requirements for an already deployed application are not met anymore) has been enhanced and made smarter thanks to the usage of metrics collected from the infrastructure.

14. *(SF, MF) Execution of the application*: while running, the application can take advantage of the services offered by the DECENTER Platform and Services. Thanks to the IoT Platform, the application can gather and effectively process at the edge data produced from an heterogeneous set of sensors and devices. Furthermore, the Application Services offer libraries and tools for ease the creation and composition of AI components and models. Finally, the infrastructure and the applications running on the fog platform can rely upon the security services offered by DECENTER to protect against attacks. The integration of the platform and the services is provided in year 2 of the project.

15. *(SF, MF) Check Service Level Agreements*: During the lifetime of the application, the SLA Manager, which is the component in charge of SLA monitoring, will use Monitoring System to detect possible violation of SLOs and will write such violations to the blockchain using the REB API. Violations of SLA related to the service may also trigger service re-deployments. This is a new deployment provided in the second year of the project.

3 Updates on the fog and brokerage platforms

This section provides an overview of the improvements made to DECENTER at an implementation level, in relation to Y1.

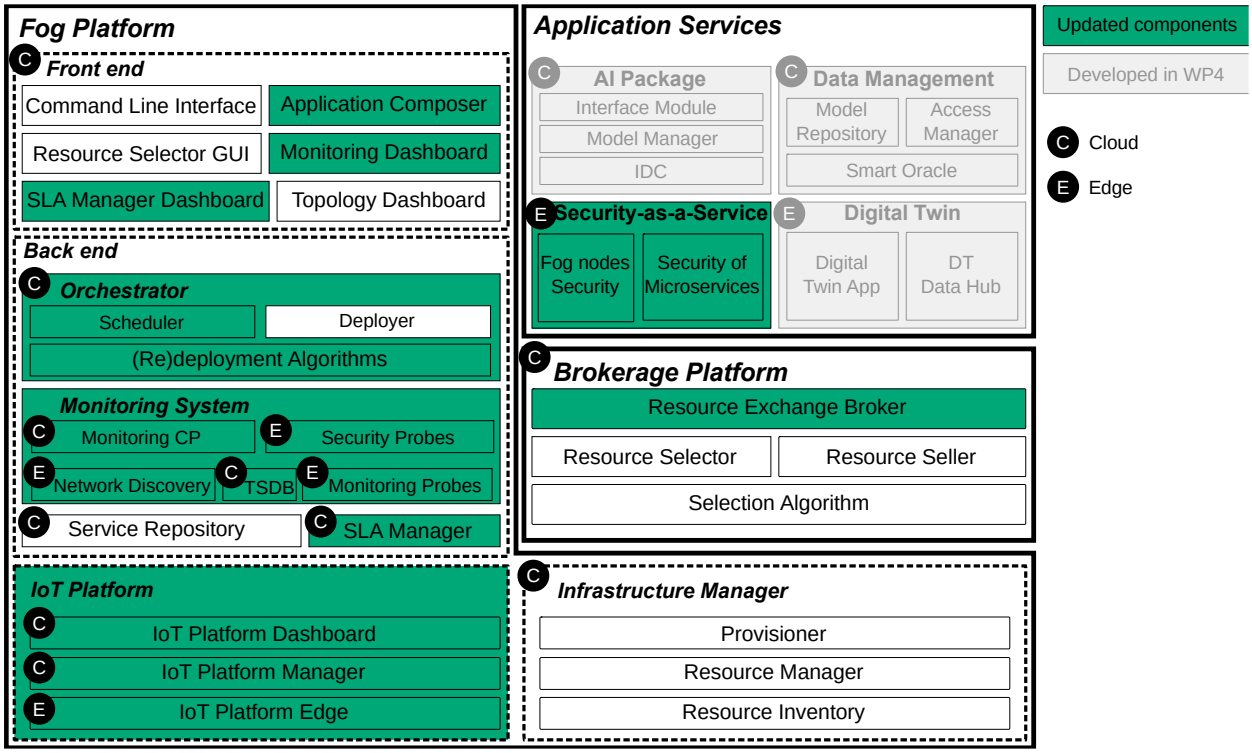

Figure 2: DECENTER Platform Layer architecture with high-level updates

While providing an updated version of the architecture of the DECENTER platform is out the scope of this document (see D2.2 [16] for a discussion of the whole DECENTER architecture), in this section we report the architectural changes that affect WP3 in order to support the WP3 implementation work as envisioned for Year 2 of the project.

Figure 2 shows an updated high-level view of the DECENTER computing platform architecture. The figure is based on the final architecture of the DECENTER platform documented in Deliverable D2.2 “Final release of the DECENTER architecture specification and use cases”. WP3 components that have been added or updated in Year 2 are depicted in green. Opaque boxes are instead related to WP4 activities and their description is deferred to WP4 deliverables.

The updated figure includes the following components/functionalities:

- **Application Composer GUI and Orchestrator:** One of the activities that has been carried out is better supporting, within WP3, the composition and deployment on the Fog Computing Platform of *AI applications* as developed in WP4. To this aim, the Application Composer GUI and the Orchestrator components have been extended so that containerized services and applications can be properly composed and more effectively deployed on a distributed infrastructure also considering AI related resources (i.e. Graphical Processing Units - GPUs).
- **IoT Platforms:** What was missed in the overall D3.1 architectural picture is the role of the IoT platforms that are adopted in DECENTER, i.e., ThingPlus and SensiNact.

DECENTER is working on the containerization and decentralization of the ThingPlus platforms, which is currently cloud-based. The idea is to move part of the platform functionalities at the edge, so that some processing on collected data can be performed locally. SensiNact is instead being extended to make it decentralized at the edge, by enabling the possibility to make different SensiNact instances, deployed at different edges, to communicate.

- **SLA Manager**: the SLA Manager is also a main innovation in year 2, deploying a component that allows to verify whether an SLA stipulated between two Infrastructure Owners (when they share resources through the Resource Exchange Broker) is fulfilled or not. The SLA Manager needs to interface to the Monitoring System to collect monitoring data from involved infrastructure and needs to use the REB APIs to gather SLAs that need to be verified or to write the result of a SLA after the completion of the application lifetime.
- **Monitoring System:** the design and implementation of a Prometheus-based Monitoring System is one of the main activities in the DECENTER platform for the second year of the project. Such a Monitoring System provides information related to the infrastructure status (computational resources and network resources) together with the collection of application related metrics. Those metrics are mainly used to serve as input data for:
    1. the algorithms for *application deployment* and *QoS-aware re-deployment*, which run on the Orchestrator, on the Resource Selector, or both and need monitoring data as input for deployment/re-deployment decisions;
    2. the *SLA Manager* component to evaluate whether SLAs stipulated between federated Infrastructure Owners and stored in the Resource Exchange Broker are being fulfilled or not.
- **Robustness and Security:** T3.4 (Infrastructure security and robustness) activities started in M13. Concerning *robustness* aspects, the task is working to make the Fog Computing Platform more resilient exploiting the Kubernetes Cluster Federation. Concerning *security* aspects, the task is working on securing the fog nodes by developing a lightweight fog nodes intrusion detection and mitigation system.
- **Brokerage Platform:** a subsection of this deliverable is dedicated to showing the advances on the development of a brokerage platform to allocate resources in DECENTER. During year 2, we have investigated on the challenges between using a public blockchain infrastructure or a private one to address the requirements associated with the exchange of resources between administrative domains.

In the following sections, we present the updates to the above components/functionalities in greater detail.

3.1 Application Composer and Container Orchestration

The Fog Computing platform, and the Orchestrator and Application Composer components, has been evolved during the second year of the project in order to be better integrated with the chosen container orchestration platform (Kubernetes). While the primary solution was conceived as a new layer implemented on top of Kubernetes, consequently offering its own API together with the Kubernetes ones, the new approach stems from the Kubernetes' Customer Resource Definition. New types (i.e. resources) have been defined to extend Kubernetes towards a platform able to manage distributed/fog infrastructures.

Therefore, on the one hand the new DECENTER Fog Platform is an evolution of Kubernetes, but on the other hand, it is significantly simplified with respect to the one implemented in year 1, better exploiting the components offered off-the-shelf by Kubernetes (e.g. the API server). Essentially, this new solution consists of:

- Kubernetes vanilla;
- A set of CRDs (Custom Resource Definition) that extend the resources handled and the API offered by Kubernetes, in particular:
    - *Region*, which models a geographical location where some computational capacity, together with some IoT devices, are deployed. Kubernetes worker nodes belong to a Region;
    - *Link* between two Regions in terms of its endpoints and properties (bandwidth and latency);
    - *ExternalEndPoint*, either a device (i.e. sensor) not directly managed by the platform or an external application service, and is used by the applications deployed by the Fog Platform;
    - and *FADeply (FogAtlas Deployment)*, which extend the Kubernetes Deployment resource to describe a graph of microservices interconnected with data flows. These microservices and data flows can have requirements and constraints that the Fog Platform orchestrator must satisfy in order to carry out the placement task.
- A custom Kubernetes controller able to handle those new CRDs;
- An adaptation of the Graphical User Interfaces (mainly the Application Composer) to the new CRD feature.

Implementation of CRDs

Using these CRDs, together with the Kubernetes tools (e.g. client-go libraries, code generators), a custom Kubernetes controller has been developed in order to handle the FADepl resource. Moreover, the Kubernetes' set of APIs has been extended in order to comprise the new API for executing CRUD operations on the CRDs. Such APIs are then used by the Application Composer GUI to interact with the Kubernetes backend.

More Detail on the implementation of CRDs, together with relevant examples, is shown in Annex A.

Updates of the Platform for the AI components

In order to support complex computation to bring AI services to the cloud and the edge, the management of the accelerator resources from the Fog Platform is essential.

The most popular accelerator for AI is GPU, and it is a common practice using GPU for acceleration of deep learning methods, however, since GPU is a specific device which is not orchestrated as a generic resource such as CPU, memory or storage, it needs some extra implementation to support those resources.

During the second year, we have investigated on the accelerator resources, to implement their orchestration within the DECENTER Platform. Among various hardware accelerators, GPU devices from Nvidia have been chosen for the resource orchestration due to their popularity, however these resource descriptions are designed to support different kinds of hardware acceleration devices in the future.

Nvidia manufactures different acceleration devices, which can be categorized in two groups: graphic cards for PC and ARM-integrated SoC devices. Nvidia has a set of GPU families with various kinds of specifications. Among those specifications, two of them are important from

the viewpoint of deep learning - number of cores (CUDA core or Tensor core), and the size of memory associated with the GPU. For Nvidia GPU chips, those two specs are subject to the type of chips, which is identified by the model name.

Nvidia has a SoC family named Tegra. It integrates ARM architecture CPU with GPU and can be used for acceleration for deep learning just like normal GPU. However, since it is an SoC, the memory is shared between GPU and CPU. There are few types of Tegra SoC, and each one is identified by the model name, as shown in Table 2.

Table 2: Comparison of Tegra SoCs

	GTX 1080	GTX 1080ti	GTX 1070	GTX 1070ti
Number of Cores	2560	3584	1920	2432
Size of Memory	8GB	11GB	8GB	8GB

GPU is a special type of device, and it needs some support software to be installed together with the common deep learning platform (i.e. Tensorflow, Keras or PyThorch). This software is composed of drivers and acceleration libraries. On the one hand, the device drivers handle the communication between the device and the kernel; on the other hand, the acceleration library communicates the device driver with the deep learning platform.

In the case of Nvidia, the bundle of device drivers and acceleration libraries is commercialized under the name “CUDA toolkit”. The CUDA toolkit consists of various libraries that help complex computation, as shown in Figure 3.

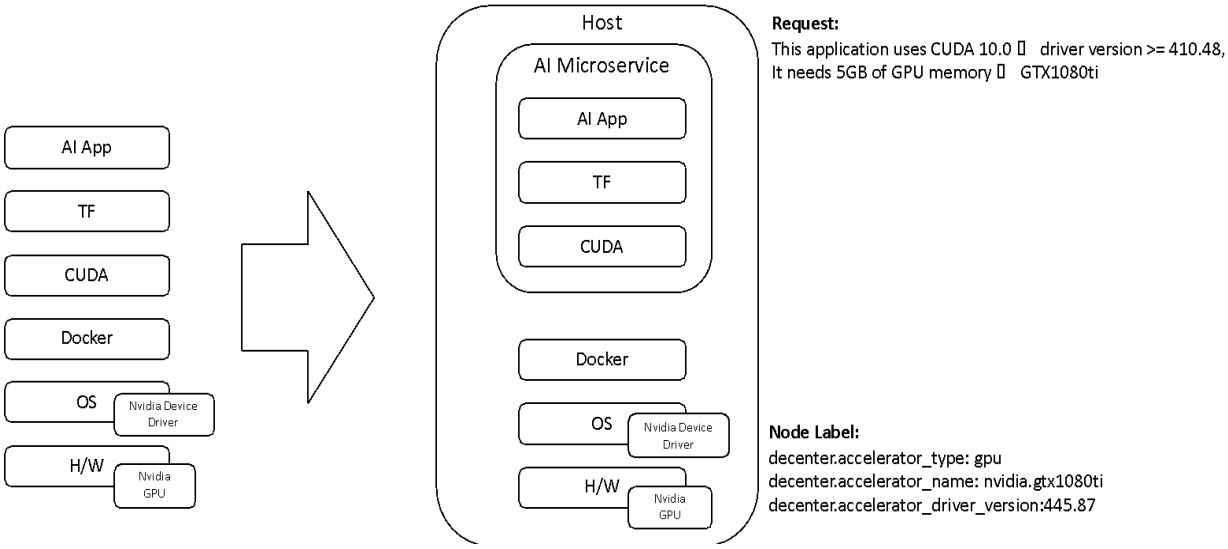

Figure 3: Software stack of a GPU-accelerated application

Also, the acceleration software for GPUs has dependencies with the AI libraries. For example, to run a specific version of Tensorflow it needs a combination of compatible version of device driver and CUDA toolkit. If the versions do not match, the application will fail to run properly.

From the viewpoint of resource orchestration and containerization it needs to consider which part of the software stack to put into a container. Since device drivers are software on a host and cannot be containerized, it needs to be put on the characteristics of hardware.

GPU Support in Kubernetes

To better adapt the Fog Platform to the needs of the AI applications, we added support to the scheduling of workloads on GPUs to both the Orchestrator and the Application Composer components. While Kubernetes already offers support to GPU, this is still in an experimental phase and strongly depends on the availability of the “device plugin” feature that must be developed by the different hardware vendors. For this reason, we implemented a simple yet flexible solution, based on node labelling, in order to map the allocation of GPUs on the distributed infrastructure and to properly schedule the workload requesting the usage of GPUs.

Currently a simple set of labels that specify the presence of a given type of GPU on the nodes of the infrastructure are matched with corresponding node selector labels specified in the application deployment (FADepl) allowing to place an AI component on the node hosting the requested GPU.

In the future, we can extend such label set adding more GPU characteristics as GPU type, name, version and memory available. See the following table as an example.

Table 3: Labels for identification of accelerators

label	values	Description
decenter.accelerator_type	gpu soc	Type of GPU unit
decenter.accelerator_name	nvidia.gtx1080ti nvidia.rxt2080ti nvidia.gtx1660ti nvidia.tegra.x2	Name of GPU device
decenter.accelerator_driver_version	int.int (major.minor)	Version of Nvidia driver installed on the node
decenter.accelerator_memory	2gb, 6gb, 10gb...	Byte of memory of GPU (SoC only)

3.2 IoT Platforms

In this section, we focus of the integration of the IoT platforms that were chosen for the project into the DECENTER platform (i.e. sensiNact & Things+), as those are an integral part of the support of the platform for AI. Both platforms have been evolved within the project to go beyond the cloud-based approach followed in current commercial deployments, by including components responsible for collecting data from IoT devices and make it available at the edge of the infrastructure to the microservices of the AI applications.

3.2.1 SensiNact

The middleware of sensiNact has a dual role in DECENTER, acting as an IoT provider and as a Digital Twin provider. In this subsection, we analyse both characteristics, highlighting the flexibility of sensiNact in providing different roles and elaborating on the following aspects of sensiNact:

- provide the list of necessary modules for deploying sensiNact,

- describe the available interfaces for interacting with sensiNact and
- explain the integration plan with Fog Computing Platform of DECENTER.

Finally, we dedicate a subsection to highlighting why those two roles should interact with each other and we describe two mechanisms for achieving this interaction.

sensiNact as IoT provider

Figure 4 illustrates the modular architecture of sensiNact, as already discussed in D3.1 [15]. Each one of these modules comes in the form of a bundle, which can dynamically and remotely be installed in the OSGi environment without requiring a reboot of the entire middleware.

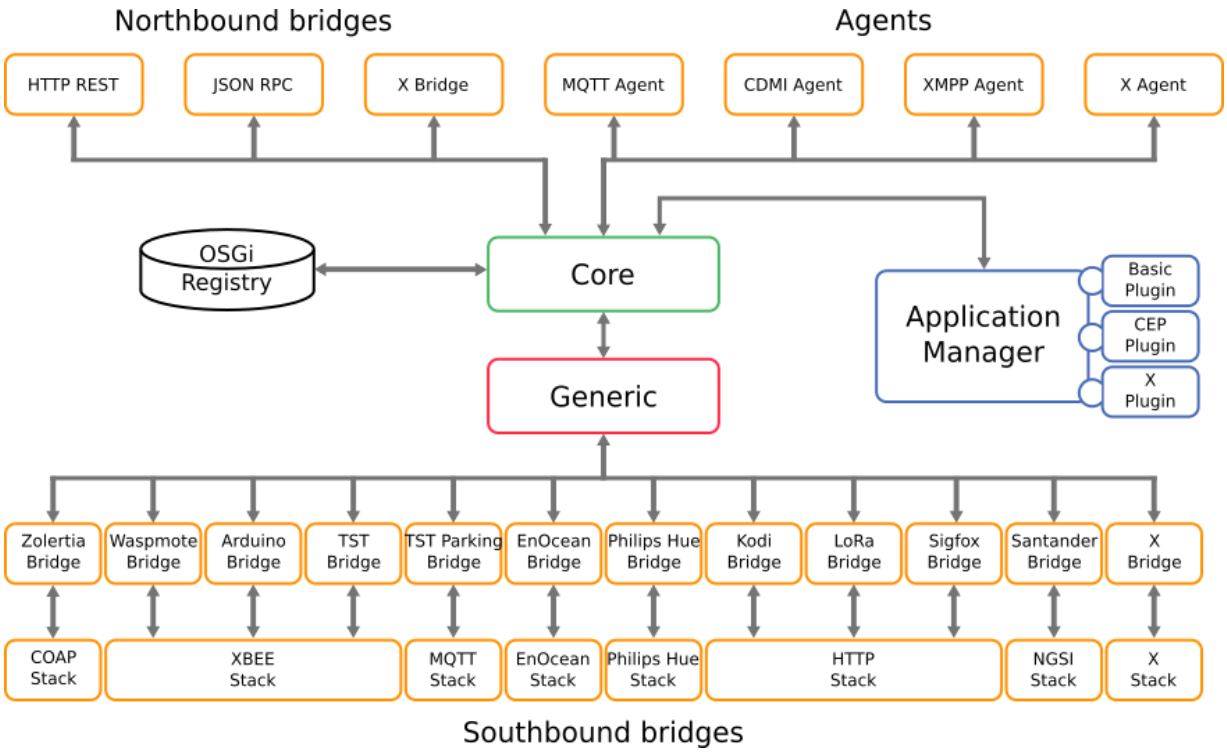

Figure 4: Architecture of sensiNact

Table 4 summarizes some of these necessary bundles that allow sensiNact to provide data deriving from IoT devices. These bundles can be found in the eclipse, open-source repository. This table does not provide the complete list of bundles and this list can be extended to cover further communication protocols of IoT devices. However, HTTP, MQTT and Openhab protocols are sufficient for communicating with any device needed by the Use Cases of DECENTER.

Table 4: Bundles of sensiNact as IoT provider

Northbound	**Southbound**	**Other**
rest-access *storage-agent* *sensinact-access*	*http* *mqtt-device* *openhab*	*sensinact-core* *sensinact-common* *sensinact-generic* *sensinact-utils* swagger

To facilitate the interaction of any third-party application with sensiNact IoT provider, an interface using the swagger tool has been built, giving access to a REST API for easy retrieval and actuation on the IoT devices. Also, the northbound bundle of storage-agent is available for storing all incoming data to a Cassandra database.

In order to easily integrate sensiNact in the Fog computing platform of DECENTER, during year 2 we provided a containerization tool within the eclipse source repository allowing to build a dedicated Docker container image including the selected sensiNact's modules; This makes sensiNact available as an application service, accessible via the Application layer of DECENTER. A light configuration of the resulting container might be necessary before deployment to define swagger and Cassandra ports.

sensiNact as Digital Twin provider

Table 5 summarizes the minimum necessary bundles for sensiNact to provide AI data. Two new bundles were created for this purpose; *digital-twin* and *mqtt-decenter*. The former takes as input a data model, which describes the AI entities to be represented, and adapts the incoming data to that model. The latter implements a publish/subscribe pattern to communicate with AI models and retrieve updates on their virtual detected entities.

Table 5: Modules of sensiNact as Digital Twin provider

Northbound	**Southbound**	**Other**
rest-access *sensinact-access*	*http* *digital-twin* *mqtt-decenter*	*sensinact-core* *sensinact-common* *sensinact-generic* *sensinact-utils*

Figure 5 illustrates the interactions of the sensiNact Digital Twin with the third-party user applications, the AI models, and the core of sensiNact middleware. While the Digital Twin is part of the sensiNact middleware, for demonstration purposes they are illustrated separately. This figure shows the interfaces, which have been implemented to realize all necessary interactions. Specifically, a swagger tool over a REST API is available for the user applications and an MQTT client is provided to connect with AI applications.

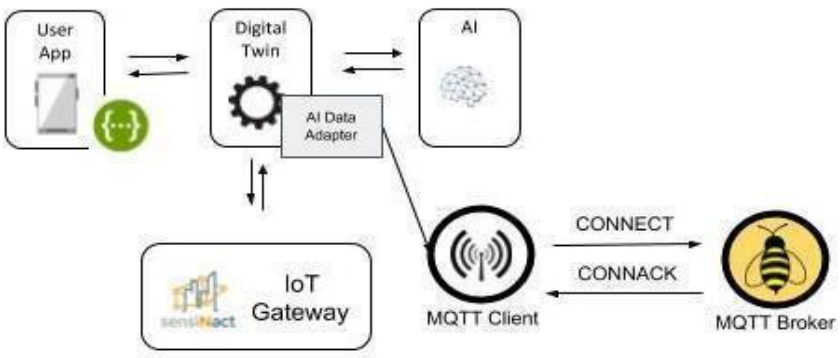

Figure 5: Interactions with Digital Twin [1]

To easily integrate sensiNact Digital Twin provider in the Fog computing platform of DECENTER, during year to we built a container image with Docker. This image is also available via the Application layer. Similarly, some configuration of the container is required (i.e., define ports of swagger tool, MQTT).

Interaction between sensiNact instances

The two roles of sensiNact are complementary and it is useful to the system to interact with each other. For instance, we can consider the scenario where sensiNact Digital Twin provider represents IoT devices from the sensiNact IoT provider, where the latter is deployed remotely, in a different edge than the Digital Twin. Thus, the communication between the two roles of sensiNact becomes imperative. SensiNact supports such interactions using the OSGi remote service model, as shown in Figure 6, allowing two distinct OSGi environments to communicate. This implies that for N sensiNact instances to be installed, N+1 containers must be deployed, each one embedding the necessary configuration tools and a potential broker for shared messages, respectively Apache Zookeeper and an embedded Moquette MQTT broker.

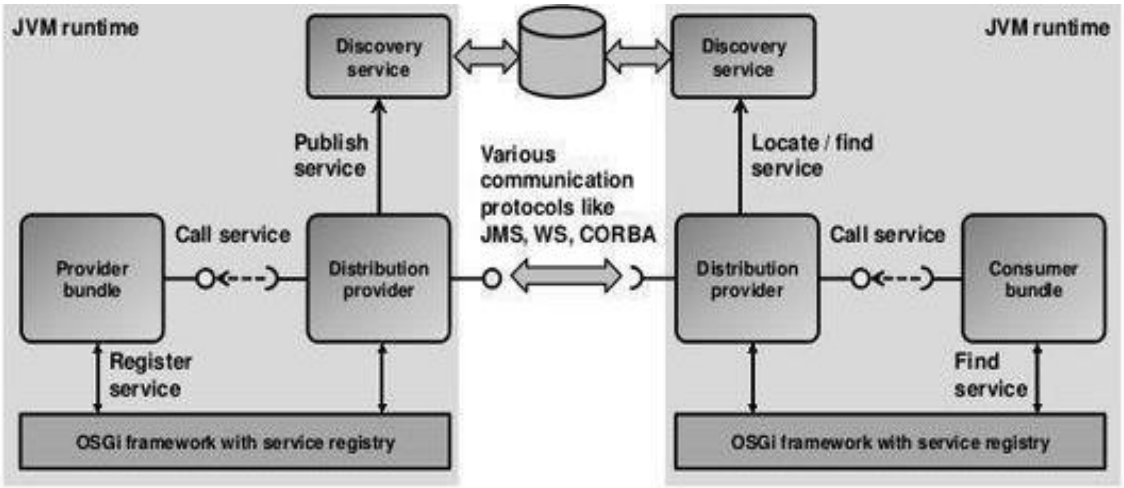

Figure 6: OSGi remove service model

3.2.2 Thingplus

Thingplus is an IoT cloud platform which enables rapid development, management, and scaling of IoT services. Thingplus enables IoT service providers to collect and visualize data from devices and assets, analyze incoming telemetry and trigger alarms with complex event processing, control devices using remote procedure calls, and design dynamic and responsive dashboards and present device and asset telemetry and insights to their customers. Thingplus, which is built with leading open-source technologies, is horizontally scalable and reliable platform without single-point-of-failure.

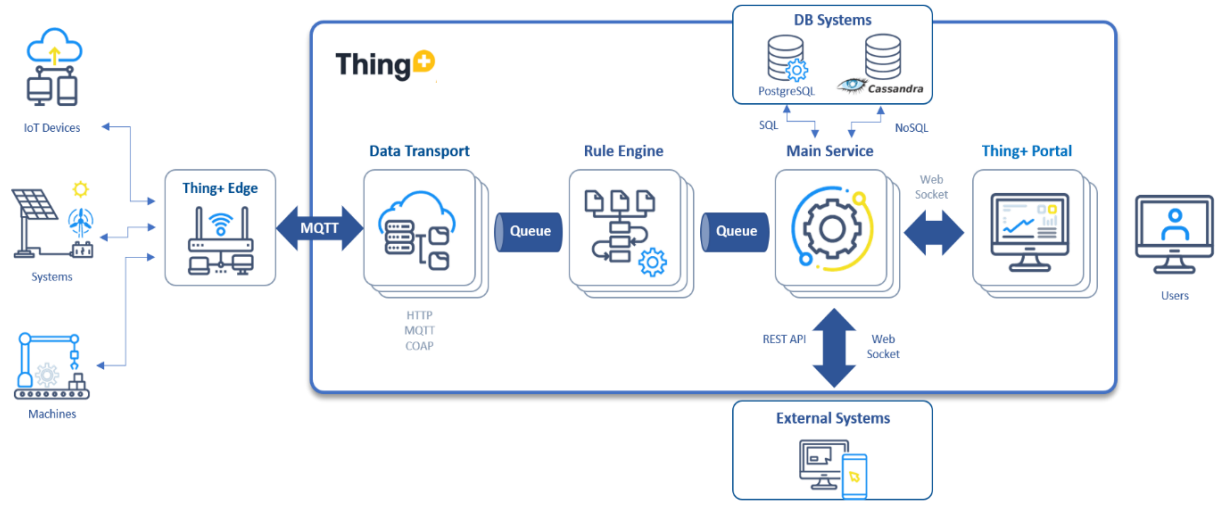

Figure 7: Thing+ Architecture

To ensure communication, Thingplus supports MQTT, HTTP and CoAP protocols and provides protocol APIs, each through a separate server component. Once the Data Transport receives the message from the device, it is parsed and pushed to the message queue. The Rule engine component, which is responsible for processing incoming messages, subscribes to incoming data feed from queues and acknowledge the message only once it is processed. The Main Service, which is responsible for handling REST API calls and WebSocket subscriptions, stores up to data information about active device sessions and monitors device connectivity state.

Thingplus is integrated into the DECENTER Fog Computing platform to provide IoT service features in each use cases in the DECENTER project. The Thingplus Edge, which is responsible for collecting data from IoT devices, is implemented as a microservice to provide data transmission between IoT platform and AI applications.

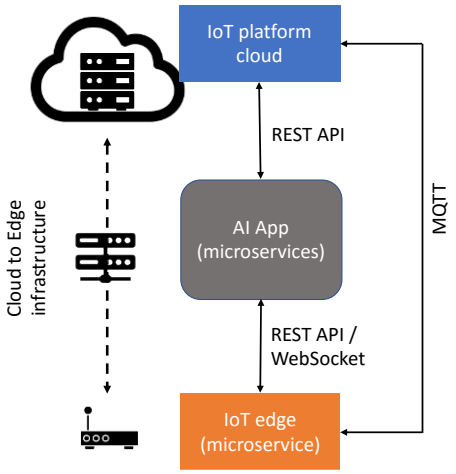

Figure 8: IoT Platform Integration in DECENTER Cloud to Edge computing

3.3 SLA Management

This section summarizes the advances on the Quality of Service assurance. That is, it shows how the SLA manager and monitoring tools have been designed to be integrated in the system, and the advances in the implementation of both.

3.3.1 Smart Contracts for SLA management in Edge-to-Cloud Environments

The management of Service-Level Agreements (SLAs) in Edge-to-Cloud computing is a complex task due to the great heterogeneity of computing infrastructures and networks and their varying runtime conditions, which influences the resulting Quality of Service (QoS). SLA-management should be supported by formal assurances, ranking and verification of various microservice deployment options, as shown in D2.2. Our work aims at introducing a novel Smart Contract (SC) based approach for SLA management that would support various entities and actors in a decentralised computing environment composed of multiple clusters. These mainly addresses the needs of Cloud service consumers and Cloud providers.

As described on D2.1 [10] and D3.1 [15], in the context of DECENTER Service Level Agreements (SLAs) have been defined as part of a smart contract, which is stored in a decentralized system, leveraging blockchain technologies. In this SLA, it is included information, among others, on the duration of the lease (life of the SLA), the user and provider of the service, and the Service Level Objectives (SLOs) which will be measured to ensure that the SLA is being respected. As described in these deliverables, the components needed for a correct integration of smart contracts in the system are:

- **Blockchain nodes**, which represent a set of nodes that store the blocks (SLA, transactions and SLA violations). The use of these nodes ensures that blocks are not tampered. Each block is signed using a cryptographic value, and all blocks are known by all nodes. Thus, if a node tampers a block it is automatically discovered by all the others.
- **Oracle**, which is a single point entry to the chain. This component centralizes the requests to the blockchain, thus if a write is made in the chain, it needs to be centralized in this.
- **External evaluation tools** that ensure the completion of the contract and alerts the oracle in the case where it has not been respected.

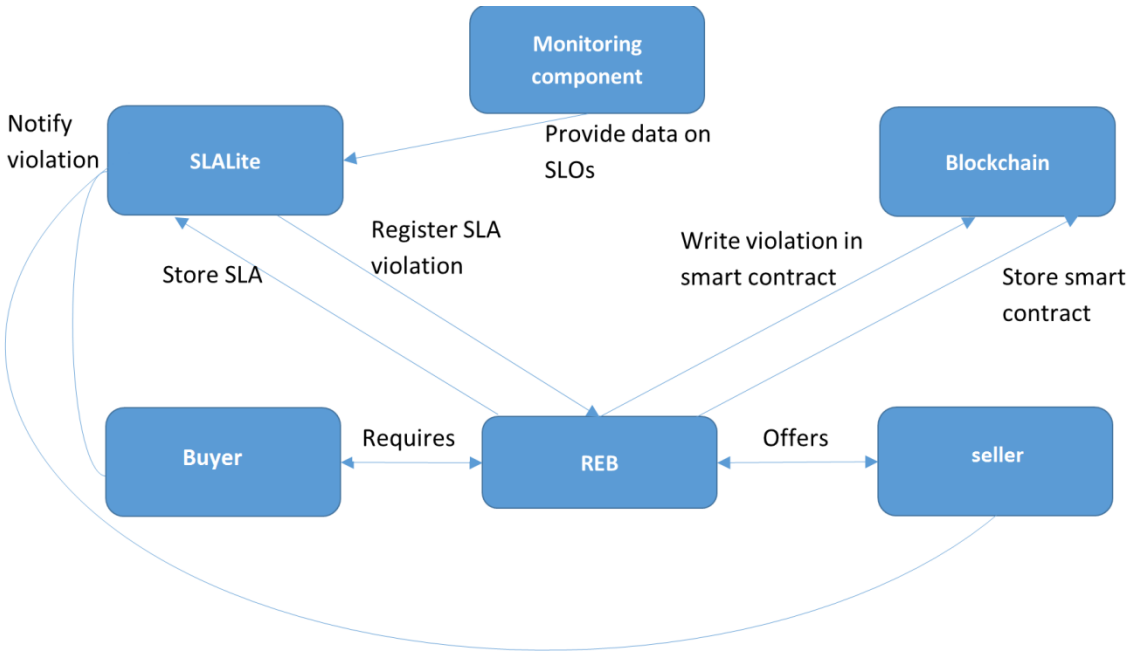

Figure 9: SLA Manager integration in DECENTER

As shown in Figure 9, the DECENTER platform already integrates an Oracle, which function is taken by the Resource Exchange Broker (REB), being the only component allowed to write in the blockchain. Regarding the SLA completion, the functionality of the REB is acting as an oracle to the system, writing in the blockchain the new smart-contract signatures and, if necessary, the violations made to those. The REB does not only write the smart contract in the blockchain, but also sends the main information, in a simpler format, inside the SLA Lite tool. Additionally, the resource selector will ensure the agreement between provider and user, by automatically matching each user to the provider that satisfies their needs. During the second year of the project, an API REST has been developed to satisfy this interaction from the side of the REB, allowing the SLA Manager component (SLA Lite) to communicate to the blockchain.

The second component to be shown in this figure is SLA Lite. This component (specifically developed by Atos to ensure the SLA management in Fog and Edge systems), ensures that the SLA is respected during its whole lifecycle. To do this, it evaluates frequently the data on the SLOs defined for each SLA and, if these SLOs have been broken, then notifies the REB to register the violation on the blockchain. It also notifies each involved stakeholder (namely the user and the provider) of this situation. This notification can be done through a push-based protocol (e-mail). During the second year of the project, this integration with the REB has been done.

Finally, the last independent component of this design is the monitoring system. This oversees retrieving the data related to the SLO, and sends these data to the SLA Lite, which later is evaluated following the information presented in the SLA. The monitoring system is discussed on Section 3.4 and has also been integrated with the SLA Lite during year 2.

3.3.2 SLA Manager

The integration of the SLA Manager in the system, as described above, includes three main steps.

1. **Integration with the monitoring component.** In order to receive the SLO data, the SLA manager needs to be able to establish a communication system with the monitoring component to receive the SLO data and to start/stop the measuring of these. This means that the monitoring system does not only need to be able to provide monitoring data, but also to ensure that it can start measurements “on- demand”.
2. **Integration with the REB.** Second, the SLA manager needs to be integrated with the REB. This integration will be used for two different actions, receiving the SLA signed between both actors (user and provider) and to communicate SLA violations which will be written.
3. **Integration with the user/provider.** This communication channel will be used to ensure that the actors involved with every SLA are notified whenever this is not being respected. This communication channel is one way, that is, the user/provider cannot send data to the SLA manager, and in the first iteration we consider using a direct communication method between the component and the actors. In future iterations we will consider if it can be centralized in the REB in the form of a queue.

All these communication channels can be opened in two different protocols, either synchronously or asynchronously. In the first case, the monitoring component’s API can be used to retrieve data, while for asynchronous communication it will be necessary to implement a pub/sub system. In an initial implementation, it will be investigated the use of synchronous communications, while in later iterations we will consider the use of pub/sub systems and their integration on the platform.

The language used in all these communication channels was JavaScript Object Notation (JSON), an open-standard format that is human-readable. It is used to transmit data objects, using value pairs and array types.

3.4 Monitoring system

3.4.1 Resource monitoring

The DECENTER Fog platform provides features such as resource availability, service deployment / QoS monitoring, SLA violations, usage accounting, and resource sharing for a variety of AI applications. These functions operate based on the collected resource information from the monitoring system (CPU, GPU, Memory, File System, Network, K8s Pod, etc.), and this resource information must be well defined in advance and reliable resources in order to expect the function to work properly. Therefore, the monitoring system is a central, main component to the DECENTER platform.

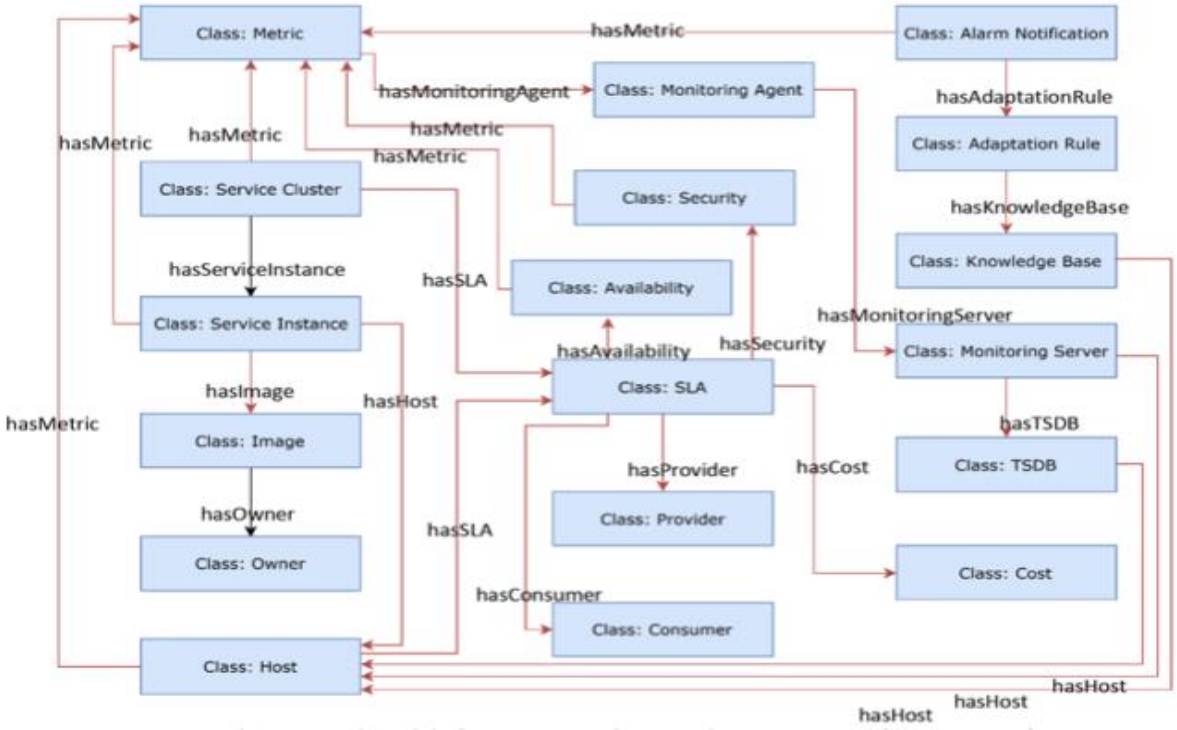

Figure 10: Resource Map

3.4.2 *Define resource metrics*

Figure 10 shows the class structure associated with the metric on the resource model defined by the platform. It defines the resource and metric to be monitored based on the metric of each class specified in Resource Map of D2.1. The key elements of the resource model are:

1. Host: fog nodes that can host services
2. Service Instance: Microservices (K8s Pod) that make up an application.
3. Service Cluster: An application composed of Service Instances
4. Availability: How to check the availability of each resource and service.
5. Security: Security flaw detection method that can occur when storing/transmitting data
6. Alarm Notification: notification information for each monitoring metric

3.4.3 *Resource Monitoring System Diagram*

DECENTER is a platform that provides allocation and sharing of resources between cloud-fog and cloud-cloud in a distributed cloud system environment. DECENTER's Resource Exchange Broker (REB), SLA Manager, manages resources that cross the boundaries of the cloud, and allocates the required resources where they are needed. DECENTER Resource Monitoring System monitors resources and provides core resource management functions of services that provide resource status and usage information to resource providers, users, and administrators. Figure 11 shows the DECENTER system structure linked with the monitoring system.

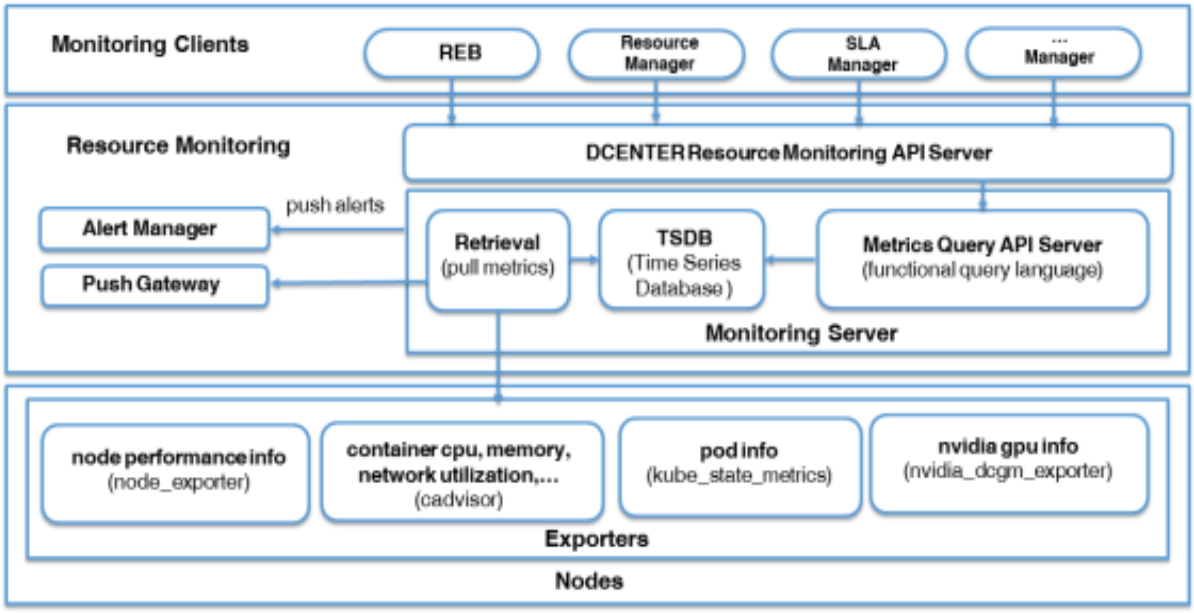

Figure 11: High level resource monitoring components view

The sub-components of the resource monitoring are:

- **Exporters:** which monitors each resource and collects necessary information. Various physical and logical resources defined in D2.1 are collected by each exporter and delivered to the monitoring server.
- **Node_Exporter:** A module that monitors the physical resources of the nodes, as shown in Figure 12.

Category	Resource	Names	Examples
Utiliztion	Node CPU	node_cpu	sum(rate(**node_cpu**{mode!="idle",mode!="iowait", mode!~"^(?:guest.*)$"}[5m])) BY (instance)
	Node Memory	node_memory_MemAvailable node_memory_MemTotal kube_node_status_capacity_memory_bytes kube_node_status_allocatable_memory_bytes	1 - sum(**node_memory_MemAvailable**) by (node) / sum(**node_memory_MemTotal**) by (node) 1- sum(**kube_node_status_allocatable_memory_bytes**) by (exported_node) / sum(**kube_node_status_capacity_memory_bytes**) by (exported_node)
Saturation	Node CPU	node_load1	sum(node_load1) by (node) / count(node_cpu{mode="system"})by (node) * 100
Errors	Node Memory	node_edac_correctable_errors_total node_edac_uncorrectable_errors_total node_edac_csrow_correctable_errors_total node_edac_csrow_uncorrectable_errors_total	

Figure 12: e.g. Use for Node CPU and Memory

- **Container_Exporter (cadvisor):** Module that monitors virtual container resources. The cadvisor is built into Kubernetes to collect metrics from container virtual resources, as shown in Figure 13.

Category	Resource	Names	Examples
Utiliztion	Container CPU	**container_cpu_usage_seconds_total**	sum(rate(**container_cpu_usage_seconds_total**[5m])) by (container_name)
	Container Memory	**container_memory_usage_bytes** **container_memory_working_set_bytes**	sum(**container_memory_working_set_bytes**{name!~"POD"}) by (name)
Saturation	Container CPU	**container_cpu_cfs_throttled_seconds_total**	sum(rate(**container_cpu_cfs_throttled_seconds_total**[5m]) by (container_name)
	Container Memory	Ratio of: **container_memory_working_set_bytes / kube_pod_container_resource_limits_memory_bytes**	sum(**container_memory_working_set_bytes**) by (container_name) / sum(label_join(**kube_pod_container_resource_limits_memory_bytes**, "container_name", "", "container")) by (container_name)
Errors	Container Memory	**container_memory_failcnt** -- Number of memory usage hits limits. **container_memory_failures_total** -- Cumulative count of memory allocation failures.	sum(rate(**container_memory_failures_total**{type="pgmajfault"}[5m])) by (container_name)

Figure 13: e.g. Use for Container CPU and Memory

- **K8s_Exporter (K8s API Server, kube-state-metrics):** Kubernetes provides metrics for various resources, including performance and status of virtual containers and application resources running on them. DECENTER monitoring system uses various metrics provided by K8s to understand the status of system resources and utilizes them, show in Table 6. The contents of the metric are enormous and can be collected and used selectively according to the purpose.

Table 6: Monitoring metrics used by DECENTER

CronJob Metrics	ResourceQuota Metrics
DaemonSet Metrics	Service Metrics
Deployment Metrics	StatefulSet Metrics
Job Metrics	StorageClass Metrics
LimitRange Metrics	Namespace Metrics
Node Metrics	Horizontal Pod Autoscaler Metrics
PersistentVolume Metrics	Endpoint Metrics
PersistentVolumeClaim Metrics	Secret Metrics
Pod Metrics	ConfigMap Metrics
Pod Disruption Budget Metrics	Ingress Metrics
ReplicaSet Metrics	CertificateSigningRequest Metrics
ReplicationController Metrics	VerticalPodAutoscaler Metrics

- **GPU_Exporter (dcgm_exporter):** DECENTER must be provided with allocation and monitoring functions for GPU resources to support AI application services. This study provides a function to collect metrics on NVIDIA GPU resources using NVIDIA Data Center GPU Manager (DCGM). The monitoring of GPUs is currently vendor-dependent and requires the development of a separate Exporter module when using other types of GPUs, as shown in Figure 14.

Category	Name	Use Case	Granularity	Frequency
Utilization	**Power usage**	Proxy for load on the GPU	Per GPU	Per second
	Power limit	Maximum GPU power limit	Per GPU	Per second
	GPU Utilization	GPU utilization rate [0.0 ~ 1.0)	Per GPU	Per second
GPU Memory	**GPU Total Memory**	Total GPU Memory, in bytes	Per GPU	Per second
	GPU Used Memory	Used GPU Memory, in bytes	Per GPU	Per second
Count GPU & CPU	**Request count**	Number of inference requests	Per model	Per request
	Execution count	Number of model inference executions Request count / Execution count = Avg dynamic request batching	Per model	Per request
	Inference count	Number of inferences performed (one request counts as "batch size" inferences)	Per model	Per request
Latency GPU & CPU	**Latency: request time**	End-to-end inference request handling time	Per model	Per request
	Latency: compute time	Time a request spends executing the inference model (in the appropriate framework)	Per model	Per request
	Latency: queue time	Time a request spends waiting in the queue before being executed	Per model	Per request

Figure 14: Available metrics

- **User_defined_Exporter:** For collecting metrics that are not provided by the vendor or the ready-made library, it can be created and exported the Exporter module separately.
- **Monitoring Server:** DECENTER Monitoring Server stores the metrics of resources collected in each Exporter in TSDM (Time Series Database) format. Stored resource metrics are provided as standardized query behavior for easy use by monitoring clients such as REB and SLA Manager.
- **Retrieval Module:** Metrics collected through the Exporter are collected into the server's DB by the Retrieval Module. Collected by pull method, it is designed to minimize the overhead of collecting metrics in each exporter. If the resource information cannot be pulled, a Push Gateway can be configured in the middle, and the Push gateway enables monitoring of resources in various network environments.
- **Query API Server (PromQL):** DECENTER uses the Prometheus open source module as a core structure for resource monitoring. Prometheus defines and uses PromQL as its own Query language that works with TSDM. This study utilizes PromQL to provide a standard interface to monitoring clients.
- **DECENTER Resource API Server:** "Resource API Server" is an interface defined for cross-border convergence of DECENTER. It is an interface that is not clearly defined so far. In order to reflect the variable monitoring interface requirements in the cross-border convergence environment, an interface including user-defined functions is provided, as shown in Figure 15.

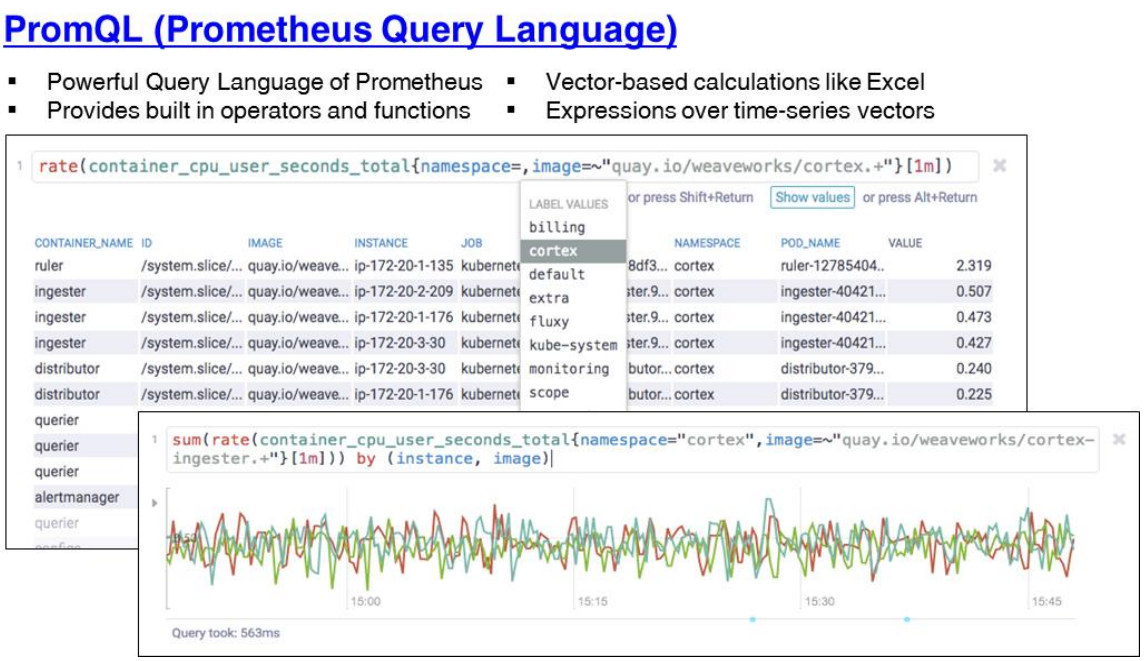

Figure 15: Prometheus Query Language

- **Monitoring Clients:** DECENTER's REB (Resource Exchange Broker) and SLA Manager manages resources that cross the boundaries of the cloud and allocates the required resources where they are needed. The following figure is an example of the operation of the Monitoring module that works with REB and SLA Manager, as shown in Figure 16.

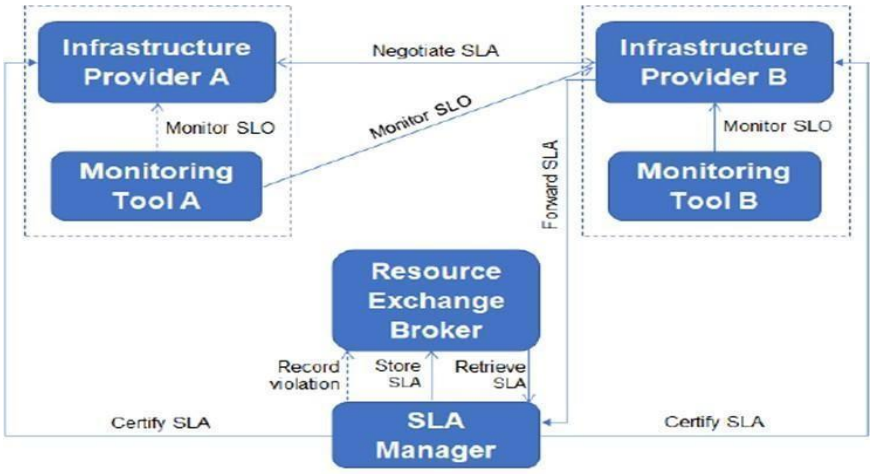

Figure 16: Relationship between entities for SLA management

- **Alert manager:** DECENTER monitoring system provides push alarm function. Through this, it is possible to provide alarm functions for resource abnormalities to various clients including E-mail and SNS.

- **Data Visualization:** Monitoring module provides data view to manage collected re- source information efficiently. It provides a view based on Grafana open source that provides various views. A separate management UI can be configured according to future needs.

Monitor cross-platform shared resources

The DECENTER monitoring system has also the possibility to monitor cross-platform shared resources, from two different viewpoints: IaaS and PaaS.

- **IaaS-based shared resource monitoring:** IaaS-based shared resources consist of lending of cross-platform nodes, as shown on Figure 17. The node may be a newly created VM or an existing VM or physical device. The following figure is an example of collecting resource metrics after cross-platform node borrowing. When platform A borrows a node from platform B, the rented node is managed by platform A, and platform A collects resource metrics of the rented node like a normal node. Because platform B provides a node and loses control of that node, it collects only the service-hostable state of the node, not resource metrics about its internal state. The service hosting status of the rented node collected by Platform B may be used to verify SLA violations.

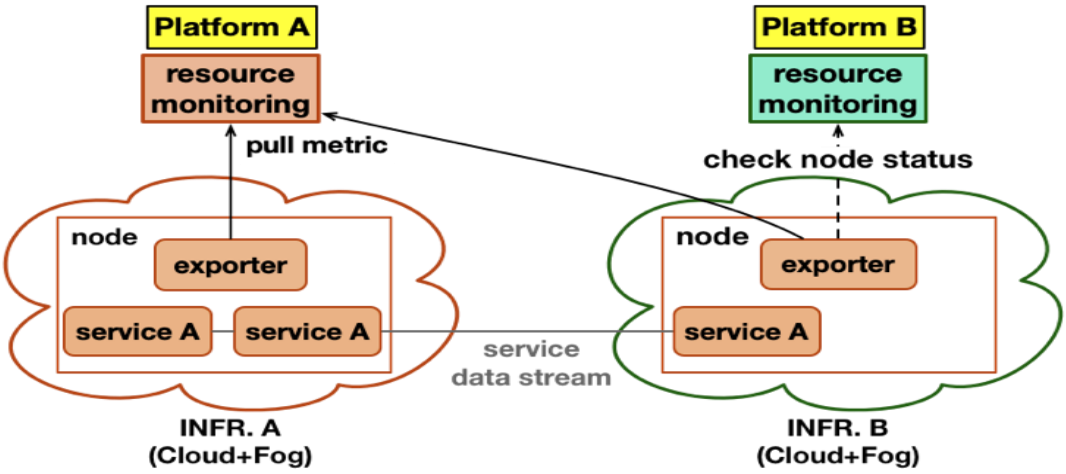

Figure 17: IaaS based shared resource monitoring

- **PaaS-based shared resource monitoring:** PaaS-based shared resources are achieved by distributing services to nodes managed by other platforms, as it is shown on Figure 18. Basically, resource metric collection of each platform collects all metric of node managed by itself regardless of shared resources and collects metrics of shared resources between other platforms as necessary between resource monitoring. Metric sharing between resource monitoring can be implemented by utilizing Prometheus federation.

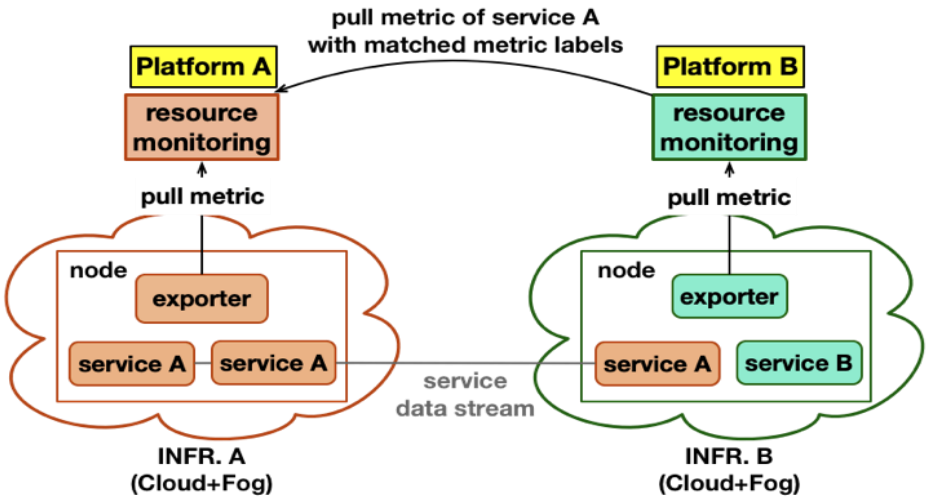

Figure 18: PaaS based shared resource monitoring

- **Share metrics between resource monitoring:** Just as platforms share resources through REBs, resource metrics can be shared through resource monitoring. We use Prometheus federation to provide this functionality. Prometheus federation is the ability of one Prometheus server to recollect metrics from an- other Prometheus and can also selectively collect metrics using regular expressions for metric names, labels in the form of keys and values.
- **Information Required for Sharing Resource Metrics Across Platforms:** In order to perform IaaS / PaaS-based shared resource monitoring, the fog platform must submit some information to the resource monitoring system after cross-platform shared resources are created. When an IaaS-based shared resource is created, resource monitoring on the platform that provided the shared resource (node) does not know which IP and port should be used to determine the node's running state. Therefore, information about the public IP of the node and the port number of the open exporter should be submit- ted to the resource monitoring of the platform that provided the shared resource (node). If a PaaS-based shared resource is created, the platform that requested the shared re- source (service deployment) must access resource monitoring on another platform to get metrics for the shared resource but cannot know which IP and port should collect the metrics for the shared resource. Therefore, the service cluster name to distinguish the shared resource metric among the public IP, open port number, and multiple metrics of the resource monitoring server of the other platform on which the shared resource is running must be provided
- **Querying Resource Metrics**: The monitoring system collects data by exporters in the form of time series. The time series are built through a pull model: the monitoring server queries a list of data sources (exporters) at a specific polling frequency. Each of the data sources serves the current values of the metrics for that data source at the endpoint queried. The monitoring server then aggregates data across the data sources. Data is stored in the form of metrics, with each metric having a name that is used for referencing and querying it. Each metric can be drilled down by an arbitrary number of key-value pairs (labels). Labels can include information on the data source (which server the data is coming from) and other application-specific breakdown information such as the HTTP status code (for metrics related to HTTP responses), query method (GET versus POST), endpoint by the multi-

dimensional way. For more information on the querying of Prometheus, please refer to the official API[2].

3.5 Robustness and Security

This section builds on the advances by year 2, on the task for security and robustness (started in this year).

3.5.1 Security of the microservices

Despite all the benefits of the software architecture based on microservices, the containers' security is often neglected in literature. Security should not be independently implemented by every single container, but it should be uniform for all microservices, in order to offer an overall perspective of what is happening in the system.

The FMT (Fleet Management System) presented in the UC Robotic Logistic in Deliverable 2.1. [10] is a centralized system. It can manage and dispatch all the movements and actions of the robots.

When designing the security of a microservice's architecture, the common characteristics of the application are the communications between the robots and the FMT inside the infrastructure.

3.5.1.1 L-ADS

In our case study, we care about the existing internal and external network interactions. To monitor and evaluate these, we introduce an asset called L-ADS (Live Anomaly Detection System). This asset was presented in [11], and it has been evolved for achieving a better performance. L-ADS captures the traffic in a network and predicts the type of connections (malicious, legit) using algorithms of Deep Learning.

3.5.1.2 Data from Netflow

To capture the network traffic, we use Netflow [12], a protocol developed by Cisco. This protocol is designed at package level, including features such as source IP/port, destination IP/port, protocol, bytes, packets, etc.

Date first seen	Duration	Proto	Src IP Addr	Src Pt	Dst IP Addr	Dst Pt	Packets	Bytes	Flows	Flags	Tos
2017-04-12 00:02:16.977	0.000	TCP	192.168.220.11	34079	192.168.100.5	445.0	1	66	1	.A....	0
2017-04-12 00:02:20.290	0.000	TCP	EXT_SERVER	8082	192.168.200.9	53089.0	1	200	1	.AP...	32
2017-04-12 00:02:20.272	0.224	TCP	192.168.200.9	53089	EXT_SERVER	8082.0	2	319	1	.AP...	0
2017-04-12 00:02:20.901	0.003	TCP	192.168.220.7	50015	192.168.100.5	445.0	2	174	1	.AP...	0
2017-04-12 00:02:20.902	0.000	TCP	192.168.100.5	445	192.168.220.7	50015.0	1	108	1	.AP...	0
2017-04-12 00:02:24.876	0.500	TCP	192.168.100.5	445	192.168.220.16	33572.0	2	178	1	.AP...	0
2017-04-12 00:02:24.878	0.498	TCP	192.168.220.16	33572	192.168.100.5	445.0	3	240	1	.AP...	0

Figure 19: Example extracted from softflow

To evaluate the network flow, it is necessary to use an implementation of Netflow, for that we used Softflowd. Softflowd supports data export of traffic flows (between 2 IPs) in different versions of Netflow (1, 5 and 9).

Figure 19 shows an example of the generated structure by Softflowd. The information provided by Softflowd has the following structure:

- **Date first seen.** Is the start time of the flow.

[2] https://prometheus.io/docs/prometheus/latest/querying/api/

- **Duration.** Duration of the flow.
- **Proto.** Protocol used in the connection.
- **Src IP Addr.** Source IP address.
- **Src Pt.** Source Port.
- **Dst IP Addr.** Destination IP address.
- **Dst Pt.** Destination Port.
- **Flags.** TCP flags of the connection.
- **Packets.** The number of packets in the flow.
- **Bytes.** The number of bytes in the flow.
- **Tos.** Type of service.

3.5.1.3 Architecture of the solution

Figure 20 shows how the preliminary version of the architecture will be implemented into the FBK cluster. The architecture consists of two main parts: the Softflowd container and the Netflow Collector.

The Softflowd container captures the whole traffic generated in FogAtlas. The container is responsible for capturing and sending the traffic to a visible IP/port. This information will be retrieved by the Netflow Collector.

The Netflow Collector will be replaced by the asset L-ADS as a container in future versions. The L-ADS uses the traffic flows as an input to evaluate if the flows are anomalous or not.

The asset L-ADS (Figure 21) has some modes:

- **Clean**. Delete all models stored in the database.
- **Capture**. Store in .json files the Netflow traffic received.
- **Train**. Generate the models and store them in the database.
- **Monitor**. Evaluate in real time the Netflow traffic received.
- **Predict**. Test the models against text files.

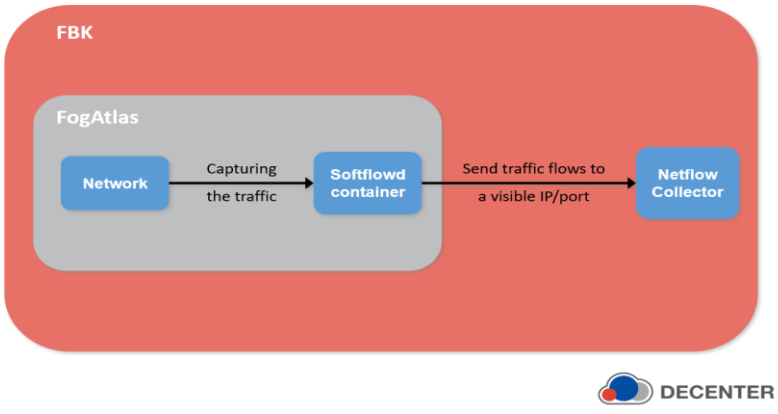

Figure 20: Architecture of L-ADS

It is possible to receive as input for the training of the models the .json files from the capture mode, .csv files or nfcapd files.

The component responsible for training the models and generating the output is the Brain. There are two possible outputs: warnings or alarms. The warnings are used when the flows

have a certain IP that is not modelled or when there is not a valid model. The alarms are the output when the flows are classified as an anomalous connection.

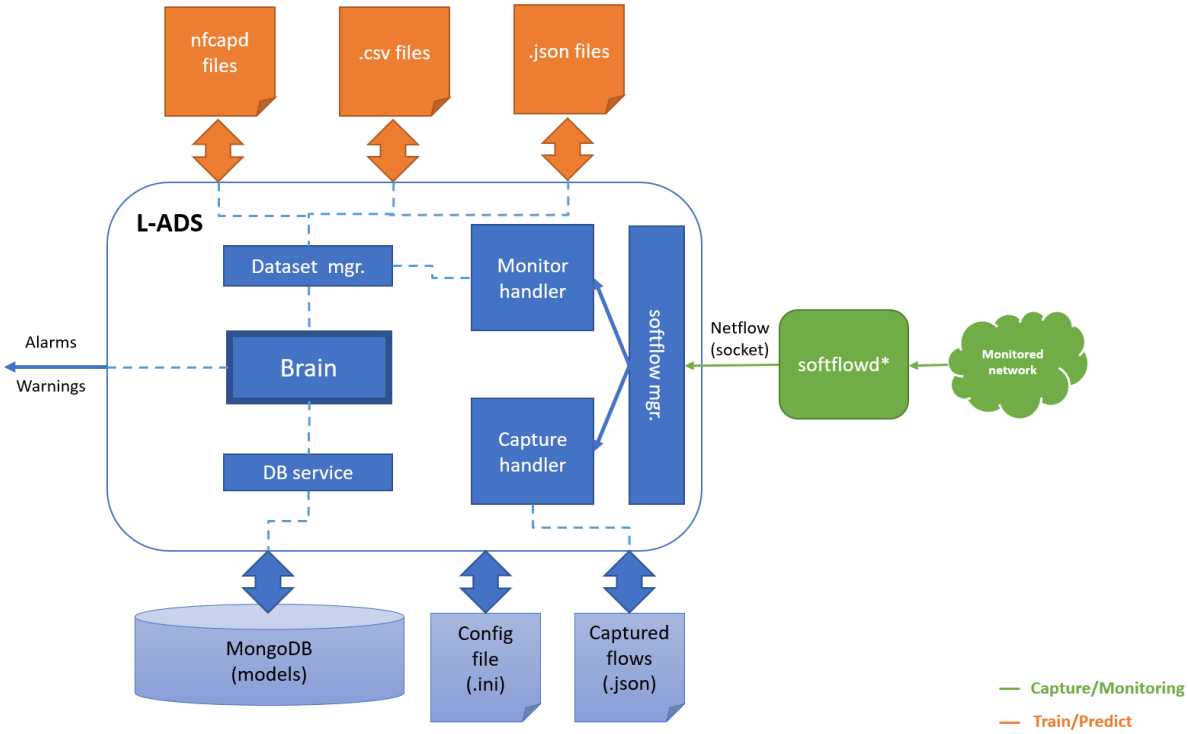

Figure 21: The asset L-ADS

3.5.1.4 Versions of L-ADS

The L-ADS asset has two available versions and it will be chosen according to their performance in the DECENTER environment.

L-ADS version 1

The first version of L-ADS attempts to independently model the different IPs. Once the dataset generated by Softflowd was retrieved, it is possible to get both the source and destination IPs. For every IP, it can be filtered the flow of this IP.

This implementation of the L-ADS reduces the number of flows to each subset, which improves efficiency. By filtering the dataset, we filter the flow of a single IP, which reduces the noise.

However, as it generates *k* subsets (*k* is the number of unique IPs), then *k* associated algorithms are generated simultaneously, which will be trained with a subset in a local environment. In this case, it is a local environment from a certain IP. In addition, a further issue can arise related to some subsets containing insufficient flows to train the associated algorithm, which is translated into a bad performance.

In order to not lose information, we create a new dummy variable called ***is_source*** if the IP appears as source or as destination (*1* if Source, *0* if Destination).

Figure 22 shows what would happen if we filter the dataset of Figure 21 using the next IP: *192.168.100.5*.

Date first seen	Duration	Proto	Src IP Addr	Src Pt	Dst IP Addr	Dst Pt	Packets	Bytes	Flows	Flags	Tos	is_source
2017-04-12 00:02:16.977	0.000	TCP	192.168.220.11	34079	192.168.100.5	445.0	1	66	1	.A....	0	0
2017-04-12 00:02:20.901	0.003	TCP	192.168.220.7	50015	192.168.100.5	445.0	2	174	1	.AP...	0	0
2017-04-12 00:02:20.902	0.000	TCP	192.168.100.5	445	192.168.220.7	50015.0	1	108	1	.AP...	0	1
2017-04-12 00:02:24.876	0.500	TCP	192.168.100.5	445	192.168.220.16	33572.0	2	178	1	.AP...	0	1
2017-04-12 00:02:24.878	0.498	TCP	192.168.220.16	33572	192.168.100.5	445.0	3	240	1	.AP...	0	0

Figure 22: Filtered dataset

L-ADS version 2

In this implementation of L-ADS, the Softflowd dataset is not filtered, and a single algorithm will be then trained for all the IPs. In other words, the algorithm tries to get the similarities between all the features except for the IPs.

Unlike the first version, in that case there is just one dataset with an algorithm associated. Consequently, it is possible to have some noisy flows, but it will be faster than the first version. For example, we have a dataset from a home network with different devices such a smart TV, a printer or a mobile phone. The Softflowd dataset will contain flows from different devices and different behaviors from each other. The printer's flows may not look like the mobile phone's flows.

3.5.1.5 L-ADS model

The remainder of the L-ADS generation model process is the same for the two versions.

To generate more features and improve the performance of the algorithms, L-ADS creates four new features: ***Packets_speed*** *(*the *number of Packets* divided by the *Duration),* ***Bytes_speed*** *(*the *number of Bytes* divided by the *Duration),* ***Packets_per_flow*** and ***Bytes_per_flow***. Then, the categorical features ***Proto*** and ***Flags*** are transformed into dummies variables. Finally, the standardization is necessary to get a valid dataset, called *train dataset,* to use as input in the algorithm.

The L-ADS uses a Deep Learning algorithm known as **Autoencoder**. This algorithm extracts the general characteristics of the dataset. During the training of Autoencoders, it is important to use legit traffic, which is used to create a "legit" flow profile.

With this process of compressing and decompressing the same information, the algorithm learns about the general characteristics from the train dataset.

The Autoencoder consists of two different parts:

- **Encoder**. Retrieves the input dataset and compresses the information. The input dimension is the same as the train dataset dimension, while the output dimension is compressed.

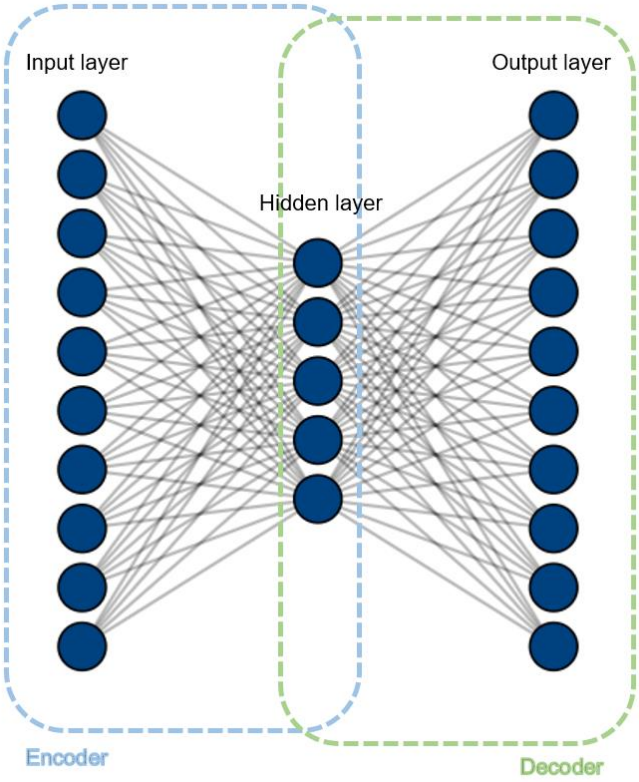

Figure 23: Architecture of the encoder

- **Decoder**. Retrieves the compressed information and decodes this information to obtain the initial train dataset (i.e. the input and the output is the same, the train dataset). This section attempts to reconstruct the compressed information.

Figure 23 shows the architecture of the Autoencoder and the different sections: the encoder and the decoder.

When making a categorization of traffic, the flow goes through the same transformations as the training dataset. Once the flow has been transformed, it will be evaluated by the Autoencoder generating a reconstruction of the flow.

To distinguish if it is an anomalous connection or not, we use the following function called ***mean square error (MSE)***.

$$MSE = \frac{1}{n}\sum_{i=1}^{n} \quad (Y_i - Y_i')^2,$$

where n is the number of features, Y_i Is the value of the i-th feature in the connection and Y_i' Is the value of the i-th feature in the reconstructed connection.

This function is used to calculate a numerical value associated with the “normality” of the connections. If the result is bigger than a specific value or threshold, the connection will be categorized as anomalous. Otherwise, it is considered that the input connection and the reconstructed connection are similar. This threshold is set through experimentation. In a supervised learning problem, the threshold could be adjusted to improve the performance. In the opposite, in an unsupervised learning problem this threshold must be predeterminate with the help of the supervised learning problem threshold.

3.5.1.6 Testing the solution – CIDDS

In order to check the validity of the Autoencoder as a classification algorithm, we used the CIDDS-001 dataset (Figure 24), which is part of the **CIDDS** (Coburg Intrusion Detection Data Sets) [13], a set of labelled Flow-based public datasets. TheCIDDS-001 dataset simulates data from a small business over the course of a month. It includes data from legit connections and different attacks such as Brute Force, DoS (Denial of Service) and port scan.

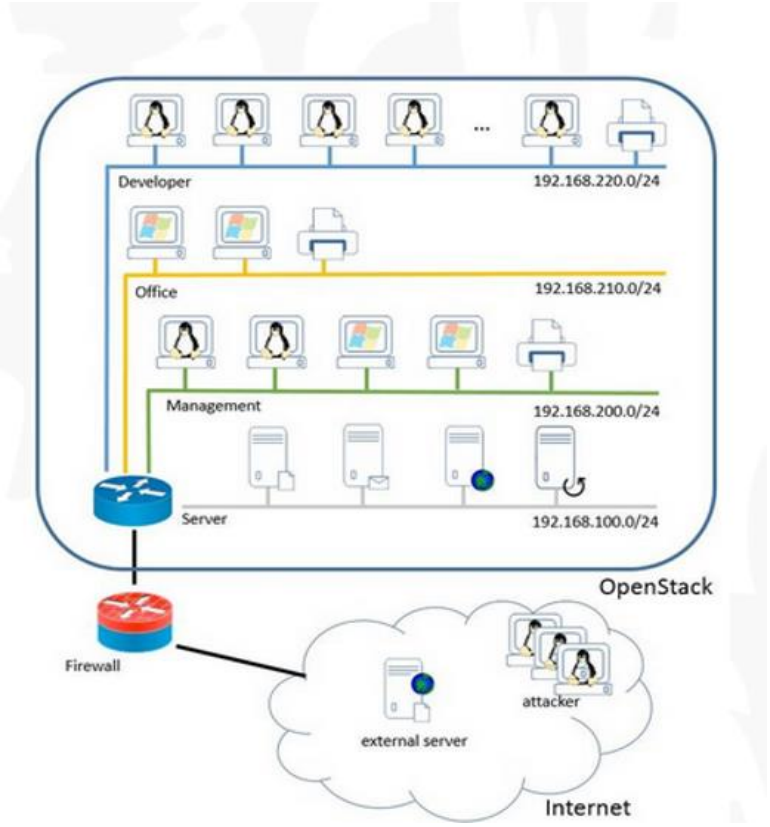

Figure 24: CIDDS-001 setup

The dataset has been split into two subsets: Train and Test. We used the data from week 4 as Train because it has not any attacks and the data from week 2 for the Test subset, because the algorithm need a normal behavior dataset as training dataset shows the number of connections per day in week 4 (Train subset) and week 2 (Test subset). Given that our algorithm classifies the connections as legit or anomalous, we treat all the attacks (victim or attacker) as anomalous connections.

Given that the CIDDS is labelled, we could adapt the models as a supervised learning problem and as an unsupervised learning problem when it is not used the labels. The unsupervised learning problem should be a scenario more realistic.

For our tests, we only use the internal connections (the IPs with the first octets “192.168.”), because they are a representative sample of our expected real-life scenario.

Figure 25 shows a sample of a supervised learning and unsupervised learning problem.

- **Supervised Learning**

SUPERVISED LEARNING PROBLEM

Date first seen	Duration	Proto	Src IP Addr	Src Pt	Dst IP Addr	Dst Pt	Packets	Bytes	Flows	Flags	Tos	class
2017-03-23 10:07:29.854	0.000	TCP	192.168.210.3	993	192.168.220.15	36867.0	1	54	1	.A.R..	0	1
2017-03-23 10:08:11.055	0.001	UDP	192.168.210.3	138	192.168.210.255	138.0	2	517	1	.....	0	0
2017-03-23 10:13:25.888	0.000	TCP	192.168.220.15	36867	192.168.210.3	110.0	1	58	1	....S.	0	1
2017-03-23 10:13:25.888	0.000	TCP	192.168.220.15	36867	192.168.210.3	110.0	1	58	1	....S.	0	1

UNSUPERVISED LEARNING PROBLEM

Date first seen	Duration	Proto	Src IP Addr	Src Pt	Dst IP Addr	Dst Pt	Packets	Bytes	Flows	Flags	Tos
2017-03-23 10:07:29.854	0.000	TCP	192.168.210.3	993	192.168.220.15	36867.0	1	54	1	.A.R..	0
2017-03-23 10:08:11.055	0.001	UDP	192.168.210.3	138	192.168.210.255	138.0	2	517	1	.....	0
2017-03-23 10:13:25.888	0.000	TCP	192.168.220.15	36867	192.168.210.3	110.0	1	58	1	....S.	0
2017-03-23 10:13:25.888	0.000	TCP	192.168.220.15	36867	192.168.210.3	110.0	1	58	1	....S.	0

Figure 25: Supervised and unsupervised learning problems

In this case, the dataset is labelled. The L-ADS has two possible versions and they are evaluated with the same labelled dataset. The test subset has 2,232,715 internal connections.

The L-ADS have filtered 31 IPs, all of them are internal connections. The performance of the Autoencoder with the test subset is:

Table 7: Metrics Supervised Learning

L-ADS version	Nº Errors	Accuracy	F1-Score normal	F1-Score anomalous
1	30,704	0.97	0.78	0.33
2	140,546	0.94	0.81	0.96

- **Unsupervised Learning**

The unsupervised learning scenario requires for the threshold to be determined before the training of the Autoencoder. For the L-ADS version 2 (no filters), we used the results of the supervised learning project to determine the value of the threshold (*0.1)*. On the other hand, the L-ADS version 1 was not defined because this threshold depends on the IP. The metrics have been evaluated using different thresholds and the best performance of the L-ADS version 1 reach with the value of *threshold=10.0.*

Table 8: Metrics Unsupervised Learning

L-ADS version	Nº Errors	Accuracy	F1-Score normal	F1-Score anomalous

1	978,778	0.91	0.86	0.28
2	140,546	0.94	0.81	0.96

3.5.1.7 Next Steps

Once the L-ADS is evaluated using the CIDDS dataset, we will evaluate our algorithm using real environment data from the DECENTER project. We plan to extract a dataset of the utilization of DECENTER in the use case of Robotnik. This dataset will show internal and external connections between the robots, including:

- a subset of data with only normal connections, and
- one or more subsets of data with simulated attacks to the system to check if the performance obtained in CIDDS is maintained with the simulated dataset. The attack will be simulated using the tool Kali Linux[3] and the simulated attacks are: Brute Force, Port Scan, Ping Scan and DoS (Denial of Service).

Once this dataset is retrieved, then the Softflowd tool will be deployed together with the rest of the DECENTER infrastructure, to enact experiments on real-life demos. These demos will be evaluated by the L-ADS to check again the performance.

Finally, we will deploy the rest of the L-ADS components on top of the decenter platform and run again the demo.

3.5.2 Security of the Fog nodes

One of the security aspects that are being taken by DECENTER is the defence against cyber threats in the fog environment. We then propose an **Intrusion Prevention System (IPS)** for the fog nodes with minimal requirements in terms of computing resources. The goal is to design a stand-alone defence system that can perform network traffic monitoring and security policy enforcement, without compromising the operations of other services running on the node where it is executed. This is challenging, since the hosts acting as fog nodes might possess limited computing and memory resources.

In this direction, we have done some preliminary work to evaluate the capabilities of Deep Learning (DL) and recent Linux Kernel technologies in detecting Distributed Denial of Service (DDoS) attacks with high accuracy and in filtering such malicious traffic with limited CPU load overhead. The results of this study have demonstrated that a combination of a Convolutional Neural Network (CNN) [1] and kernel-space software modules based on the extended Berkeley Packet Filter (eBPF) [2][3] and the eXpress Data Path (XDP) [2][4] can be very accurate and efficient in classifying the traffic either as benign or DDoS and in blocking the sources of the attacks.

The overall architecture of the proposed DDoS defence system is depicted in Figure 26. In the user space, the traffic attributes such as packet size, IP flags, TCP windows size, ICMP type etc. are extracted from the packets and organized in arrays and then used as input for the CNN. The output of the CNN is the list of malicious source IP addresses detected in a pre-defined time interval. This output is used to fill a blacklist of source IP addresses implemented as an eBPF hash map [5]. The packet filtering operation is executed in the kernel space by an

[3] https://www.kali.org/docs/introduction/what-is-kali-linux/

eBPF/XDP program that matches the incoming packets' source IPs with the blacklist of malicious addresses.

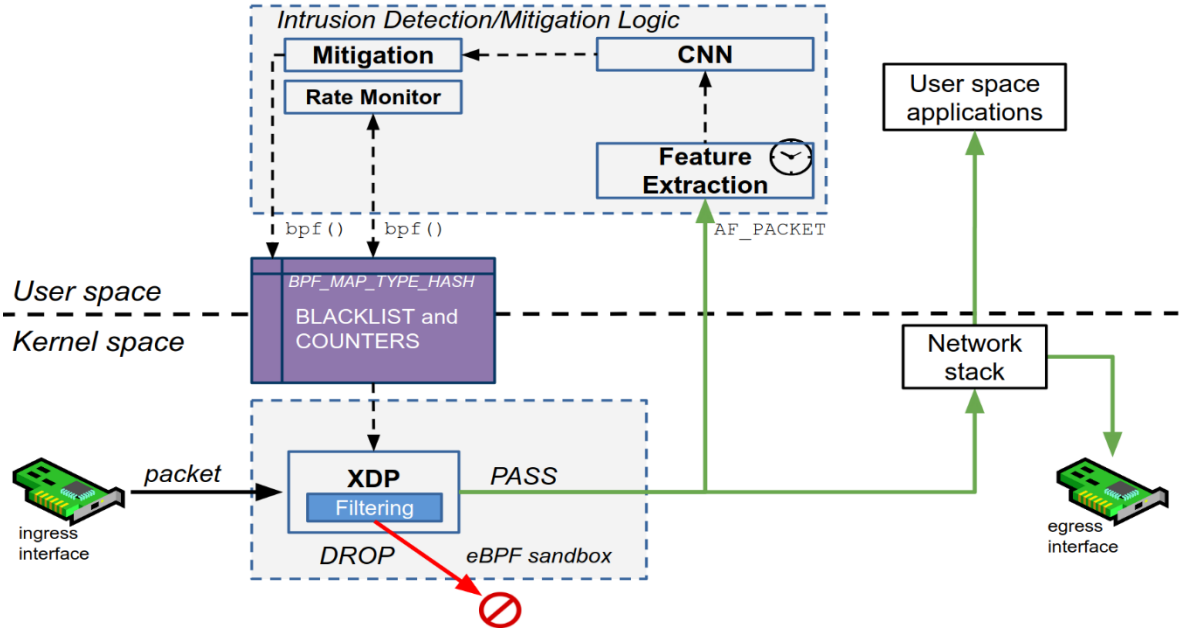

Figure 26: Architecture of the DDoS defence system

The CNN leverages a spatial representation of traffic flows to distinguish between patterns of malicious DDoS behaviour and benign traffic. This approach suits well resource-constrained devices such as fog nodes, as it requires a limited number of weights compared to state-of-the-art solutions, while producing high classification accuracy.

On the other hand, the combination of XDP and eBPF can be exploited to efficiently process the network traffic and to drop malicious packets before they reach the system TCP/IP stack and the user space applications, with minimal consumption of the node's CPU resources. This aspect is particularly relevant in the case of volumetric DDoS attacks, in which the mitigation involves inspecting large volumes of traffic and comparing the traffic's characteristics with the information contained in large databases. Executing this operation before the traffic enters the TCP/IP stack not only saves CPU and memory resources (used by the system for packet decapsulation), but it also reduces the latency overhead on the traffic in its path inside the node from the ingress network card to the final destination (either an application running on the node or an egress network card).

The following sections present the above components in greater detail.

3.5.2.1 Traffic filtering

The first program encountered in the DDoS defence pipeline is the *Filtering* module, which matches the incoming packets against the content of a blacklist and drops them if the result is positive; surviving packets are continue toward their final destination through the network stack of the operating system.

As represented in Figure 27, the filtering program is implemented in the kernel space of the operating system using eBPF and XDP technologies, available in the kernel of recent Linux Operating systems (kernel version 4.15 and above). eBPF, which stands for extended Berkeley Packet Filter, is a kernel packet filtering mechanism used to implement network

utilities such as tcpdump. eBPF programs can be safely injected in various kernel subsystems for tracing (e.g., kprobes, tracepoints, etc.) and networking purposes (through the XDP and Traffic Control (TC) hooks). eBPF programs share information with user space applications through data structures called *eBPF maps*. In our prototype, an eBPF map is used to implement a blacklist containing the IP addresses of the sources of DDoS attacks. The blacklist is populated with the output of the CNN and is used to block malicious traffic.

3.5.2.2 Feature extraction

The *Feature Extraction* module monitors the incoming traffic and collects relevant packet attributes required by the CNN. Being placed right after the traffic filtering module, it receives only the (presumed) benign traffic that has not been previously dropped.

The feature extraction is performed in user-space by means of a raw socket of type `AF_PACKET` [6], which is used to intercept the incoming network traffic not previously blocked by the filtering module. Of course, this module takes a copy of the packets, while the original traffic data is passed to the network stack of the host, which forwards it to the target user space applications or to the next hop in the path towards the final destination.

The output of this module is sent to the CNN for traffic classification every a pre-defined time window (e.g., 10 seconds). The output is a collection of arrays, where each array represents a single traffic flow, as identified by the 5-tuple:

```
<source_IP, source_port, dest_IP dest_port, protocol>
```

Which means that each array collects the packets with the same source and destination IP addresses, source and destination L4 port and same protocol type (e.g., TCP, UDP, ICMP).

In the array, the lines are the packets of a flow in chronological order, while the columns represent the features extracted from each packet.

	Pkt #	Time (sec)[1]	IP Flags	TCP Flags	TCP Window Size	UDP Len	ICMP Type
Packets	0	0	0x4000	0x018	1444	0	0
	1	0.092	0x4000	0x018	510	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	j	0.513	0x4000	0x010	1444	0	0
Padding	$j+1$	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	n	0	0	0	0	0	0

Figure 27: Array-like representation of a traffic flow

Figure 27 shows an array-like representation of a TCP flow as collected in a time window. The flow is composed of *j* packets, while *n* is the maximum number of packets that the array can contain. Flows longer than *n* are truncated, while shorter flows are zero-padded at the end.

The features extracted from each packet are shown in Figure 27 from *Time*, which represents the relative capture time of a packet with respect to the time of the first packet in the array, to *ICMP Type*. These are the features that have been used to train the CNN with the dataset provided by the Canadian Institute for Cybersecurity of the University of New Brunswick

(UNB), Canada [7], and with a custom dataset containing SYN flood attacks generated at the FBK laboratories.

For our initial prototype documented in [8], we used a larger set of packet attributes for evaluation purposes. For those experiments, we used tshark to extract the attributes from the packets available in the static UNB datasets. Beyond the header fields of the packets, *tshark* presents additional information such as the list of protocols and the highest layer detected in a packet. As shown in [8], these attributes are helpful for increasing the classification accuracy. However, *tshark* is expensive in terms of CPU resources, and can represent a bottleneck in the case of large volumes of live traffic (like under DDoS attacks).

Instead, the current features extraction process reads the raw header of packets without the preliminary pre-processing performed by *tshark.*

3.5.2.3 Traffic Classification

The traffic classification module is implemented with a Convolutional Neural Network (CNN), which takes an array-like representation of a traffic flow as input and returns a value between 0 and 1. This value represents the probability *p* of a given flow being part of a DDoS attack. If *p>0.5*, the flow is classified as a DDoS flow, benign otherwise. The architecture of the CNN model is represented in Figure 28.

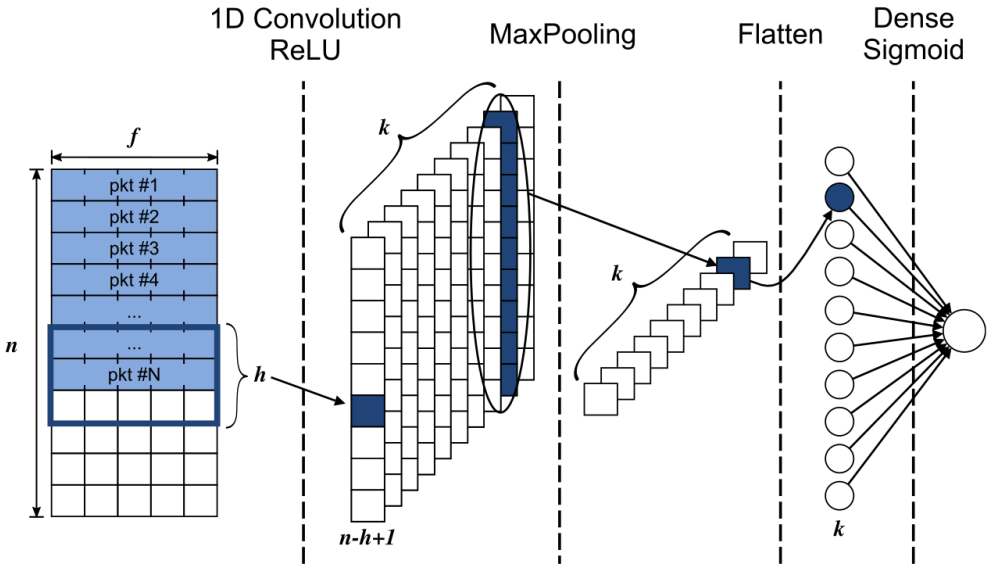

Figure 28: Architecture of the CNN model

In Figure 28, *n*f* is the size of the array-like representation of a flow, *h* is the height of the convolutional filters and *k* is the number of convolutional filters. These are hyper-parameters of the CNN that are tuned at training time. More details on the CNN model architecture are available in [8].

3.5.2.4 Mitigation and rate monitor

The output of the CNN is a list of 5-tuples identifying the traffic flows classified as DDoS during the last time window. The *Mitigation* module populates the blacklist (implemented with a eBPF hash map) with the information contained in this list by calling the `bpf()` system call [9]. In the current implementation, only the source IP addresses of the flows are used, however other fields of the 5-tuple can be used to implement more fine-grained filtering policies.

Every time a packet matches a rule in the blacklist, a counter associated to that rule is increased by one. Such statistics, called Counters, are read by the user space program *Rate Monitor* through the `bpf()` system call, and used to remove from the blacklist the source IPs that are no longer part of a DDoS attack, or that were erroneously classified as malicious by the CNN. The outcome of this process is twofold: first, the restoration of traffic forwarding from legitimate sources and second, a reduced blacklist size, hence shorter query execution time for the Filtering XDP program.

3.5.3 Robustness of the platform

The year 1 layout of the Fog Platform architecture foresaw an administrative domain corresponding to a single Fog Platform cluster (i.e. Kubernetes cluster). This means that, despite the infrastructure administered by an owner could be distributed in different locations (edge nodes, fog nodes, cloud environment etc.) and despite different regions (intended as sets of computational power, devices and services located in a specific geographical position) are comprised in such an infrastructure, the control plane of the Fog Platform is still located in one central place, called Master Region.

Figure 29 shows the layout of this architecture.

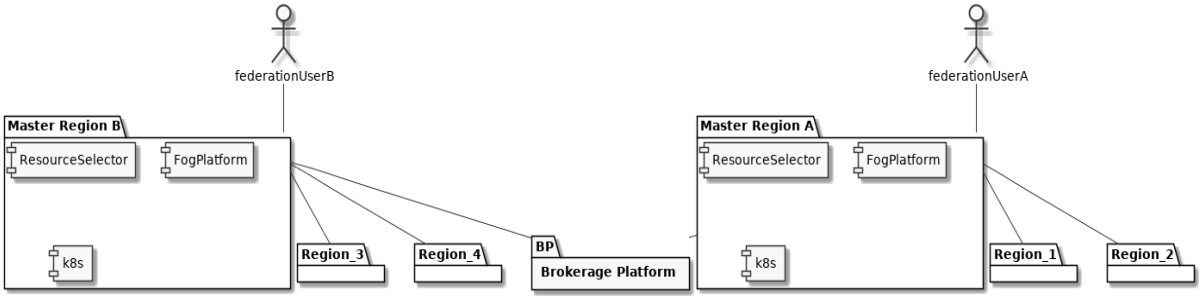

Figure 29: Y1 DECENTER Robustness layout

The limitations of this approach in terms of robustness of the solution are obvious:

- each region is not independent from the master regions: a network problem between the master and a given region or a fault in the master region can tamper the services offered by the region itself;
- if an external client wants to connect directly to a service in a remote region, it could need to pass through the centralized control plane cancelling the advantages of edge computing in terms of latency, bandwidth and data privacy.

More generally the proposed architecture is distributed but not decentralized limiting the resilience to the faults of the entire system.

In order to make the Fog Platform more decentralized and resilient, we propose an evolution of the architecture adopting the concept of federation: each region of an administrative domain corresponds to a cluster of the Fog Platform (i.e. a Kubernetes cluster), independent but federated with the other clusters. The pattern chosen to implement such a federation of clusters leverages the Kubernetes Cluster Federation (KubeFed) project. The following figure depicts the new architecture.

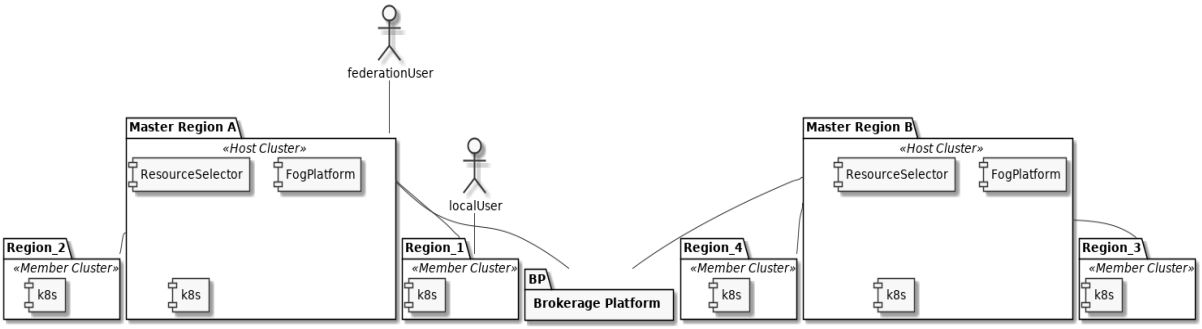

Figure 30: Y2 DECENTER Robustness layout

Following the KubeFed naming convention we will have one or many member clusters, one for each region, where the workload will be scheduled and just one host cluster (i.e. the master cluster of the federation) where the federation control plan will run. The Fog Platform orchestrator (running on the host cluster) will decide in which region/cluster place the workload, while each member cluster control plane will determine on which node, inside the chosen region, to run such a workload. If there is a need to acquire resources from other administrative domains, this can be done in two ways:

- acquiring a single resource to be joined to the host cluster as a new “satellite” region belonging to the same cluster as the host cluster;
- acquiring an entire region to be joined to the federation as a new member cluster

With this solution two objectives are met:

- the architecture is decentralized, and robustness and resilience are improved: even though the host cluster experiences a failure, the different member clusters can continue to operate and process the local workload;
- each member cluster is an independent entity that an external client/user can contact directly when a local processing (at the edge) is requested.

3.5.3.1 Implementation

Apart from the setup of a Kubernetes federation following the KubeFed project, it is needed to enhance both the Orchestrator and the Application Composer components of the Fog Platform (see Annex A) to make them able to submit and to place the workload on a federated infrastructure. For more information on the implementation of the Kubernetes federation implementation, refer to Annex B.

3.5.3.2 Possible failures in Kubernetes and related countermeasures

In this section we briefly summarize the possible failures that can happen in an infrastructure managed by Kubernetes and the countermeasures that can be adopted in order to guarantee the continuity of the service.

Failures in a traditional (i.e. hosted on an IaaS environment) Kubernetes cluster can happen at three different levels:

- infrastructure level. The IaaS environment where Kubernetes is installed should provide services to mitigate catastrophic events (e.g. fire, flood) or failures in the hardware, network and power supply. Generally, this is well managed by public cloud providers thanks to the concept of “availability zones”. Of course, this is more critical in case of private cloud or fog/edge computing environments.

- platform level. This refers to the installation of Kubernetes itself. In order to guarantee a production grade robustness, the Kubernetes master node must be installed in high availability considering a multi-master installation where one master is elected as leader. In case of failure of that master leader, another master node is designed as leader.
- application level. Failures at application level are well handled by Kubernetes itself. It implements a lot of mechanisms able to improve the robustness of a cloud native application: keeping the actual state of an object equal to its desired state, handling rescheduling in case of node failure, allowing to schedule the workload in a more fault tolerant way (e.g. node selector, pod affinity and anti-affinity).

Considering more distributed and heterogeneous environments like the ones foreseen by the fog computing paradigm, failures in the network connectivity among the different nodes of a given cluster becomes more and more relevant. In these environments, as we already discussed in the previous section, having one cluster spread across different geographical locations can increase the risk of failures due to connectivity problems. In this case, having many federated Kubernetes clusters each covering a single region allows reconducting the problem of management of failures to the one of traditional local clusters that we analyzed in the previous bullet list. This means that the same countermeasures applicable for a traditional cluster, hosted in a datacenter and where the network is irrelevant, can be applied also for a single local cluster in a larger federation of clusters. Moreover, the federation defines a sort of “master of master” (the host cluster) that implements the control plane of the whole federation: such host clusters can be deployed in high availability in order to make the system more resilient.

3.6 Brokerage Platform

The brokerage platform has been already described in detail in the Deliverable D3.1. Here we report on the progress made on the implementation of it and on the performance analysis that will support the choice of using a public blockchain infrastructure or rather design a private one to address the requirements associated with the exchange of resources between administrative domains.

As the project execution will go on, a number of factors influencing this choice will become also clearer in terms of what the project solutions will be able to support.

The typology of resources and the complexity of the metadata structures used to describe them will be one of the factors that influence the choice of what to store in a smart contract directly on the blockchain and what to store off-chain. The more complex a smart contract, the more expensive it will be to store it and execute it in a blockchain environment. Also, the frequency of exchanging resources is a factor that influences the overhead costs of using blockchains. Some costs are incurred upon writing smart contracts on a public blockchain; for these reasons, frequency and duration of resource exchange are important elements that dictate what is the lower limit, as a percentage of the value produced by any DECENTER resource exchange, one can afford to lose as an overhead for using the services on a blockchain. It goes without saying that if the cost of setting and executing a smart contract on a public blockchain outweighs what the seller will get in rewards / payment for her/his services, then the whole system becomes economically unsustainable.

Besides frequency of exchanges, one must also consider the delay we are prepared to tolerate when exchanging resources. If the time it takes for the blockchain to authorise and initiate an exchange is too long compared to the duration of the exchange, here as well we are in a situation where the technical solution cannot be used.

3.6.1 Preliminary REB assessment

The first assessment was done checking, on a public blockchain, different options for implementing the DECENTER exchange broker (also referred to as the Marketplace) using Ethereum test network.

As metrics for the user experience, we focused on monetary cost and system performance. To support design decisions, we evaluated different scenarios and illustrate here the results of our evaluations. This analysis is particularly useful to assess the feasibility of adopting a public blockchain implementation for DECENTER.

To run the tests, we used a DELL Latitude E5470 laptop equipped with a 4x Intel Corei5-6440HQ at 2.60GHz, 8GB RAM, 256GB SSD Disk, running a Linux Ubuntu Operating System (16.04.6 LTS). Regarding the software stacks and tools, we implemented the *CLI* using Python 3.6, used the web3 library as an interface for the Ethereum network, the flask library to provide a public REST API, and the gzip utility for file compression (to reduce the size of smart contracts). To evaluate the time response of executing a smart contract we used Ganache (version 2.0.1), that is part of the Truffle suite 2, for a local implementation.

This software system emulates a real Ethereum blockchain interaction in terms of interfaces etc. so it provides the same results as sending a transaction to a real Ethereum node, but without the network delay and the transaction processing time needed for the mining process.

For this part instead (i.e. assessing the network performance) we ran a node of the Ethereum Ropsten3 test network, using the official Ethereum client geth (version 1.9).

In the implemented marketplace, meant to replicate part of the DECENTER REB, we have two main actors, the seller and the buyer. The picture below illustrates the experiment in the wider DECENTER context introduced in D3.1. The seller is on the left and the buyer is pictured on the right of the REB red boundaries. In the experiment the seller produces an Advertising Smart Contract (ASC) and the seller issues a Reservation for that Smart Contract (RSC).

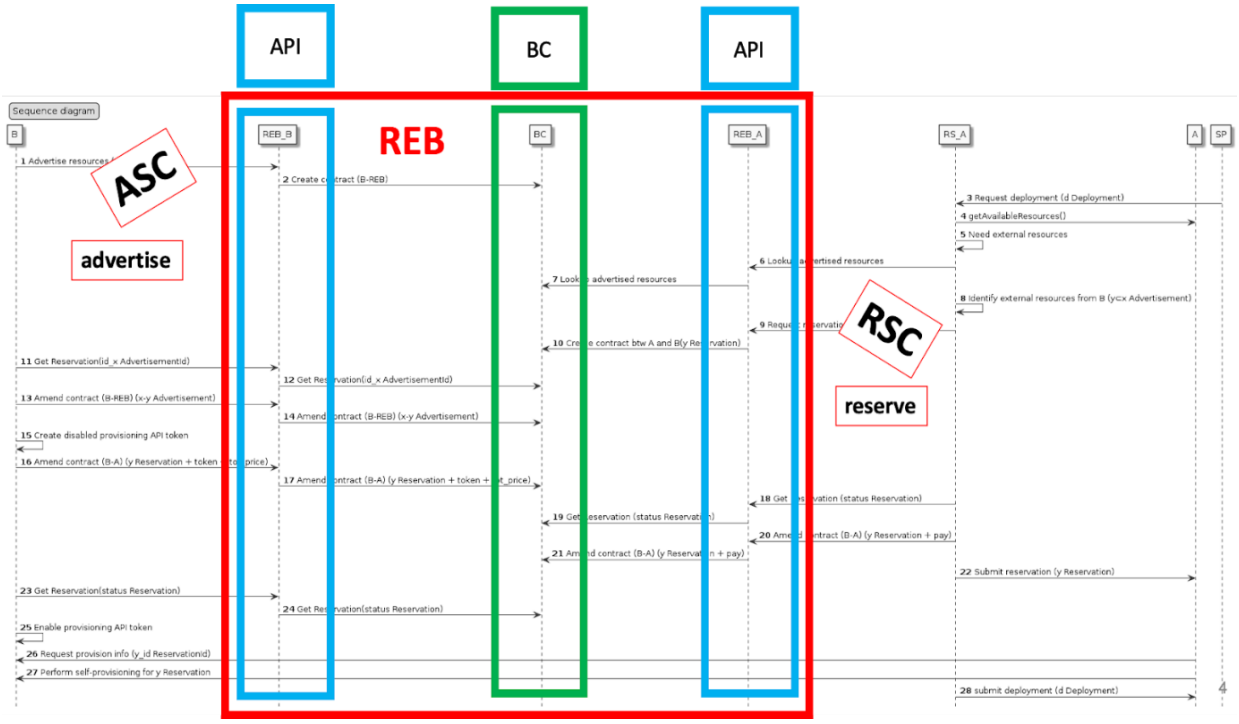

Figure 31: Resources exchange workflow

As already mentioned, the ASC can be characterised by a number of different fields, according to the complexity of the metadata used to describe the resources to be exchanged. Here the implementation choice can be made on how many fields are to be used in the Smart Contract: the more the fields the more expensive is to store the ASC on the public blockchain.

Considering this, one can decide to stringify some of the fields and delegate their parsing to the software outside the REB boundaries. This will certainly contribute to a cost reduction every time an ASC is to be uploaded.

Our reservation is based on the escrow contract provided by OpenZepellin, that is an industry-approved library for secure smart contract development. This base escrow also stores the reservation details raw bytes on a single variable. Since such data does not need to be processed internally by the RSC, it can be also stored in compressed format.

In order to obtain an objective evaluation, the test was run issuing ASC throughout a 24h period and using different field numbers. The tables below show the results of this first set of tests. The first one reports on the average costs of creating and updating ASCs, whereas the second shows how the cost is related to the complexity of the Smart Contract. As it was to be expected, the average costs are lower when the blockchain is less used (i.e. updating vs.

creating and handling smaller number of fields). Similar trends can be obtained compressing the contents of the Smart Contract reducing its memory footprint.

AVERAGE COST FOR CREATING AND UPDATING ADVERTISEMENTS ($T_p = 10\ gwei$ AND $C_e = 205\ USD$).

	Normal		Compressed	
a) Maximum	***Gas***	***USD***	***Gas***	***USD***
Create	4151378	8,5	4040785	8,3
Update	724653	1,5	488352	1,0
b) Medium	***Gas***	***USD***	***Gas***	***USD***
Create	3460243	7,1	2877232	5,9
Update	661437	1,4	270337	0,6
c) Minimum	***Gas***	***USD***	***Gas***	***USD***
Create	3352120	6,9	2467789	5,1
Update	587023	1,2	263409	0,5

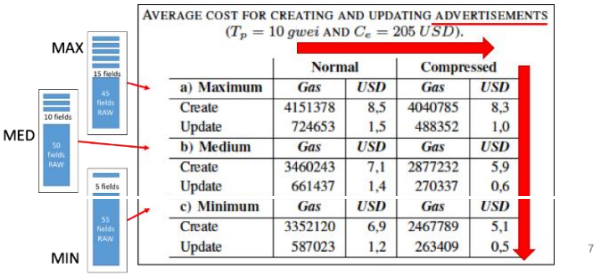

Figure 32: Costs

The next test was run to evaluate the overall costs of the marketplace, which depends on both the RSC and the ASC, also differentiating on where the registry of advertised resources would be stored. In fact, the marketplace also needs to maintain a registry R of all the existing advertisements based on the unique identifier (fuid) and the blockchain address (fadd) of each advertisement. Since this registry can be queried, we consider only two types of possible queries, namely a) All and b) Filtered. The *filtered query* is simply based on the region freg ∈ F, leading to three alternatives to implement this registry R = {fuid, fadd, freg} in the REB:

a) On-Chain Registry: the full registry R (fuid, fadd, freg) is stored as fields in the contract. This alternative allows to identify when a fuid is being repeated, and to parse the registry without off-chain processing.
b) Off-Chain Registry: advertisements are stored as events outside the blockchain; this is a cheaper alternative for registry storage purposes. In this case, the contract cannot identify if an fuid is being repeated, neither it can parse the registry. These checks must be done off-chain, since smart contracts cannot access event information.
c) Hybrid Registry: with this option we minimize the blockchain internal storage, using it for the sole purpose of identifying if an fuid is being repeated or not. We use events to store the rest of the registry and the parsing is done off-chain.

Figure 33 illustrates how the costs of creating the marketplace supporting the REB change according to size and various registry storage alternatives. As usual, the lower costs are obtained with a reduced memory usage within the blockchain.

AVERAGE TRANSACTION COST FOR CREATING THE MARKETPLACE ($T_p = 10\ gwei$ AND $C_e = 205\ USD$).

	d) On-Chain		e) Off-Chain		f) Hybrid	
	GAS	**USD**	**GAS**	**USD**	**GAS**	**USD**
a) Maximum	5830842	12,0	4939778	10,1	5070884	10,4
b) Medium	4797904	9,8	3888487	8,0	4019514	8,2
c) Minimum	3807595	7,8	2897972	5,9	3032280	6,2

Figure 33: Average costs for marketplace creation

The other important factor when evaluating a blockchain is the response time for all interactions (in the case of the REB we need to assess the time for writing Advertising Smart Contracts, Reservation Smart Contracts and, in case of using a blockchain also for the registry, the time it takes to retrieve all offers). Our experiments were also designed to assess

how would the REB perform under different conditions such as overall size of the envisaged marketplace (related to the number of advertisements and reservations). This would help in designing marketplaces at different regional level hierarchies to ensure the needed performances are achieved.

Figure 34 summarize the obtained results so far. The presence of a local buffer for the Registry as it can be seen, brings considerably down the average response time.

AVERAGE TIME RESPONSE (SECONDS) FOR QUERYING DETAILS OF ALL THE ADVERTISEMENT.

	a) Maximum		b) Medium		c) Minimum	
size(l)	*All*	*Filter*	*All*	*Filter*	*All*	*Filter*
128	15.4	3.8	15.9	3.3	14.4	3.1
256	29.2	7.4	28.7	5.9	28.3	5.8
512	59.2	15.1	57.5	11.8	57.3	11.8
1024	118.9	30.1	116.1	23.5	114.2	23.4
2048	257.8	-	231.9	47.5	229.8	46.5
4086	503.7	-	470.1	96.1	465.2	95.1
8192	980.1	-	975.7	198.9	966.9	196.5

AVERAGE TIME RESPONSE (IN SECONDS) FOR QUERYING THE DETAILED LIST OF ADVERTISEMENTS USING LOCAL BUFFER.

	a) Maximum		b) Medium		c) Minimum	
size(l)	*All*	*Filter*	*All*	*Filter*	*All*	*Filter*
128	0.007	0.004	0.006	0.005	0.005	0.005
256	0.014	0.007	0.012	0.008	0.011	0.007
512	0.024	0.011	0.025	0.011	0.025	0.011
1024	0.043	0.022	0.045	0.021	0.051	0.021
2048	0.091	0.036	0.089	0.032	0.096	0.041
4086	0.173	0.071	0.172	0.069	0.181	0.075
8192	0.379	0.133	0.383	0.128	0.404	0.135

Figure 34: Average response time dependencies

The results of this work have also been recently published in a paper to be presented at the International Workshop on Blockchain Applications and Theory (BAT 2020) in July 2020.

In terms of moving forward within the context of DECENTER project execution, the REB will be integrated with the SLA Manager, as already mentioned, given the results of the monitoring will need to be processed to ensure smart contracts can deliver the compensation to the right parties (i.e. the resource owner if all went well or returned back to the resource requestor if performance agreements were not met).

4 Initial evaluation of DECENTER scenarios

In this section, we show an initial evaluation of two different scenarios in the DECENTER platform. The development status of the components, which are involved in each scenario is currently at integration and testing phase.

In the Y3 project period our plan is to perform more detailed testing of the implemented approach and to further progress with the ongoing implementation that will result with components' implementation as part of the overall DECENTER architecture.

4.1 Redeployment scenario

For this scenario we show how, with the use of redeployment algorithms integrated into the DECENTER platform, we can improve the performance of applications. Also, since standard Kubernetes does not support the redeployment of applications, as its scheduler is designed for deployment of unscheduled Pods only, we have designed and implemented a specific Kubernetes controller in year 2.

4.1.1 Introduction to Kubernetes Controller

A Generic Description of Control Theory and Controllers

In control theory, an engineered process or a machine is a dynamic system that has to be continuously operated in a way such that its observed state approaches a desired state. This transition is achieved by performing a set of actions, which change the state of the system. In our case, a dynamic system is characterised by a (re)deployment process in which the observed state is represented by application performance, once deployed or its expected performance, before it is deployed, and the action to change the state is redeployment.

A controller usually runs in a control loop in which it continuously observes the state and acts with respect to its control algorithm aiming to match the observed state with a desired state.

Controllers in Kubernetes

Kubernetes adopted this behaviour from the control theory because of several advantages, for instance, to the Kubernetes control plane a controller behaves as a client and can be deployed either as a Pod or a set of Pods that run in the same cluster, or as an external processes. The advantage is that if a part of a control plane crashes or becomes unresponsive, it does not necessarily affect the controller itself. Another advantage is in the simplicity of implementing the controller, because an infinite loop can be implemented synchronously and by using the same APIs as a regular Kubernetes client would.

CRDs allow to introduce custom types into Kubernetes control plane. While the control plane can verify if an instance of a specific type conforms to its definition, it does not know how to handle such an object; it simply makes that object listable and readable to the clients. A custom controller can be implemented that observes when a particular object appears, disappears or changes in a Kubernetes control plane. Each of the object's life-cycle states as well as the content of the object itself can result in a different set of actions performed by a controller.

4.1.2 DECENTER Orchestration Algorithms

In deliverable D3.2 we have described two different orchestration algorithms. One expresses the deployment problem as a mixed-integer linear programming problem and focuses on finding the mapping between the regions and application's components with respect to a given requirements and constraints. The result of this algorithm is therefore an assignment of

regions to each of the application components. In the architectural figure presented in D2.2 this part of the deployment process refers to the scheduler component. It requires then to perform a set of additional steps to instruct one or more operated Kubernetes clusters, each associated to different region, to first map the regional application components to individual nodes and to finally perform the deployment according to the mapping. The realisation of multi-regional deployment might involve the configuration of networking, such that application components deployed in different regions can communicate with each other. The mapping to individual nodes and the actual deployment are handled by a standard Kubernetes. These additional steps performed after the algorithm assigns the regions is handled by the deployer component, as presented in architectural figure in D2.2. The scheduler has been implemented as a Kubernetes controller which inspects FADepl objects in Kubernetes control plane. The deployer has been implemented as a service, called by the scheduler to which it communicates the regional mapping of application components. Further details are omitted.

The other deployment algorithm described in deliverable D3.2 employs Markov Decision Problem (MDP) theory to map individual application components to a particular region, node or both and in addition implicitly remembers previous deployment decisions, modelled as a finite stochastic automaton. It can potentially operate as a part of a standalone subsystem, responsible for application deployment and maintenance of its target QoS. However, in year two we focused on its (re)deployment capabilities, implemented as a Kubernetes controller that can be integrated into DECENTER Fog and Brokerage platform and is in the following subsection described more into details.

The different capabilities of the presented orchestration algorithms and the flexibility of DECENTER Fog and Brokerage platform thanks to the adoption of extensible Kubernetes platform naturally and respectively led to two different designs of controllers. Both controllers only require a standard Kubernetes installation, which makes switching between the two algorithms possible, as desired by cluster operator. The following subsections detail the MDP-based controller.

4.1.3 Implementation

Our implementation comprises 5 components: (1) a Kubernetes controller, (2) a redeployment algorithm, based on Markov Decision Process theory (MDP), (3) the DECENTER monitoring system, (4) a message queue (MQTT), and (5) the DECENTER Kubernetes cluster within which the applications' redeployment occur.
Each component of the k8s Controller can be deployed either inside of the observed k8s cluster, thus extending the k8s control plane, or as an external service. The configuration of each component, however, depends on this decision.

4.1.4 Logical Flow of the Application Redeployment

The MQTT component publishes data on topics which are distributed to all subscribed peers. The k8s Controller uses this component by subscribing to the messages related to the redeployment and, upon the reception a message triggers the redeployment procedure. When a new request for redeployment arrives in a form of a message to the MQTT, the message is delivered to the k8s Controller service. The message contains a subset of the FADepl CRD type. Next, the k8s Controller sends a query to the k8s API Service to obtain recent information about nodes in the observed k8s cluster. This information is augmented with dynamic data reported by the Monitoring System. Combined data is shaped accordingly and sent to the MDP component in a form that the MDP understands. The MDP processes the redeployment request and makes an automated decision. The result of this service is node ranking with

respect to the quality of service they are estimated to provide. Finally, the k8s Controller consumes the result and in the case that the MDP suggests redeploying some application component to another node, the k8s Controller instructs the k8s API Service to perform the redeployment.

4.1.5 Controller's Implementation Details

This section provides more insight about each of the k8s controller's components, the content and the format of the messages exchanged, the APIs, and the technology choice.

Message Queue

The message queue is used in this scenario to control the redeployment flow of the platform, and detach from the DECENTER infrastructure, including starting/stopping a deployment/redeployment. We used *Eclipse Mosquitto*[4] message queue during the development and testing of the prototype, but any message queue service that supports the MQTT protocol could be used as well.

A deployment request is initiated by publishing a message on the topic *orchestration/new* into the MQTT. The message comprises the subset of the FADepl CRD type and is formatted as a JSON, as in the following example:

```
{
	"serviceName": "nginx",
	"exposedPort": 32000,
	"container": {
		"image ": "nginx",
		"port": 80,
		"environmentVariables": {
			"NGINX_HOST": "example.com",
			"NGINX_PORT": 80
		}
	}
}
```

Upon reception and processing of the request for the new deployment, the k8s Controller responds by publishing a simple message to the MQTT on the topic *orchestration/events*, confirming the success of the deployment action, e.g.:

```
{
	"serviceName": "nginx",
	"status": "deployed",
	"message": "Deployment was created.",
	"timestamp": "1592375274"
}
```

It is also possible to instruct the control loop to consider redeployment or to stop a service. This can be done by respectively publishing a message on the topic *orchestration/redeploy* or *orchestration/stop* of the form

```
{
	"serviceName": "nginx"
}
```

[4] https://mosquitto.org/

This description is by no means complete, but since the k8s controller is focused on redeployment actions rather than the initial deployment, the example provided here is very minimal. With little implementation effort, the k8s Controller can be extended to watch for FaDepl objects appearing on the k8s instead.

Querying k8s Control Plane about Cluster Nodes

Upon the reception of the message from MQTT, either for the redeployment action, the controller queries the k8s Control Plane to obtain the information about cluster's nodes. Among relevant information is node's name, the IP address, the number of CPU cores, the amount of memory installed, the amount of storage, the CPU architecture, and the underlying operating system.

More information on the querying format of the k8s Control Plane and multiple examples can be found on Annex A.

Monitoring Metrics

The controller relies upon the DECENTER Monitoring System, as it has been defined in Section 6.3, to extract relevant metrics from the infrastructure in order to obtain the insights about the expected behavior of the system. The considered metrics are of two kind: (1) host-related metrics and (2) network-link-related metrics. For all the nodes managed by the respective k8s control plane, the relevant host metrics are the CPU utilization, available memory and available storage, while the relevant network metrics are the bandwidth and the latency.

The network metrics are measured on the links between a video camera (usually its respective reverse-proxy/gateway) and each of the managed host nodes of a k8s cluster. For the development and testing, we relied upon *iperf3*[5] tool to estimate the bandwidth between two endpoints. We also extracted the bitrate directly from the incoming video stream. Unlike the bandwidth, the bitrate is always possible to estimate and its measurement is less intrusive, but depending on a video encoding used in the video stream, that metric might vary (e.g. between I-frames vs P-frames vs B-frames) and in case of adaptive streaming, such as MPEG-DASH or HTTP Live Streaming, the relative bitrate changes every time the video resolution of a video stream is adapted. Therefore, a better estimate, which is considered out of the scope here, would ask a video decoder for other metrics as well, as it has the information about lost/missing parts of video frames and so on.

MDP

The main decision-making component is implemented as a standalone service, which exposes the MDP-based algorithm over RESTful API. The algorithm has been described into detail in deliverable D3.2 and in the paper of Kochovski et al. [14] In this subsection, we focus on describing the RESTful API.

The redeployment algorithm is flexible with the choice of metrics and their respective threshold values. As an example, we considered the following set of metrics: cost, the number of CPU cores available, CPU utilization, memory utilization, available memory, video bitrate achieved over a network line between a host node and a video camera endpoint, network latency and

[5] https://iperf.fr

network throughput (earlier referred to as the bandwidth). Given the observed (i.e. measured) averaged values for each of the listed metrics and their respective threshold values, as defined in an SLA, the logic behind the MDP counts the number of metrics respecting the SLA.

The RESTful API defines a single resource that can be queried by sending an HTTP POST request at the resource /mdp. The body of the POST message is a JSON containing the list of candidate nodes, described by its name, UUID and the observed metric values, and followed by the metrics' thresholds. An example JSON containing a single node is as follows:


```
{
  "infrastructures": [
    {
      "cost": 0.023,
      "cpu_cores": 2,
      "cpu_util": 0.36,
      "location": "3F29+3H Ljubljana",
      "mem_util": 0.471,
      "memory": 2000000000,
      "name": "fog3",
      "networkBitrate": 8165537,
      "networkLatency": 9,
      "networkThroughput": 1104896372,
      "uuid": "e4b266f0-9dbc-4e4b-a33d-18c6b8b3b25c"
    }
  ],
  "threshold": {
    "cost": 0.1,
    "cpu_cores": 1,
    "cpu_util": 0.8,
    "mem_util": 0.7,
    "memory": 1000000000,
    "networkBitrate": 4000000,
    "networkLatency": 17,
    "networkThroughput": 8500000
  }
}
```


For each of the nodes in the infrastructures list the MDP algorithm calculates the reward and the utility scores. Sorting the nodes in the descending order with respect to the utility gives the ranking of candidate nodes. For the k8s Controller, a reasonable choice is to try to move the application service to the top ranked node first, although it is possible that the MDP algorithm valuates several nodes with the same utility score, in which case the k8s Controller can choose between them arbitrarily.

Once a node is selected, the MDP sends a JSON structured response, containing a list of the input nodes and their assigned scores, as shown in the example below:


```
[
  {
    "name": "fog3",
    "reward": 0.657834,
    "utility": 0.784322,
    "uuid": "e4b266f0-9dbc-4e4b-a33d-18c6b8b3b25c"
  }
]
```

4.1.6 Instructing k8s Control Plane to Perform Redeployment

Once the k8s Controller obtains the results from the MDP, the last step to take is instructing the k8s Control Plane to perform the deployment or redeployment (i.e. live migration). During an initial deployment, it just creates a new k8s object of type Deployment in which the node selector specification is used to force the mapping of Pods to the desired nodes. The k8s Scheduler in a control loop queries the k8s API Service to find all unscheduled Pods and assigns them to suitable nodes. If node selector is defined for a Pod, the assignment of a node is predetermined. Finally, k8s (Kubelet) then performs the actual deployment.

In the case of a redeployment, updating (or patching) an existing k8s Deployment object, is not a possibility, since some attributes of the specification are immutable (i.e. node/label selectors). Therefore, it is not possible to move Pods to another node by simply updating/patching the corresponding k8s Deployment object. We have investigated different possibilities to move Pods from one node to another:

1. **Recreate the deployment.** With this method, the Pods (and possibly their corresponding higher-level abstractions such as Deployments, StatefulSets, ReplicaSets, etc.) are deleted and recreated using the same context, but with different node selector labels. The advantage of this method is that, compared to other possibilities, is the most intuitive and straightforward way on how to move Pods from one node to another. One drawback is that the controller needs to ensure that the context is copied over to the new node. This is not limited to the configuration of the associated k8s objects but might involve snapshotting and copying of storage volumes from the old node to the new one. Another drawback is that the move might fail due to anti-affinity rules set between Pods, which means that a proper implementation of this strategy would need to redo a lot of work that other k8s services in the control plane already do. In the end, however, by careful observation of all the possible methods, this was the method we implemented.
2. **Using node/Pod affinity and anti-affinity rules.** This method is an upgrade of a simple node selector strategy as it is more expressive and flexible. Affinity rules can be specified for nodes and Pods, meaning that it is possible to restrict on which nodes a Pod can run and which Pods need to be co-located. Anti-affinity rules allow us to specify which Pods should not be co-located at the same node/region/zone. In addition, it is possible to distinguish the rules on hard and soft ones, which the k8s Scheduler respectively interprets as the rules that need to be respected at all cost versus the rules that are set as a preference in case there might be many feasible options to choose. Clearly, as the rules only apply at Pods' schedule time and not during the Pods' execution, this method has all the advantages of the node selector strategy, but it still has the same disadvantage in that it is not useful for moving Pods from one node to another. Fortunately, this limitation is subject to change in the future. In fact, some incubator branches of k8s already support applying the rules during Pods' execution time, which is the functionality we would require. What then remains to do is to evict a Pod from the old node which can be achieved by simply changing/deleting certain node labels. Similarly, to ensure that a Pod is moved to a

specific node, we should label the destination node accordingly and possibly remove certain labels from other candidate nodes.

3. **Using node taints.** Mainly for maintenance reasons, it is possible to cordon nodes (also known as performing node taints) the effect of which is to evict all the Pods from a node and mark them as unscheduled. The k8s Scheduler would then discover all unscheduled Pods and assign them to un-cordoned nodes, thus effectively avoiding the cordoned ones. By using this method, two problems appear: (1) how to specify exactly to which node should the k8s Scheduler assign a Pod, and (2) how to prevent the eviction of the other, possibly unrelated Pods. Problem (2) can be achieved by using Pod's tolerations, which is a set of key-value pairs that a Pod can tolerate. More precisely, when performing node taints, one needs to specify a set of labels as key-value pairs, resembling the reason for cordoning a node. If a Pod that is running on a cordoned node contains in its list of tolerations all the key-value pairs of the corresponding taint, it will tolerate the taint, meaning that it will not get evicted from the node. Let us now assume that an observed node contains n Pods, one of which we would like to move to another node while preventing the other n-1 Pods from getting evicted. We can see that such cherry-picking of Pods would require us to assign to each of the other n-1 Pods the same key-value pair as their tolerations, while not assigning it to the Pod we want to move. Fortunately, while it is impossible to change or delete a list of tolerations for a Pod during its execution, it is possible to add them. This last feature allows us to only add one (and the same) key-value pair to each of the preserved Pods, thus avoiding the potential combinatorial explosion of the key-value pair combinations we would have been required to assign otherwise. Problem (1) can be addressed by using affinity rules (which are immutable) and node labels (which are mutable).

4. **Descheduler**[6]. Due to the limitations of k8s Scheduler, which only assigns pending Pods to a node once and does not care if conditions have changed during Pod's execution, potentially leading to under- or over-provisioning of nodes, the Kubernetes community has developed a descheduler to counter these limitations. Its task is to find, based on its policy, a set of Pods that are scheduled on less desired nodes in a cluster and evict them. It can be run as a k8s Job or CronJob and runs as a critical Pod in the kube-system namespace, which makes it resistant to evictions. Currently, seven strategies are implemented that govern its behavior. For us particularly interesting is the rule RemovePodsViolatingNodeAffinity, which, as the name suggests, removes all Pods currently violating the node affinity rules. In other words, it resembles requiredDuringSchedulingRequiredDuringExecution affinity rule policy, which is currently missing in the standard k8s distribution, as mentioned above. Although this might be the best option at present to achieve the redeployment, as already written above, we implemented that step by recreating the deployment.

4.2 SLA Management scenario

Our approach in this subsection is evaluate the design of Smart Contract templates that contain specially designed Smart Contract functions that support container (re)deployment

[6] https://github.com/kubernetes-sigs/descheduler

operations according to a QoS model of the specific application. The idea is to support different algorithms that can be implemented to operate in connection to trustless Smart Oracle. In our work we rely on our existing implementation of a probabilistic Markov Decision Process, particularly do to the capability of the method to provide probabilistic assurances related to the chosen deployment configurations and based on application, infrastructure and other context related metrics that are served as input to the Markov Decision Process from the monitoring system. Our goal was to implement and test our novel approach on Ethereum ledger (testnet). The preliminary results show its feasibility for SLA management including low costs operation within dynamic and decentralised Edge-to-Cloud federations.

Generic use case

The motivation for this study is derived from the latest trends in software engineering, which are based on building flexible and reusable AI applications by implementing a multi-tier application architecture (as part of WP4 activities) with the use of microservices packed onto containers. However, splitting AI methods into several computing tiers, such as into Front and Rear parts of Deep Learning Neural Networks.

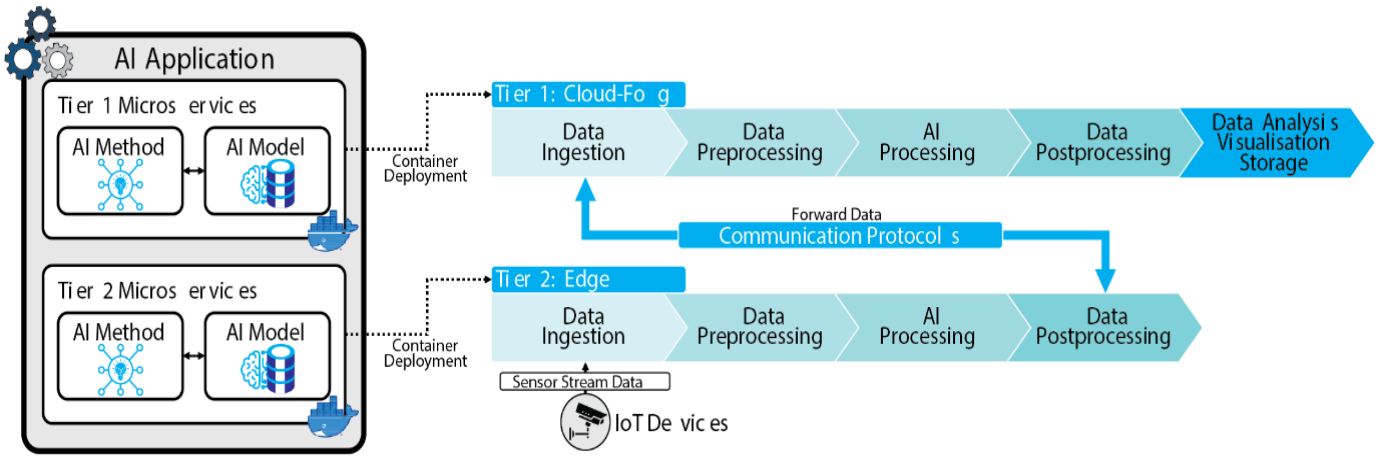

Figure 35: Two tiered AI based video analysis application

This generic two-tiered application design is depicted in Figure 35 and based on the DECENTER UCs. The application is composed of several containerised components, which are organised for deployment onto two different tiers. The first tier is composed of Cloud computing resources that allow executing software components requiring greater computing power. For instance, the first tier executes the Rear part of deep learning methods (e.g. TensorFlow) in order to analyse the incoming already preprocessed video frames. The second tier represents the Edge computing resources that execute the software component near the video surveillance data sources. Here the sensor data is pre-processed, and an AI method is executed in order to perform time-critical data analysis.

The decision upon a deployment option for this application is a complex problem. There are various internal and external metrics that could influence the resulting QoS of the application. For example, a definitive QoS metric for the application is its response time, that is, the time for a video frame to pass from the tier-two components up to the end of the pipeline. This resulting QoS can be influenced by network related conditions (e.g. attributes like latency,

throughput, packet loss and similar) and computing resource related operational conditions (e.g. CPU cores and available memory).

In our practical use case, the second tier is in Ljubljana, Slovenia, and is composed of five deployment options, whereas the first tier is composed of 40 deployment options that are spread across Europe, Asia, Australia and North America. Based on our current understanding, focusing on a two-tier AI application is an appropriate approach that can be further extended towards multitier application designs that may rely on different QoS models. For example, two-tiered AI applications can be used in the area of the smart construction sector in order to detect various safety violations, such as detecting, if workers wear safety equipment. The execution of the Front part of the Deep Neural Network close to the Edge may contribute to greater privacy, while the execution of the Rear part in a more powerful Cloud computing resource can be used to improve the QoS of the operation of such application.

In order to achieve high QoS in its operation, the smart AI application should be capable of immediate reaction in case of a high probability of an SLA violation. This means that the user may set this probability in a way that a redeployment of the application component in the first tier will happen whenever the probability of not withholding the required QoS threshold is too high. The information on the violations of SLAs is provided by the SLA Manager.

Blockchain based SLA Management

In the following, we elaborate on the use of the SLA management to provide high QoS operation to DECENTER's smart applications. The proposed architecture implements MDP methods that are used to automatically rank the available deployment options according to prior usage information, current monitoring data and QoS requirements that are precisely defined within the SLA.

In order to achieve these technical goals, the proposed approach allows registering available deployment options by providers, definition of SLA user requirements and autonomous deployment and redeployment of applications among the available deployment options. The approach can be explained through three usage scenarios, which are the following: (1) registering a certified deployment option which may be done by a Cloud provider, (2) automated deployment of containers and (3) automated redeployment of containers. These three scenarios are elaborated in the following subsections.

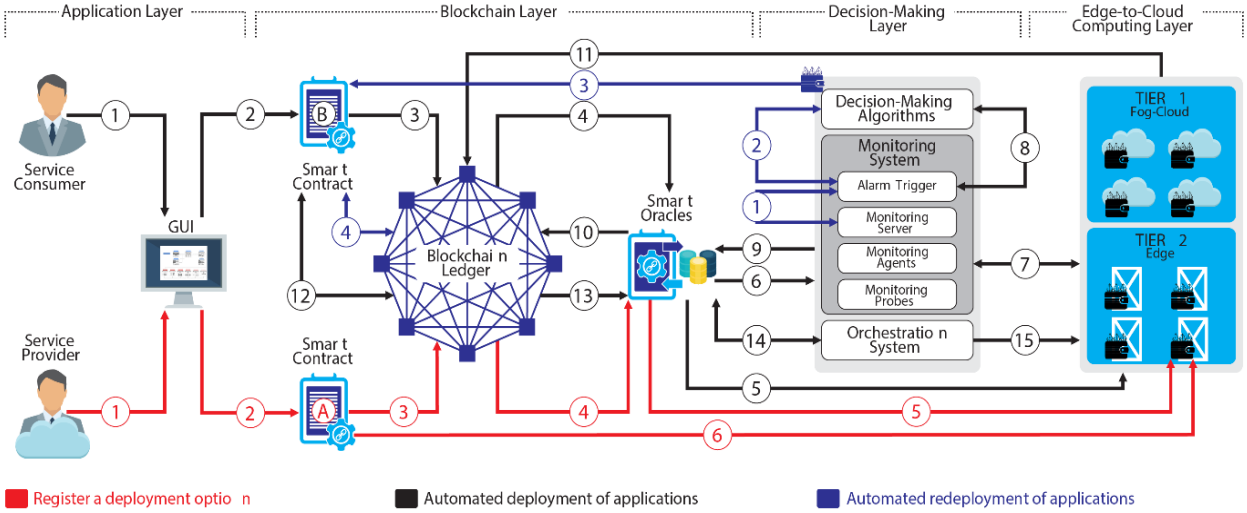

Figure 36: Architecture for SLA management and detailed design of loosely coupled system components

The key implemented components are the Smart Contracts used for the different actions, the MDP method that is implemented as a service, the specific monitoring system part that feeds data to the MDP method from the runtime environment, and the Smart Oracle that feeds specific data to the Smart Contract, in order to manage the SLAs.

Monitoring QoS metrics

In order to develop a proof-of-concept QoS model and provide autonomous SLA management, the proposed architecture uses infrastructure- and application-level metrics. Infrastructure-level metrics show the current status of the deployment options; thus, they are necessary to successfully perform the deployment process. The application-level metrics are accumulated once a deployment is finished; thus, they are used as additional metrics for performing redeployment operations. The used attributes for the (re)deployment processes are as follows:

1. Infrastructure-level attributes:

    - Network throughput (Mb/s) -- the rate at which data is transferred between two endpoints. At this level the endpoints are the source of data on one side and the deployment options on other side;

    - Network latency (ms) -- the time required for a packet to be transferred between the source of data and the deployment options;

    - Cost ($/month) -- cost for monthly use of a deployment option;

    - Amount of memory -- amount of memory that a deployment option offers;

    - Amount of CPU cores -- amount of CPU cores that a deployment option offers.

2. Application-level attributes:

    - Throughput (Mb/s) -- the rate at which data is transferred between the two tiers of deployment options;

    - Latency (ms) -- the time required for a packet to be transferred between the two tiers of deployment options.

The proposed approach is not limited to the specific set of QoS metrics and NFRs, because it supports any quantitative attribute that may be of interest to the software engineer and can be fed to the MDP method via a monitoring system. The Prometheus monitoring system apparently supports a plethora of such metrics that can be used.

Implementation

Service components shown in Figure 36 were implemented in order to analyse the ability to manage the SLAs. When developing a microservices based application the Cloud service consumer can define QoS requirements, which are incorporated into the SLA. This is communicated directly to Blockchain via the selected Smart Contract.

In order to communicate with the Ethereum ecosystem, we have implemented an ETH bridge called Metamask, which plays a pivotal role in the process. Primarily, Metamask is an Ethereum wallet that allows users to: (1) create and switch accounts that can be used in various ETH networks (e.g. Main ETH network, Ropsten, Kovan or Rinkeby); (2) perform transactions between accounts. It facilitates the interaction with the Ethereum ecosystem by injecting a Javascript library called web3.js.

Blockchain is used to automate the SLA management process and empower fairness between the involved parties (i.e. service providers and consumers) and executes traceable and transparent transactions on the Blockchain. The blockchain implementation is based on the public Ethereum (ETH) ledger as a public blockchain environment, and it is composed of two main components: SC templates and Smart Oracles. There are two main types of SC utilised in the system: SCs for registration of deployment options on blockchain and SCs for automated (re)deployment of applications.

The deployment of the SCs occurs on demand through the blockchain service which plays the role of a non-biased system that executes the SC and pays the service provider in case the SLA is not violated. However, in case there is an SLA violation, the SC terminates, pays the service provider for the service provided until the moment of the SLA violation, whilst compensating the service consumer for the remaining time.

SCs by default cannot act outside the blockchain, thus they are not capable of retrieving off-chain data. Since SCs in this SLA management system communicate with external services, such as: computing nodes, QoS monitoring system and the MDP decision-making mechanism, Smart Oracles were implemented. Smart Oracles are trusted third-party services that provide means for SCs to communicate with registered APIs from the external services. This approach results in enhanced integrity of the functions that verify the correctness of the API queries by using unique API keys and thus avoid calls from potential malicious SCs.

The MDP method implemented as a service estimates the optimal deployment option for deployment of containerised software components and initiates the container deployment process. In order to estimate an optimal deployment option, this layer queries a Smart Oracle from Blockchain to retrieve monitoring data and prior usage knowledge only for deployment options, which are registered on the blockchain.

The Markov Decision Process (MDP) is used to generate a probabilistic finite automaton that is built for each microservice. MDP utilises the automaton to derive utility value for each deployment option. These values are later used to produce a ranking list, where the first ranked deployment option is considered as an optimal deployment option and returned as an output result that satisfies the engineer's QoS requirements. The MDP automaton contains: a

set of states (i.e. deployment options); a set of actions (i.e. deployment actions applicable to the set of states); a set of transition probabilities, which represent the transition probability between states due to the available deployment actions; and a set of rewards, which represent the expected reward for transitioning from one state to another. Transition probabilities, which are estimated from prior usage experience of the deployment options and state rewards, which are estimated from the monitoring metrics are essential when calculating the utility of each state. A detailed description of the algorithm including the calculation of rewards and transition probabilities is available elsewhere in DECENTER publications.

The monitoring system constantly gathers QoS data from the deployment options that are registered on the blockchain. Each of those deployment options run Monitoring Agents and Monitoring Probes, which accumulate the QoS metrics and forward them to the Monitoring Server. In addition, the monitoring system contains an Alarm Trigger, which is a rule-based entity that continuously verifies the incoming monitoring data. If the Alarm Trigger experiences abnormal behaviour (i.e. SLA violation) it is responsible to initiate the redeployment process.

The orchestration is performed automatically with a special purpose controller built by UL. It uses the Resource Models (aligned with Custom Resource Descriptors) and runs along with Kubernetes. Once the decision-making mechanism delivers an optimal deployment solution and a SC is successfully executed, the orchestrator receives deployment instructions (i.e. YAML script) that provide information on the deployment infrastructure, applications for deployment, backup and replication policies. This approach is also aligned with the other controller FogAtlas built by FBK.

The Edge-to-Cloud computing continuum is in fact composed of heterogeneous deployment options, which are registered on the blockchain by the various Cloud service providers. In our case, the deployment options are used for deployment of the containerised two-tier applications, where each tier of deployment options play a different role. For instance, the Edge-based deployment options are responsible for running the software components that require less computing power, whereas the other components run on the Cloud/Fog-based deployment options. Hence, they have been grouped in two groups and represent two possible computing tiers.

Executing the application

The proposed system consists of two types of SC: (1) a SC for registering deployment options on the BC and (2) a SC for executing (re)deployment operations through the blockchain whilst following the SLA. The workflow of the proposed architecture is illustrated with three correlated scenarios. The detailed flow of interactions between the fundamental components in the presented system within the three scenarios is shown in *Figure 37*.

The first scenario is registration of certified deployment options. It allows infrastructure providers to register via blockchain their available deployment options in the pool of certified deployment options. In order to do so, the infrastructure provider (1-2) invokes a SC to execute methods for (3) registration of a deployment option on the BC. Because the deployment options are off-chain data, (4) BC communicates with them through a Smart Oracle and (5) assigns to the specific deployment option a public address (i.e. digital wallet). When the public address is assigned, the SC registers the new public address onto the BC. Finally, (6) the provider verifies the public address of the deployment option.

The second scenario is automated deployment of applications. It performs SLA assessment, preparation, negotiation, contracting and deployment of software components on the optimal deployment options. This scenario is executed in the following 15 consecutive steps: (1) the software engineer defines application QoS requirements and preferred usage-time for the deployment option; (2) the GUI through the triggerSC() method invokes the SC through the Metamask Ethereum bridge; (3) the SC initiates deployment process and (4) triggers the Smart Oracle through the selectDO() method; (5) the Smart Oracle gathers information on blockchain-certified deployment options and (6) triggers the decision-making mechanism; (7) the decision-making mechanism retrieves prior usage data and current monitoring metrics for the available deployment options from the Monitoring System, (8) estimates the optimal option and (9) returns the results to the Smart Oracle, which (10) triggers the SC to (11--13) verify the wallet address, executes the deployment process; (14) the Smart Oracle triggers the Kubernetes Orchestrator cluster to (15) deploy the two-tier application on the selected deployment options.

The third scenario extends the second scenario by adding the automated redeployment functionality. The monitoring system feeds to the MDP service; contains an Alarm Trigger that constantly examines the monitoring data for threshold violations (1). Once a violation is detected it triggers the decision-making mechanism to reassess the probability of achieving high QoS (2). In case the probability for QoS violation is high (i.e. above a certain threshold), the decision-making mechanism retrieves a new optimal deployment option. The decision-making mechanism then directly forwards the solution to the SC through its public address that is dedicated for the redeployment process (3-4). Once the SC is triggered, it initiated the compensation process through an initiateRefund() function that is represented within the alternative scenario (i.e. alt) from *Figure 37*. In particular, the SC responsible for the initial deployment estimates the amount of time the deployment option was utilised and pays the service provider, whilst the service consumer is reimbursed for the remaining time for which the deployment option remained unused through the SC event FundsReleaseEvent(). After the compensation is finished, the SC has the same workflow as for the deployment scenario, which is: price determination, reaching consensus, executing payment and deployment of the application.

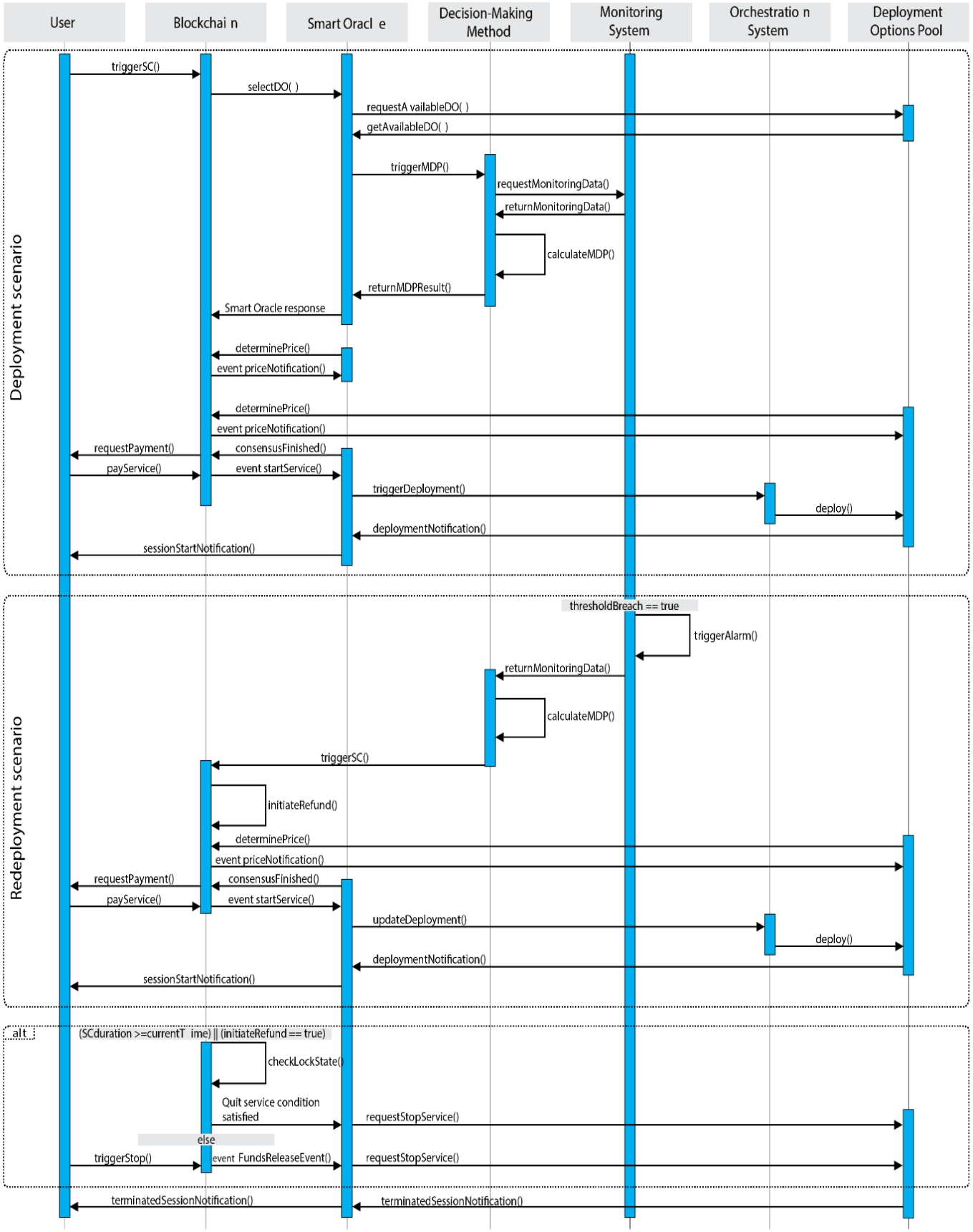

Figure 37: Sequence diagram of the deployment and redeployment scenarios

5 Conclusion

This report presents our update on the investigations and implementations performed during year 2 in WP3 during the second year of the DECENTER project. As a result of this work, we published the updated release of the DECENTER fog computing and brokerage platforms. In this report, we show our main indications on implementation and research work that were carried out in this year, and hint at the work to carry out in the last year of the project.

As with the work shown in D3.1, we have started our work from the architectural design shown in D2.2 and fed the works on the WP2 with the lessons learned from the implementation activities. Building on top of the work done in the first year, we have again relied on FogAltas and Kubernetes for the platform layer design and implementation, ensuring that the main functionalities relevant for DECENTER are provided. On this architecture, we have shown which are the main improvements taken during the year 2, as well as shown the new modules incorporated to it, such as the security and robustness components. Also, as result from the lessons learnt during the first year, we have improved our support for AI applications from the point of view of the platform, better integrating the tools and libraries for their support. As part of these works, we have shown how two well-known IoT platforms can be seamlessly integrated into the DECENTER platform, to improve the AI support.

From the infrastructure point of view, in the first year of activities, we provided a first working version of the fog computing platform, which can deploy applications on single-domain hierarchical infrastructures, and in this year we have improved this release with SLA management, monitoring, redeployment tools, integration with the blockchain, and security and robustness tools. Additionally, we evolved our brokerage platform, by improving the integration with infrastructure providers, resource advertising and resource exchange.

Finally, this report also highlights the works done on security and robustness of the system, a task started by year two but which has already provided very encouraging results for year three.

6 Annex A: Updates to FogAtlas

6.1 CRDs Definition

The following snippets of code show the definition of the FogAtlas CRDs

Region

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: regions.fogatlas.fbk.eu
spec:
  group: fogatlas.fbk.eu
  version: v1alpha1
  names:
    kind: Region
    plural: regions
  scope: Namespaced
  validation:
   openAPIV3Schema:
     type: object
     properties:
        spec:
          type: object
          properties:
            id:
              type: string
              description: id of the region (uuid)
            name:
              type: string
              description: name of the region
            description:
              type: string
              description: description of the region
            location:
              type: string
              description: location of the region (GPS)
            tier:
              type: integer
              description: tier of the region (0 = cloud ... n = very edge)
            type:
              type: string
              description: type of region, (nodes|clusters|hostcluster)
              enum:
                - nodes
                - clusters
                - hostcluster
```

Link

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: links.fogatlas.fbk.eu
spec:
  group: fogatlas.fbk.eu
  version: v1alpha1
  names:
    kind: Link
    plural: links
  scope: Namespaced
  validation:
```

```
    openAPIV3Schema:
      type: object
      properties:
          spec:
            type: object
            properties:
              id:
                type: string
                description: id of the link
              endpointa:
                type: string
                description: id of the region A connected through the link
              endpointb:
                type: string
                description: id of the region B connected through the link
              bwpeak:
                type: integer
                description: maximum bandwidth (bps) of the link
              bwmeasured:
                type: integer
                description: maximum bandwidth (bps) currently measured on the link
              latency:
                type: integer
                description: latency (ms) of the link
              status:
                type: string
                description: status (up/down) of the link
```

ExternalEndpoint

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: externalendpoints.fogatlas.fbk.eu
spec:
  group: fogatlas.fbk.eu
  version: v1alpha1
  names:
    kind: ExternalEndpoint
    plural: externalendpoints
  scope: Namespaced
  validation:
   openAPIV3Schema:
     type: object
     properties:
         spec:
           type: object
           properties:
             id:
               type: string
               description: id of the external endpoint (uuid)
             type:
               type: string
               description: type of the external endpoint (camera, sensor, ...)
             name:
               type: string
               description: name of the external endpoint
             description:
               type: string
               description: description of the external endpoint
             location:
               type: string
               description: location of the external endpoint (GPS)
             ipaddress:
               type: string
```

```
              description: IP address of the external endpoint
            regionid:
              type: string
              description: region Id (uuid) of the external endpoint
```

FADepl (FogAtlas Deployment)

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: fadepls.fogatlas.fbk.eu
spec:
  group: fogatlas.fbk.eu
  version: v1alpha1
  names:
    kind: FADepl
    plural: fadepls
  scope: Namespaced
  subresources:
    status: {}
  validation:
   openAPIV3Schema:
     type: object
     properties:
       apiVersion:
         type: string
       kind:
         type: string
       metadata:
         type: object
       spec:
         type: object
         properties:
           algorithm:
             type: string
             description: name of the scheduling algorithm to use
           externalendpoints:
             type: array
             items:
               type: string
               description: array of ids (uuid) of the external endpoints this
application gets/sends data from/to
           microservices:
             type: array
             items:
               properties:
                 name:
                   type: string
                   description: name of a microservice composing the application
                 regions:
                   type: array
                   items:
                     properties:
                       regionrequired:
                         type: string
                         description: region id where the microservice should be
placed
                       replicas:
                         type: integer
                         description: override, for this region, the replica value
present in the deployment
                       image:
                         type: string
```

```
                        description: override, for this region, the image value
present in the deployment
                deployment:
                  type: object
                  description: k8s appsv1.Deployment for the microservice
            dataflows:
              type: array
              items:
                properties:
                  bandwidthrequired:
                    type: integer
                    description: required bw (bps) for the data flow between
sourceid and destinationid
                  latency:
                    type: integer
                    description: required latency (ms) for the data flow between
sourceid and destinationid
                  sourceid:
                    type: string
                    description: microservice name source of this data flow
                  destinationid:
                    type: string
                    description: microservice name destination of this data flow
```

7 Annex B: Implementation of robustness measures in Kubernetes

Again this has been done using the CRD feature of Kubernetes: we defined a federated type corresponding to the FADepl type defined in. Such a type is called FedFADepl and is defined as follows.

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: fedfadepls.fogatlas.fbk.eu
spec:
  group: fogatlas.fbk.eu
  version: v1alpha1
  names:
    kind: FedFADepl
    plural: fedfadepls
  scope: Namespaced
  validation:
   openAPIV3Schema:
     type: object
     properties:
        spec:
          type: object
          properties:
            name:
              type: string
              description: name of the fedfadepl (application)
            description:
              type: string
              description: description of the fedfadepl (application)
            algorithm:
              type: string
              description: name of the scheduling algorithm to use
            externalendpoints:
              type: array
              items:
                type: string
                description: array of ids (uuid) of the external endpoints this
application gets/sends data from/to
            microservices:
              type: array
              properties:
                name:
                  type: string
                  description: name of one the microservice composing the
application
                federateddeployment:
                  type: object
                  description: k8s types.kubefed.io/v1beta1/FederatedDeployment for
this microservice
            dataflows:
              type: array
              properties:
                bandwidthrequired:
                  type: integer
                  description: required bw (bps) for the data flow between sourceid
and destinationid
                bandwidthassigned:
                  type: integer
```

```
                  description: bw assigned (bps) for the data flow between sourceid
and destinationid
                latencyrequired:
                  type: integer
                  description: required latency (ms) for the data flow between
sourceid and destinationid
                sourceid:
                  type: string
                  description: microservice name source of this data flow
                destinationid:
                  type: string
                  description: microservice name destination of this data flow
```

8 References

[1] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard,Wayne E Hubbard, and Lawrence D Jackel. 1990. Handwritten digit recognition with a back-propagation network. In Advances in neural information processing systems. 396–404.

[2] “BPF and XDP Reference Guide,” https://cilium.readthedocs.io/en/latest/bpf/, 2018.

[3] M. Fleming, “A thorough introduction to eBPF,” https://lwn.net/Articles/740157/, 2017.

[4] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel,” in Proc. of of the 14th International Conference on Emerging Networking EXperiments and Technologies, 2018.

[5] “eBPF maps”, https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html, 2019.

[6] “Packet interface on device level”, http://man7.org/linux/man-pages/man7/packet.7.html, 2020.

[7] “UNB Datasets”, https://www.unb.ca/cic/datasets/index.html, 2020.

[8] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martìnez-del-Rincón, and D. Siracusa, “LUCID: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection,” IEEE Transactions on Network and Service Management, 2020.

[9] “bpf() system call”, http://man7.org/linux/man-pages/man2/bpf.2.html, 2019.

[10] DECENTER D2.1. “Architecture and use cases”

[11] Granadillo, Gustavo & Diaz, Rodrigo & Medeiros, Ibéria & Gonzalez-Zarzosa, Susana & Machnicki, Dawid. (2019). LADS: A Live Anomaly Detection System based on Machine Learning Methods. 10.5220/0007948904640469.

[12] Introduction to Cisco IOS NetFlow - A Technical Overview. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html

[13] Ring, M., Wunderlich, S., Gruedl, D., Landes, D., Hotho, A.: "Creation of Flow-Based Data Sets for Intrusion Detection”. In: Journal of Information Warfare (JIW), Vol. 16, Issue 4, pp. 40-53, 2017

[14] Kochovski, P., Drobintsev, P., & Stankovski, V. (2019). Formal Quality of Service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method. Inf. Softw. Technol., 109, 14-25.

[15] DECENTER D3.1. “First release of the fog computing platform”

[16] DECENTER D2.2. “Final release of the DECENTER architecture specification and use cases”

9 Glossary & Abbreviations

WP	Work Package
SLA	Service Level Agreement
SLO	Service Level Objective
SMI	Service Measurement Index
ISO	International Organization for Standardization
IT	Information Technology
QoE	Quality of Experience
QoS	Quality of Service
K8S	Kubernetes
ML	Machine Learning
AI	Artificial Intelligence
SaaS	Software-as-a-service
PaaS	Platform-as-a-service
IaaS	Infrastructure-as-a-service
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
IoT	Internet of Things
AWS	Amazon Web Services
VM	Virtual Machine
GUI	Graphical User Interface
CLI	Command Line Interface
API	Application Programming Interface
REST	Representational State Transfer
REB	Resource Exchange Broker
DB	Database
VPN	Virtual Private Network
UML	Unified Modelling Language
DAG	Directed Acyclic Graph
NGI	Next Generation Internet

BC	Blockchain
DLT	Distributed Ledger Technology
MTBF	Mean Time Between Failures
MTTR	Mean Time To Repair
MQTT	Message Queuing Telemetry Transport
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
OSGI	Open Service Gateway Initiative
UC	Use Case
MDP	Markov Decision Process
CRDs	Custom Resource Definition