PROJECT "LOCUS": LOCalization and analytics on-demand embedded in the 5G ecosystem, for Ubiquitous vertical applicationS

Grant Agreement Number: 871249
(https://www.locus-project.eu/)

DELIVERABLE D4.4

"Implementation of the virtualization platform for network control and management: final version"

Deliverable Type:	R
Dissemination Level:	Public
Contractual Date of Delivery to the EU:	30/04/2022
Actual Date of Delivery to the EU:	29/04/2022
WP contributing to the Deliverable:	WP4 – Localisation & Analytics for Smart Network Management
Editor(s):	Giacomo Bernini, Nicola Venturi (NXW)
Author(s):	Giacomo Bernini, Nicola Venturi, Elian Kraja, Michael De Angelis (NXW) Athina Ropodi, Aristotelis Margaris, Kostas Tsagkaris (Incelligent) Domenico Garlisi, Flavio Morselli, Gianluca Torsoli, Andrea Conti (CNIT)

	Maria Belesioti (OTE) Emil Khatib, Carlos Alvarez Merino (UMA)
Internal Reviewer(s):	Takai Eddine Kennouche (VIA) Athina Ropodi (INCE)
Short Abstract:	The goal of this deliverable is to describe the final version of the LOCUS Virtualization Platform that enables the deployment and execution of the LOCUS functions as VNFs in distributed computing locations. The document reports on the implementation and deployment of the Virtualization Platform in the OTE testbed, which models an edge/core 5G network infrastructure. The final software prototype of the LOCUS MANO, responsible for the lifecycle management of the LOCUS analytics services on top of the Virtualization Platform, is described. A description of its deployment and validation in the OTE testbed is also provided, together with the integrated LOCUS PoC functions and services.
Keyword List:	5G, localization analytics services, MANO, virtualization, NFV, edge, Openstack, Kubernetes

Executive Summary

LOCUS provides a unified and generalized localization analytics platform for the deployment of localization analytics functions and services as virtualized assets on top of 5G infrastructures. These functions and services are then exposed towards Smart Network Management and 3rd party vertical applications that require (geo)location-awareness and analytics for their purposes. With this kind of virtualization-based approach, LOCUS can enable the reuse of common localization and analytics functions for several purposes and applications scopes. This deliverable provides details on the implementation, deployment and early validation of two key technical aspects of the LOCUS Platform. First, it describes how the LOCUS Virtualization Platform has been implemented and deployed in the project testbed running at the OTE premises. The LOCUS Virtualization Platform follows a hybrid approach, and it allows the execution of LOCUS functions and services in both containerized and virtual machine-based environments. It is oriented to a cloud native approach, in line with the high-degree of virtualization of network functions and services envisioned in the 5G system architecture. The implemented LOCUS Virtualization Platform integrates de-facto standard opensource technologies, such as Openstack and Kubernetes, to support a realistic distributed edge/core computing infrastructure where LOCUS functions and services can be deployed on-demand. On the other hand, this deliverable describes the prototype of the LOCUS Management and Orchestration (MANO), and related deployment in the project testbed along with its integration with the LOCUS Virtualization Platform. The developed LOCUS MANO framework also relies on opensource technologies, such as ETSI OSM (the de-facto standard opensource Network Function Virtualization – NFV - orchestration platform) and provides NFV-oriented automation capabilities in the deployment and runtime operation of localization analytics functions and services as Virtual Network Functions (VNFs) and NFV Network Services. Leveraging on the capabilities of the Virtualization Platform, the implemented LOCUS MANO supports cloud-native deployments and thus automated instantiation and configuration of localization analytics functions and services in Kubernetes environments. Beyond these two main implementation and deployment activities, this deliverable also reports on the actual development and packaging of LOCUS functions as ETSI OSM ready VNFs (mostly containerized ones) and their integration and automated deployment in the project testbed.

The implementation and deployment work described in this document is based on the architecture design and functional decomposition for both the LOCUS Virtualization Platform and the MANO provided as part of deliverable D4.3. Moreover, the LOCUS platform architecture principles and design approach specified in deliverable D2.5 also provide a reference baseline, especially for what concerns the aspects of software containerization, microservice architecture and dynamic resource allocation.

VERSION CONTROL TABLE			
VERSION N.	**PURPOSE/CHANGES**	**AUTHOR (S)**	**DATE**
0.1	First draft of ToC	G. Bernini / E. Kraja	03/02/2022
0.2	CNIT Content on privacy service	D.Garlisi	18/02/2022
0.3	UMA Content on High precision UC	S.Fortes	22/02/2022
0.4	NXW content on packaging of functionalities	N.Venturi / E.Kraja	07/03/2022
0.5	INCE content for Service discovery	A. Ropodi, A. Margaris, K. Tsagkaris	15/03/2022
0.6	NXW content on Virtualization Platform implementation	E. Kraja	31/03/2022
0.7	NXW content on LOCUS MANO implementation	E. Kraja / N. Venturi	07/04/2022
0.8	NXW and CNIT content on integration of LOCUS functions and services	E. Kraja / N. Venturi / G. Bernini / M. De Angelis / F. Morselli	14/04/2022
0.9	Final version for internal review	G. Bernini	18/04/2022
0.9_review	Internal Review	A. Ropodi, T. Kennouche	22/04/2022
1.0	Final Version	G. Bernini	28/04/2022
1.1	Final revision by the coordinator	Nicola Blefari Melazzi	29/04/2022

INDEX

List of Abbreviations

ABBREVIATION	FULL NAME
3GPP	3rd Generation Partnership Project
5G	Fifth generation technology standard for cellular networks
AMQP	Advanced message queue protocol
API	Application Program Interface
CNF	Containerized Network Function
CRUD	Create, Read, Update, Delete
DNS	Domain name system
DHCP	Dynamic Host Configuration Protocol
ETSI	European Telecommunications Standards Institute
FTM	Fine Time Measurement
GUI	Graphical user interface
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
K8s	Kubernetes
KDU	Kubernetes deployment unit
KNF	Kubernetes Network Function
LCM	Lifecycle Manager
LSB	Location Based Service
LTE	Long-Term Evolution
MANO	Management and Orchestration
MQTT	Message queue telemetry transport
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NoSQL	Not Only Structured Query Language
NS	Network Service
NSD	Network Service Descriptor
OSM	Open-Source MANO
PoC	Proof of concept
RAT	Radio access technology

REST	REpresentational State Transfer
RSRP	Reference Signal Received Power
SI	Soft Information
SQL	Structured Query Language
STOMP	Streaming text-oriented message protocol
TCP	Transmission Control Protocol
UE	User Equipment
UWB	Ultra-Wideband
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFD	Virtual Network Function Descriptor
VPN	Virtual Private Network
WP	Work package
YAML	YAML Ain't Markup Language

Table 1: Abbreviation List

Table Index

Table of Figures

1 Introduction

1.1 Document scope and objectives

This deliverable represents the final outcome of task 4.3 (T4.3) “Virtualization platform for network control and management” activities. As part of the T4.3 objectives, the LOCUS Virtualization Platform and the Management and Orchestration (MANO) functions have been implemented, deployed and early validated in the project testbed available at OTE premises.

These two key components of the LOCUS Platform provide an advanced and innovative solution for automating the lifecycle management of any localization analytics service over 5G virtualized infrastructures spanning edge and core locations.

This deliverable reports the details of the LOCUS Virtualization Platform and of the implementation of a prototype of MANO functionalities, together with their deployment and integration in the project “production” environment represented by the OTE testbed. This work is built on the reference architecture designs described in D4.3 [1], where the functional decomposition and technical approach for the LOCUS Virtualization Platform and MANO were specified. Overall, the work described in this document covers the implementation of the LOCUS Management, Orchestration and Virtualization Infrastructure components, when referred to the LOCUS system architecture defined in D2.5 [10] and highlighted in red in Figure 1.

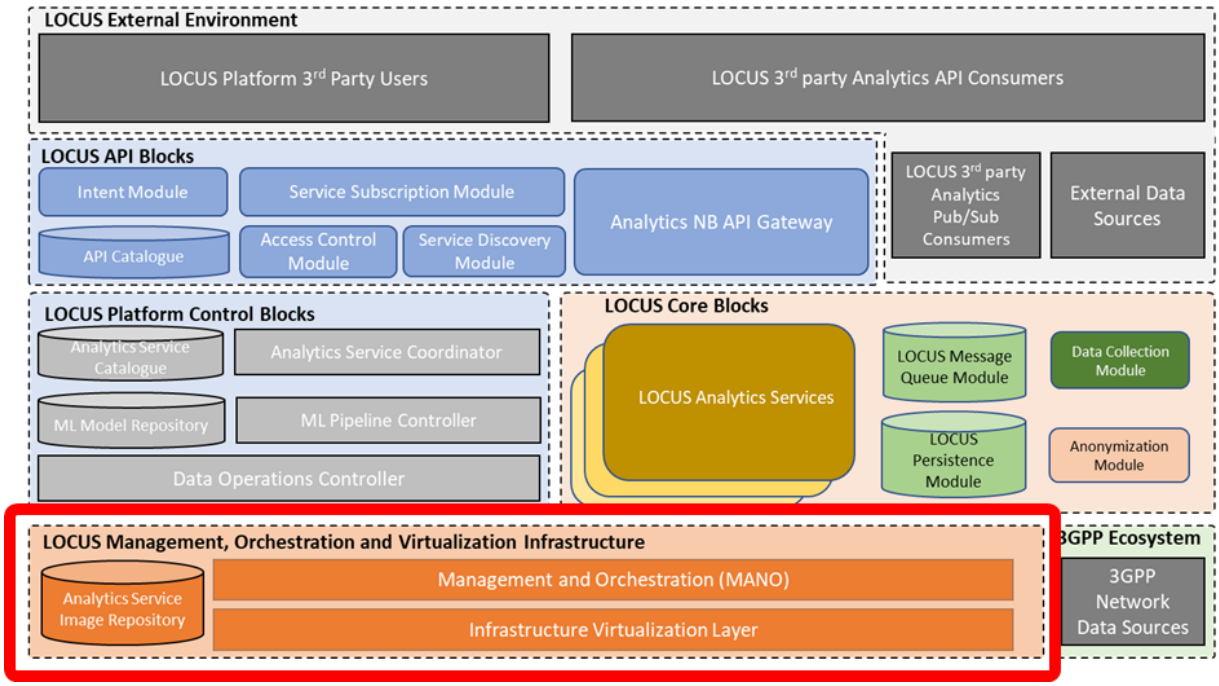

Figure 1: LOCUS System Architecture (ref. D2.5 [10])

The LOCUS Virtualization Platform described in this document implements the Infrastructure Virtualization Layer functionalities of Figure 1, and is implemented and deployed in the OTE testbed as hybrid managed cloud that models an edge/core 5G virtualized infrastructure. In particular, following the technical solution described in D4.3 [1], a Kubernetes [2] cluster is integrated with an Openstack [3] infrastructure enabling the deployment and execution of LOCUS localization analytics functions and services as cloud-native applications. On the other hand, the LOCUS MANO described in this deliverable implements the MANO functionalities of Figure 1, and thus allows to package, deploy and operate heterogeneous localization analytics functions as traditional or containerized Virtual Network Functions (VNFs), and interconnect them in the form of virtualized localization analytics services. The implemented LOCUS MANO relies on the opensource ETSI OSM platform [9] and introduces a high degree of automation, enabling the on-demand execution of the localization analytics services in response to the requirements expressed by the Smart Network Management or 3rd party vertical applications and use cases.

As this is the final outcome of task T4.3, it is worth mentioning that the activities related to the LOCUS Virtualization Platform and MANO, in terms of support to LOCUS functions and service providers will be continued in the context of the Proof-of-Concept (PoC) development, integration and demonstration activities in WP6, as well as in WP5. This support includes the packaging of functions as virtualized and containerized network functions compatible with ETSI OSM, and the realization of localization analytics services that can be automatically deployed, configured and operated on top of the cloud-native LOCUS Virtualization Platform.

1.2 Document structure

This deliverable is structured into four main sections:

- Section 2 describes the LOCUS Virtualization Platform prototype, providing details on the various components deployed and integrated in the project testbed at OTE. For each component, a description providing information on its functionalities, characteristics and configuration is provided;
- Section 3 describes the LOCUS MANO software prototype, with details on the various components that have been deployed and integrated in the project testbed at the OTE premises. The various techniques used and supported by the ETSI OSM [9] based LOCUS MANO for packaging and modelling localization analytics functions and services are also provided
- Section 4 provides details on the actual implementation of part of the LOCUS localization analytics functions and services (i.e., those ready to be virtualized until the writing of this document) as VNFs and NFV Network Services, and their integration

with the LOCUS MANO and the Virtualization Platform at OTE for the purpose of the project PoCs.

- Section 5 provides concluding remarks for this document.

2 LOCUS Virtualization Platform

The Virtualization Platform, as described in deliverable D4.3 [1], is one of the key enablers of the LOCUS system, providing a common environment where the LOCUS functions can run as virtual functions, following the current trends of cloud-native applications distributed across edge and core domains. It is capable of deploying on-demand LOCUS functions, over a unified virtualized infrastructure, providing an abstraction of edge and core clouds. This allows the deployment and execution of the functionalities, based on NFV principles, as container-based applications or virtual machine applications.

2.1 Final Platform Design

The final design of the platform follows the initial approach described in deliverable D4.3 [1], relying on multiple virtualization technologies, implementing the above-mentioned edge and core domain distribution of the cloud-native applications. The LOCUS Platform implements the edge domain through the usage of a Kubernetes [2] cluster, described in detail in the following section 2.2.1, whereas the core domain is implemented through the usage of an OpenStack [3] instance described in Deliverable D6.2 section 2.2 [4]. Along with the virtualized infrastructure, the LOCUS Platform provides a set of platform components that enable the onboarding, execution, interaction of the different LOCUS functionalities, and the interaction of them with 3rd party software and external services.

2.2 Prototype Deployment in OTE Testbed

The LOCUS Virtualization Platform is hosted in the OTE infrastructure. It runs in a virtualized environment, built on top of an OpenStack instance. As described in deliverable D6.2, the dedicated testbed for the LOCUS Virtualization Platform consists of four different physical servers, the characteristics of which are described in Table 2.

Table 2: Resources for LOCUS Platform

Server	IP Address	CPU Qty	RAM Qty	HDD Qty
Locuscontroller	172.16.0.44	16 CPU	48 GB	5TB
Locuscompute1	172.16.0.43	88 CPU	256 GB	2 TB
Locuscompute2	172.16.0.45	88 CPU	256 GB	512 GB
Locuscompute3	172.16.0.46	88 CPU	256 GB	5.5 TB

The OpenStack installation, based on Queens release [5] consists of 3 compute nodes, and a controller node. The LOCUS Virtualization Platform is implemented within this OpenStack,

relying on different OpenStack projects, allowing to separate the three main areas of the LOCUS Virtualization platform, based on the architecture of the virtualized platform, depicted in Figure 2.

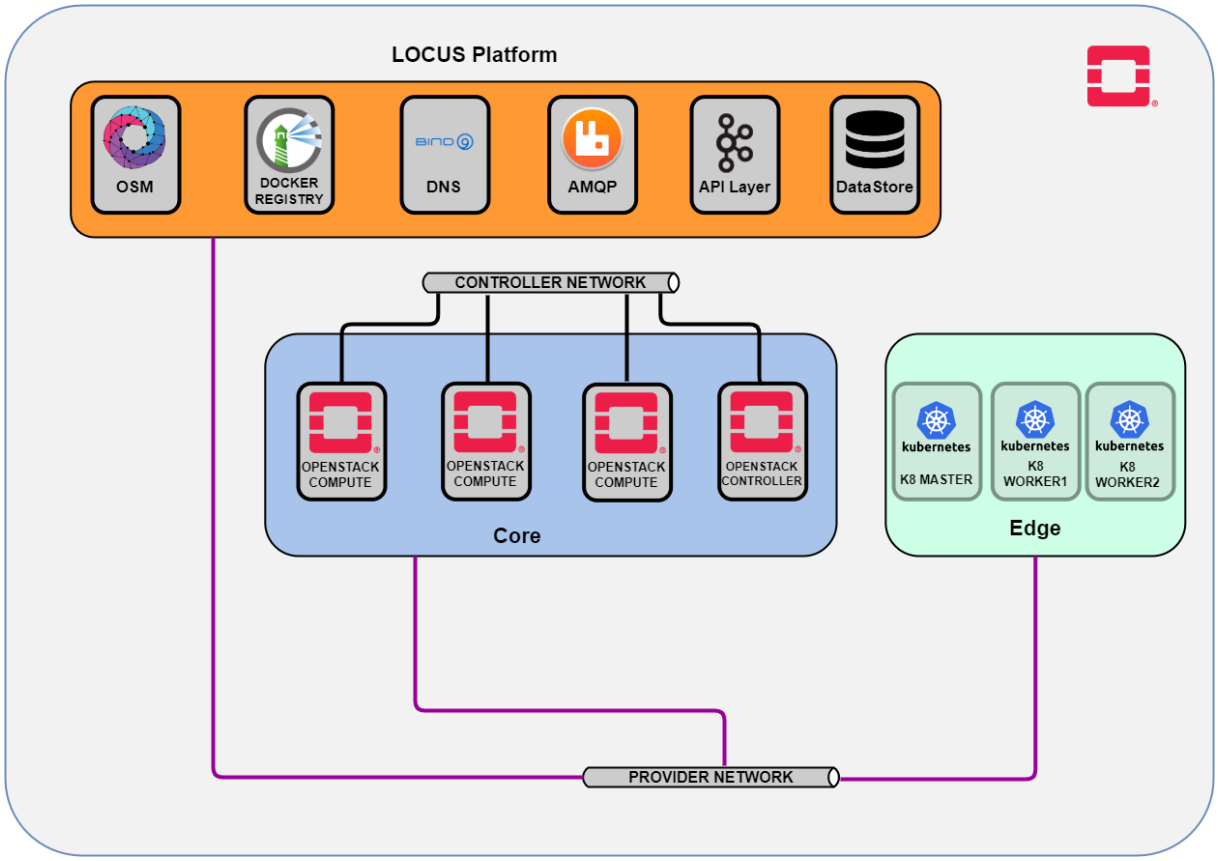

Figure 2: LOCUS Virtualization Platform in the OTE testbed

This is implemented through the creation of three different OpenStack projects: i) NFVI: space dedicated to the LOCUS functions, instantiated and orchestrated through the LOCUS MANO. This part reflects the core domain of the Virtualization Platform; ii) EDGE_CLOUD: this space is dedicated to the Kubernetes cluster. The resources allocated to this part of the infrastructure are shown in Table 3; iii) LOCUS-PLATFORM: this tenant is dedicated to the LOCUS platform components. The list of the components is shown in Table 4 and described in the following sections.

Table 3: Kubernetes cluster resources

Server	IP address	CPU Qty	RAM Qty	HDD Qty
Kube-master	k8s.locus-project.eu	4 CPU	8 GB	200 GB
Kube-worker1	172.16.12.51	20 CPU	40 GB	300 GB
Kube-worker2	172.16.12.32	20 CPU	40 GB	300 GB

The OTE testbed allows the usage of two different networks for the LOCUS Virtualization Platform. For the management purposes, all the OpenStack nodes are connected to the Controller network, as shown in Figure 2, whereas the LOCUS platform components and the Kubernetes cluster is connected to the Provider Network. This network is used by the LOCUS partners to access the LOCUS Platform and the services running into it, over the VPN connection provided by OTE.

Table 4: LOCUS platform components' resources

Component	IP address	CPU Qty	RAM Qty	HDD Qty
OSM-R10	osm.locus-project.eu	4 CPU	8 GB	80 GB
Registry	registry.locus-project.eu	4 CPU	8 GB	160 GB
DNS	helm-chart.locus-project.eu	2 CPU	2 GB	20 GB
Data movement	amqp.locus-project.eu	12 CPU	16 GB	120 GB
API Layer	api.locus-project.eu	16 CPU	32 GB	300 GB
Datastore	datastore.locus-project.eu	2 CPU	4 GB	120 GB

2.2.1 Kubernetes

The Kubernetes (k8s) cluster is part of the edge domain, able to run the LOCUS functionalities implemented as Containerized Network Functions (CNFs), and thus packaged as docker containers [6]. It consists in three nodes, a master node and two worker nodes, as described in Table 3. The LOCUS functions running into this domain are orchestrated over the LOCUS MANO and made available to other users or functionalities within the LOCUS Platform.

The K8s cluster is enriched with Rancher [7], a multi-cluster manager for Kubernetes clusters. Over a single web-interface it allows the users to manage different cluster resources. With respect to the LOCUS Virtualization Platform, it allows the Platform users to interact with their containerized functions for health-check or debugging purposes. The web-interface of Rancher is reachable on the same IP address of the k8s master node at:

https://k8s.locus-project.eu/

and allows to manage the LOCUS cluster, as depicted in the following Figure 3, showing the number of nodes that compose the cluster, and the resources dedicated, in terms of CPUs and memory.

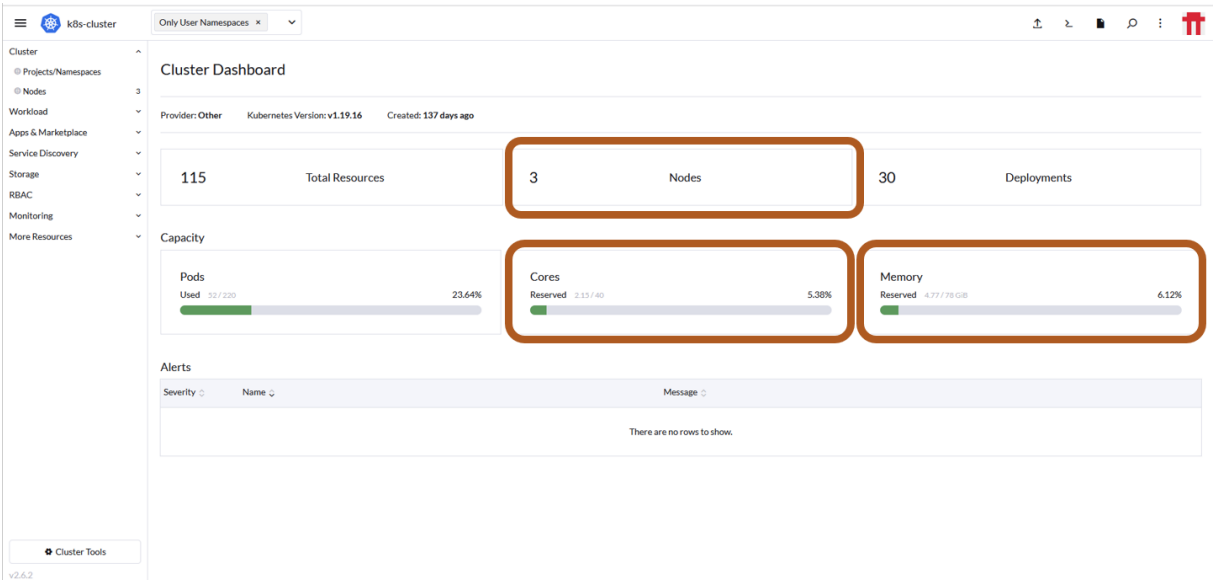

Figure 3: Rancher's web interface

Through the web-interface it is possible to check the services that are running in the cluster, and the service details to allow the LOCUS users to reach these, in case of exposed services.

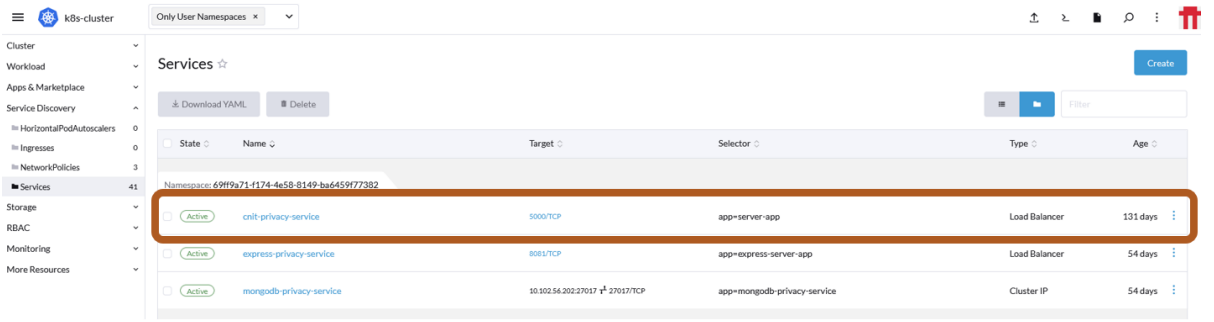

Figure 4: K8s service

2.2.2 Harbor Registry

The Harbor registry [8] is an open-source registry that secures artifacts with policies and role-based access control. It ensures that all the uploaded images are scanned and free from vulnerabilities, signing these as trusted images. Harbor registry is used to store the containerized images of the different LOCUS functions that are made available to the different PoCs when this is executed. In Figure 5, the list of container images available in the Harbor registry related to LOCUS PoC#1 is shown. The container images are retrieved automatically from the registry during the execution of the LOCUS services through the LOCUS MANO tool, ETSI OSM [9].

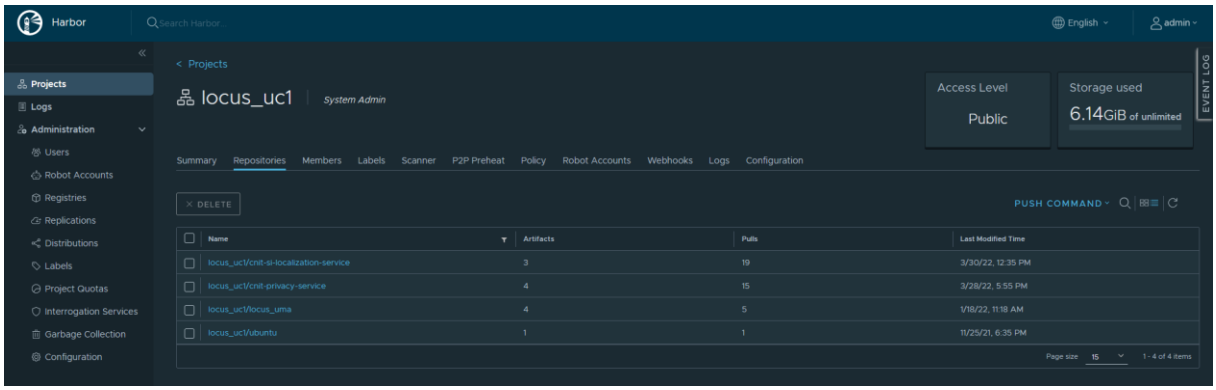

Figure 5: LOCUS functions related to PoC#1

The role-based access control feature is used in the installation of Harbor registry, to allow the distinction of the different LOCUS functions for each PoC. To enable this, the Harbor registry is configured with three different projects, as depicted in Figure 6. Each of these projects is configured with a public visibility. This allows any LOCUS user or LOCUS Platform component to interact with the Registry and retrieve the container images. Whilst the retrieval of the images is available to anyone within the LOCUS networking space, the creation of the images and the uploading to the Harbor registry is allowed only to authorized users.

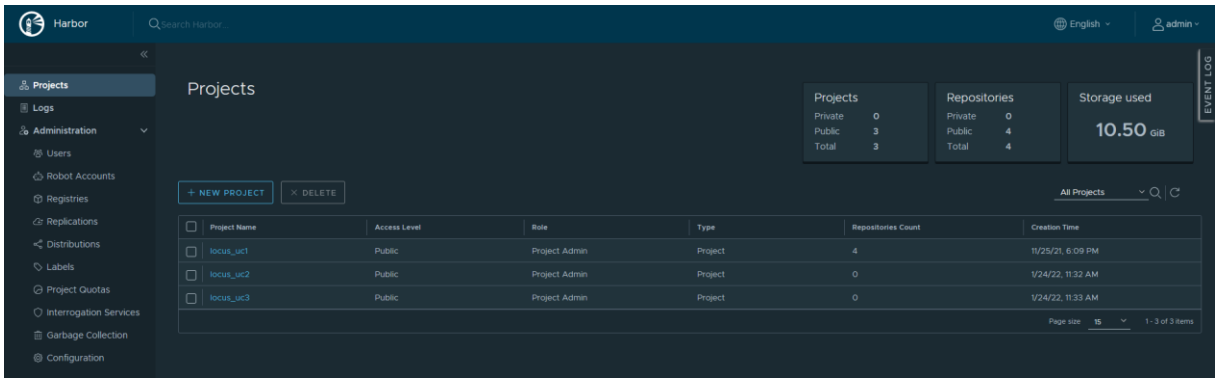

Figure 6: Harbor projects for the three PoCs

To enable this, three different users have been configured, one user for each PoC, together with maintenance and administration users, as depicted in Figure 7.

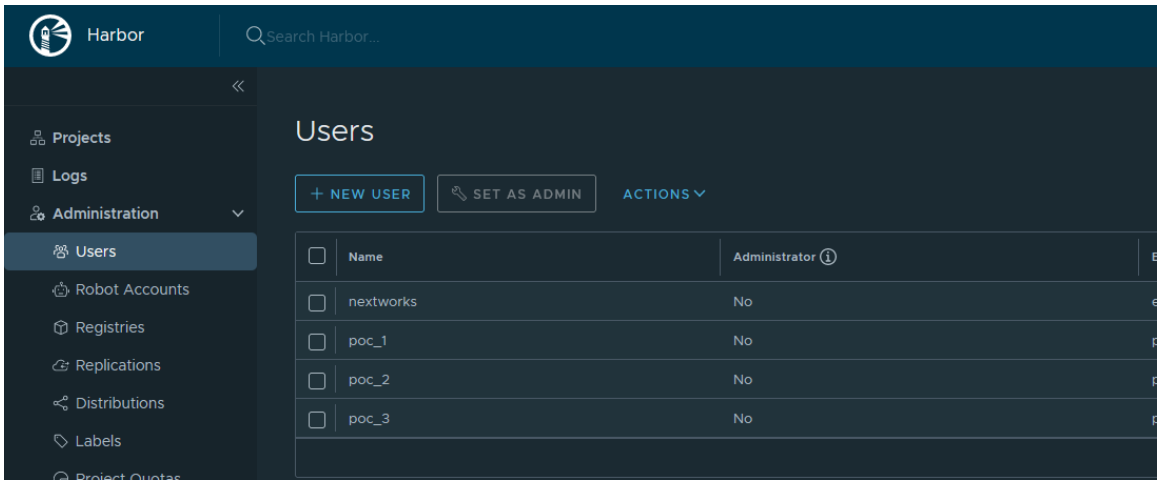

Figure 7: List of Registry users

Each PoC user is allowed to create and upload container images to the Harbor registry into the respective PoC project. Each PoC user has a maintainer role, that enables the user to perform Create Read Update Delete (CRUD) operations on the project.

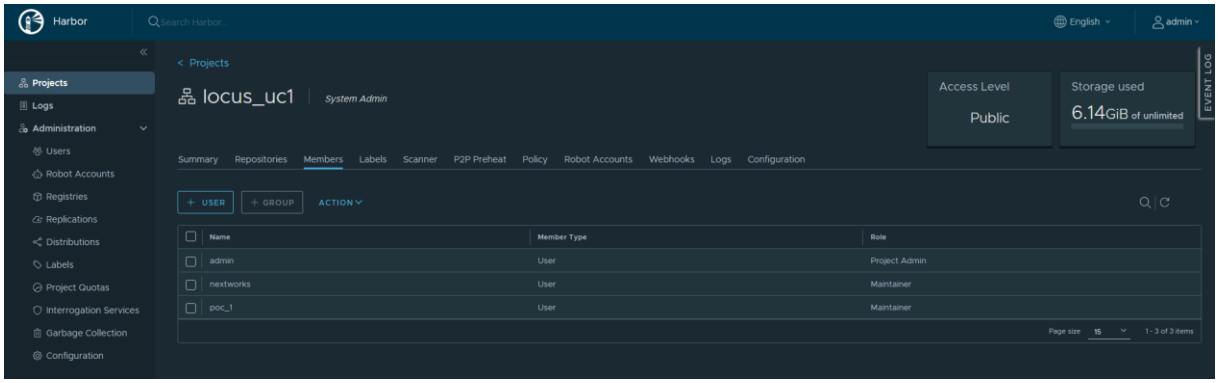

Figure 8: List of users enabled to PoC#1

2.2.3 Data movement

The LOCUS architecture, described in deliverable D2.5 [10], includes the availability of the LOCUS Message Queue Module (shown in Figure 1), that the LOCUS functions, the LOCUS platform components and third-party software can use for internal or intra-interactions. The solution implemented by the LOCUS Virtualization Platform is based on RabbitMQ [11], that supports the advanced message queue protocol (AMQP) [12], and - in the latest releases - it implements other protocols like MQ Telemetry Transport (MQTT) [13] and streaming text-oriented message protocol (STOMP) [14]. RabbitMQ provides a publish/subscribe pattern, which allows to decouple services by enabling asynchronous communications among then. RabbitMQ provides a broker component, whose main job is to filter and distribute all the produced messages to the proper subscribers. Figure 9 describes the MQTT publish subscribe architecture.

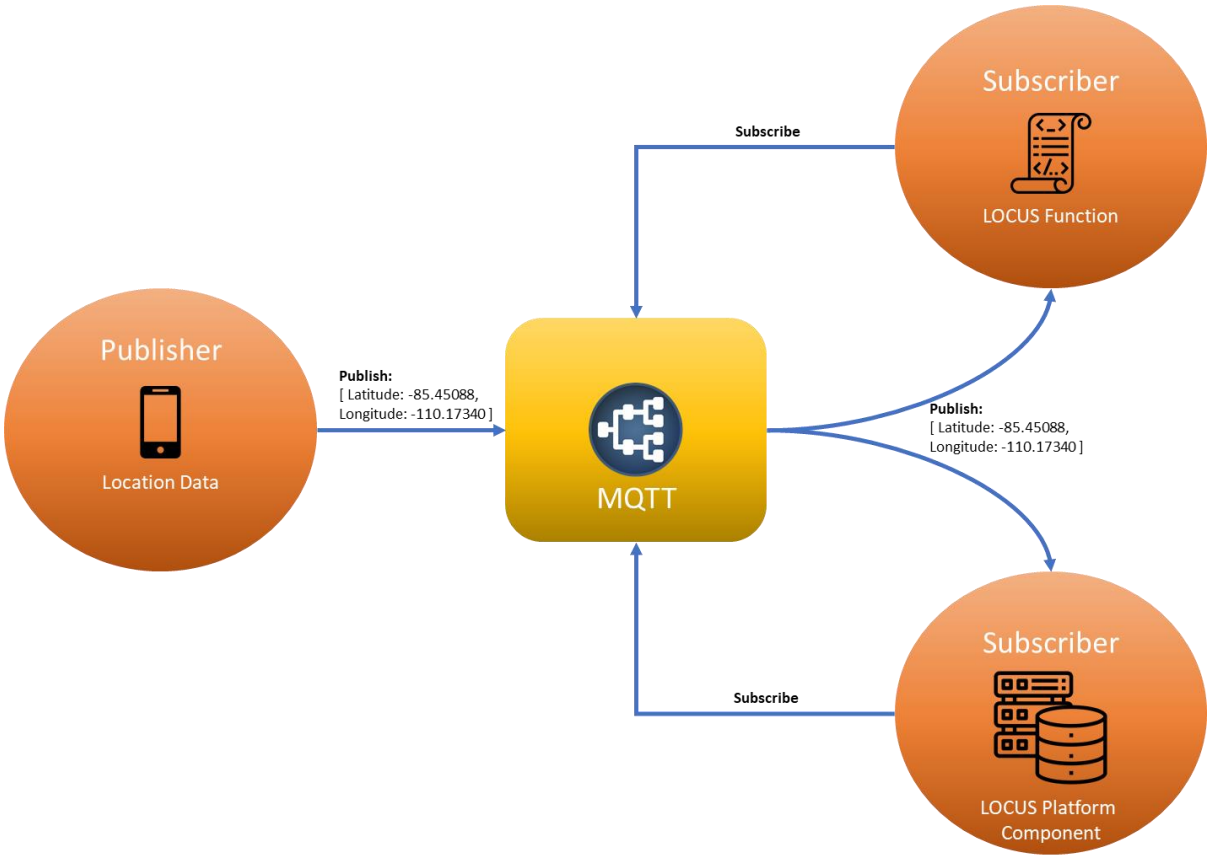

Figure 9: MQTT Publish / Subscribe Architecture

In the LOCUS Virtualization Platform, the RabbitMQ instance is reachable at:

amqp.locus-project.eu

on its default TCP port. For management and debugging purposes, this LOCUS platform component provides a web-based Graphical User Interface (GUI), reachable at:

amqp.locus-project.eu:15672

where LOCUS PoC users can check the topics and the frequency of the messages published through their components. The list of users available for the RabbitMQ is depicted in Figure 10.

Each of the LOCUS PoCs has a dedicated user account, with which they connect to the broker to publish or to subscribe to the relevant topics.

Figure 10: List of RabbitMQ users

2.2.4 Datastore

The LOCUS functions, as well as user equipment (UE) and other network devices generate different data that are used by other components of the infrastructure. To enable these components to store or to use these data, the LOCUS Virtualization Platform includes dedicated datastore component, where two different storage tools have been deployed. A SQL compliant database, namely PostgreSQL [15] is available to each PoC to store the generated data during their execution, and at the same time, provide to other PoC LOCUS functionalities to gather the data. Each PoC has a dedicated user, with whom is able to login to the dedicated database and execute the above-mentioned operations. Table 5 shows the list of the users and databases already available in the LOCUS datastore component.

Table 5: List of users and databases in PostgreSQL

User	Access to databases
postgres_poc1	poc1_db
postgres_poc2	poc2_db
postgres_poc3	poc3_db
postgres	All

The same LOCUS platform component presents the opportunity for the LOCUS functions and its consumer to store and retrieve data from a MongoDB instance [16]. MongoDB is a NoSQL database software that supports field, range query, and regular expression searches. It allows the user to use primary and secondary indices to index the fields in a MongoDB document.

Similar to the PostgreSQL instance, three different users and related databases have been created. Table 6 shows the details of the users created in MongoDB, databases and the role of each of these in the respective database.

Table 6: List of users and databases in MongoDB

User	Access to databases	Role
administrator	All	userAdminAnyDatabase
mongodb_poc1	poc1_db	userAdmin
mongodb_poc2	poc2_db	userAdmin
mongodb_poc3	poc3_db	userAdmin

Both PostgreSQL and MongoDB offer connectors to enable the connection between the consumer and the service itself. Both services are reachable at:

datastore.locus-project.eu

respectively on ports 5432 and 27017.

2.2.5 API Layer

The API Layer component is used to run, at the time of writing, part of the LOCUS Platform API blocks (with reference to Figure 1), which refer to components and capabilities provided by WP5. Indeed, the LOCUS Platform should support the demand for analytics results from varying 3rd party applications and with diverse service parameters. This is achieved only through the dynamic allocation of resources leveraged by the virtualization layer and the service discovery mechanisms incorporated in the system. In particular, the API Layer component run the Service Discovery Module.

As described in deliverable D2.5 [10], the Service Discovery Module is part of the LOCUS APIs block (realizing the API Gateway functions) and provides a centralized repository for microservices to register their IP/port pairs for each of their services. During the initial deployment tests, it was discovered that there was an issue with the static IP/ port configuration and the dynamic spawning of containers that occur when Kubernetes is in charge of the network stack. Therefore, the system needed to include a component that would store all the network locations centrally, and would map it to a specific service name. In detail, the internal analytics function instances are generated on demand, allowing for flexibility and scalability, and their internet and connectivity information is registered via a naming convention. This way, they can be accessed by client libraries though their network addresses for direct communication within the LOCUS analytics functions internal network. Furthermore, the module serves as a link between the external API and the internal services' status, keeping track of the latter while

–at the same time- communicating with the Service Subscription Module to provide information to users on the status of the requested API access.

Consul [33] is an open-source technology selected to accommodate the above-mentioned need and is currently deployed for PoC UCs demonstration purposes [4], allowing REST service discovery functions, such as service registration and service network address reverse lookup. In Figure 11, an indicative screenshot of the deployment of the Service Discovery Module at OTE premises is presented.

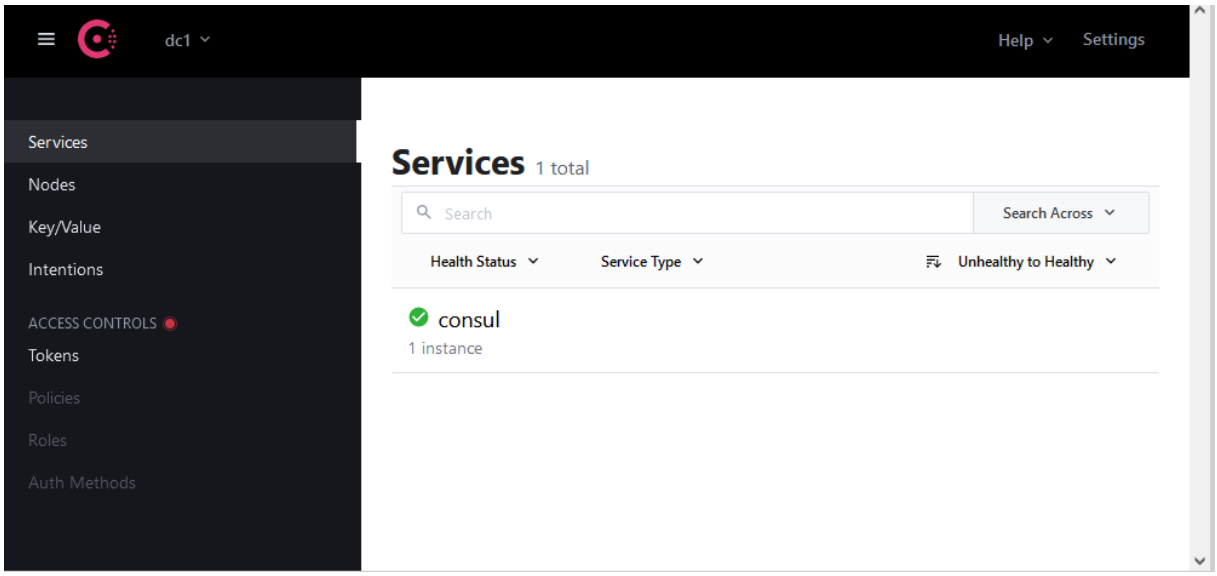

Figure 11: Deployment of the Service Discovery Module at OTE premises

2.2.6 DNS

The LOCUS Platform is enriched with two additional services, created to facilitate the interaction with other LOCUS components or to avoid the usage of external tools. These services run in a single virtual machine in the LOCUS Virtualization platform. The first one is a DNS service, built on top of BIND9 [17]. It is configured to respond as authoritative service for the domain

locus-project.eu

and its scope is limited to the LOCUS Platform namespace. As shown in the following figure (Figure 12), and also mentioned in Table 2, all the LOCUS platform components are identified by a name registered to the DNS service.

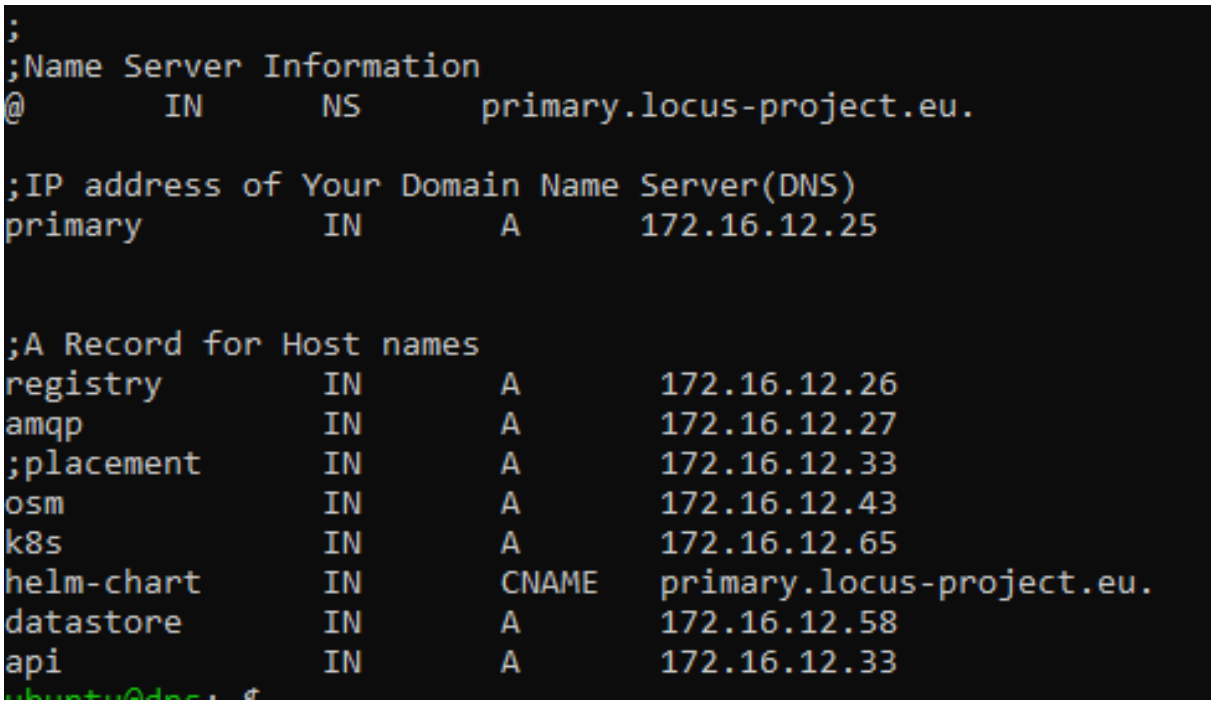

Figure 12: Configuration file for domain locus-project.eu

The DNS service is configured as primary DNS in the OpenStack installation, for the Provider Network, allowing all the components that request an IP address through Dynamic Host Configuration Protocol (DHCP) to automatically use the above DNS server.

The other service is web-based and serves the Helm Chart [19] repositories, one for each of the PoCs. It is based on Apache2 software [18], and configured to host the Helm Chart repositories in three different folders: *poc1*, *poc2* and *poc3*, containing the Helm chart data related to PoC#1, PoC#2 and PoC#3 respectively.

Each folder has:

- an *index.yaml* file that contains the list of all the helm packages available in that Helm Chart repository
- the Helm packages listed in the above index file.

Figure 13 depicts the content of the Helm Chart repository for PoC#1. Helm Charts are described in section 3.3.2.1.

```
root@dns:/var/www/html/poc1# ll
total 56
drwxr-xr-x 2 ubuntu root   4096 Feb 15 08:29 ./
drwxr-xr-x 5 ubuntu root   4096 Dec  9 14:37 ../
-rwxrwxr-x 1 ubuntu ubuntu  806 Dec  9 14:42 api-sender-0.1.0.tgz*
-rwxr-xr-x 1 ubuntu ubuntu 2272 Feb 15 08:29 index.yaml*
-rwxrwxr-x 1 ubuntu ubuntu 1656 Dec  9 14:42 locus-privacy-chart-0.1.0.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1658 Dec  9 14:42 locus-privacy-chart-0.1.1.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1658 Dec  9 14:42 locus-privacy-chart-0.1.2.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1574 Dec  9 14:42 locus-privacy-service-0.1.2.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1650 Dec  9 14:42 locus-test-0.1.1.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1953 Dec  9 14:42 mongo-privacy-service-0.1.2.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1946 Dec  9 14:42 mongo-privacy-service-0.1.3.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 1926 Dec  9 14:42 mongo-privacy-service-0.1.4.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 2137 Dec  9 14:42 mongo-privacy-service-0.1.5.tgz*
-rwxrwxr-x 1 ubuntu ubuntu 2137 Feb 10 15:52 mongo-privacy-service-0.1.6.tgz*
root@dns:/var/www/html/poc1#
```

Figure 13: PoC#1 Helm Chart repository content

3 LOCUS Management and Orchestration

The LOCUS MANO is a fundamental component within the LOCUS Platform. As described in deliverable D4.3 [1], it enables the use of the Virtualization Platform to deploy and operate the LOCUS functions as virtualized functions.

The main target of the LOCUS MANO is the automation of the lifecycle of these functions. It allows to deploy the localization and analytics services on-demand as combination of virtualized functions, and expose the produced analytics and geolocation-related data as a service to be consumed by the Smart Network Management and 3rd party vertical applications.

3.1 Final Architecture Design

The NFV MANO functionalities are key enablers for the automated management of 5G network functions and services. They are essential in the 5G network management architecture. As already specified in deliverable D4.3 [1], both 3GPP and ETSI NFV share a common approach in presenting NFV network services, where network slices are mapped as one or more recursive network services. Based on this approach, the LOCUS MANO provides capabilities to orchestrate localization and analytics functions and services in support of Smart Network Management or 3rd party vertical applications or.

The final design of the LOCUS MANO, depicted in Figure 14, follows the principle and architecture described in deliverable D4.3 [1]. In particular, the LOCUS MANO consists of three main components:

- *NFV MANO*: it provides the basic functionalities for lifecycle management of localization analytics functions and services, thus taking care of their on-demand deployment and configuration according to the NFV principles. It supports cloud-native containerized functions and services
- *Monitoring platform*: collect data and statistics from heterogenous sources, including network and computing virtualized resources as well as network and localization and analytics virtualized functions, and make them available to the other LOCUS MANO functions or dedicated dashboards
- *Smart NFV services*: they implement advanced management routines, and increase the level of automation and intelligence in operating at runtime the various services, functions and virtualized resources. These management routines include design, runtime optimization and fault management for the various virtualized services

No modifications have been introduced with respect to the D4.3 design [1], which is then still applicable and valid for this final prototype. It is worth mentioning that, with the aim of validating the main overall LOCUS platform functionalities and objectives, the final prototype of the LOCUS MANO includes the NFV MANO and the Monitoring platform as implemented

services (highlighted in green in Figure 14). The Smart NFV services have been kept at the design level only.

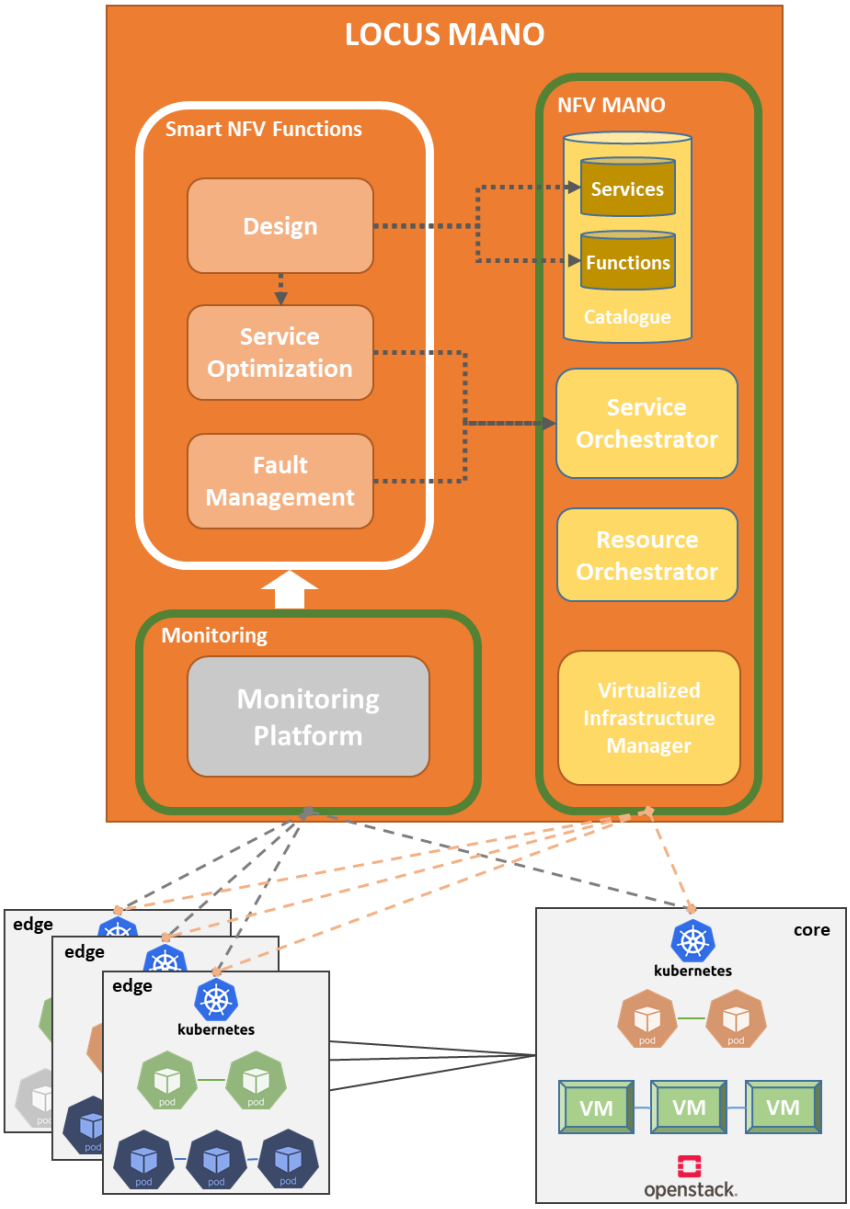

Figure 14: Evolved LOCUS MANO in the LOCUS Platform (source D4.3)

The implementation of these components is described in the following section (section 3.2).

3.2 Prototype Deployment in OTE Testbed

3.2.1 NFV MANO

The NFV MANO component, in the OTE testbed, is implemented through the ETSI OSM solution [9], a fully compliant ETSI NFV Orchestrator. It allows managing the full lifecycle

management of the LOCUS functions and services (i.e.: localization and analytics functions and services), able to catalogue, orchestrate and monitor these functions and services.

The instance of ETSI OSM in OTE testbed, based on the Release 10, allows LOCUS partners to onboard their functionalities, deploy, configure and monitor them.

The OSM instance is available at:

http://osm.locus-project.eu

and it is configured with three different users, and three projects, one for each PoC, as defined in WP6. The Table 7 shows the mapping of users and projects.

Table 7: Mapping between projects and users in LOCUS MANO

User	Project	Role
admin	All	System_admin
poc_1	PoC_1	Project_admin
poc_2	PoC_2	Project_admin
poc_3	PoC_3	Project_admin

Each project has a unique user, with admin rights to it. The different projects allow the different LOCUS partners to only access the resources of the platform dedicated to them.

For each of these, as depicted in the following Figure 15, a VIM account, a K8s cluster and a Helm Chart repository are configured.

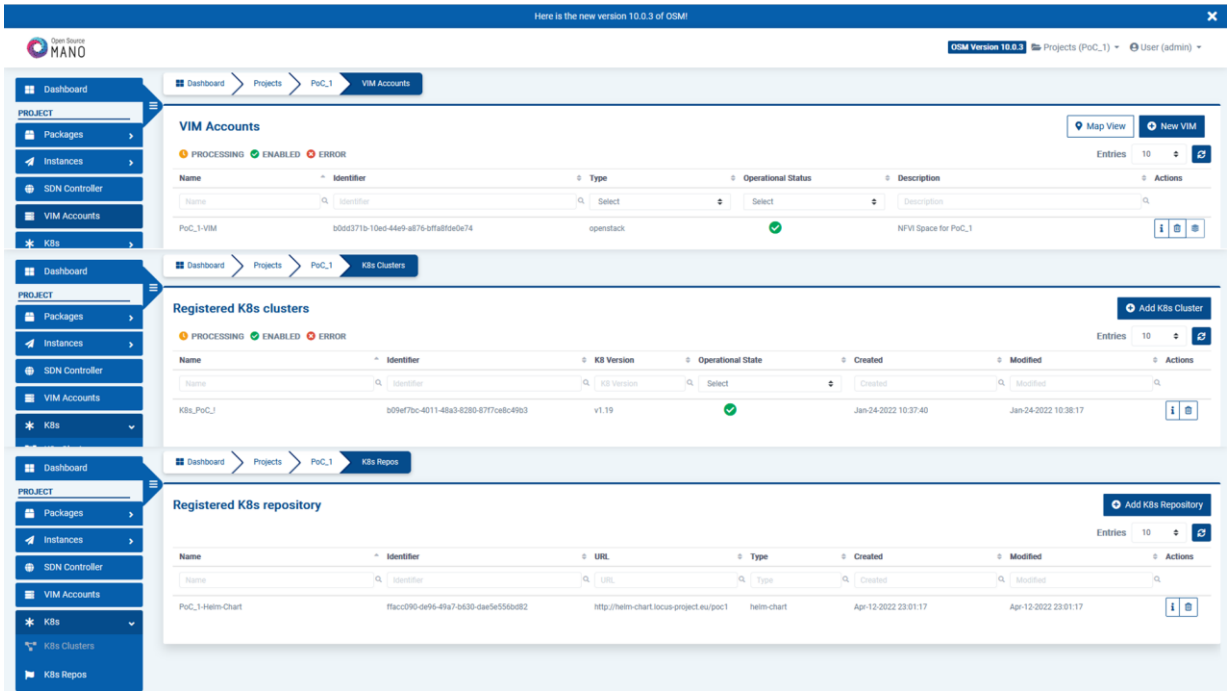

Figure 15: PoC#1 OSM Resources

Within the project the virtualized and packaged localization and analytics functions and services are onboarded to the NFV Catalogue integrated with ETSI OSM. The packaging of these functionalities is described in section 3.3. The functionalities that are onboarded, can be shown in the ETSI OSM GUI VNF Package page, within the Packages tab, as depicted in Figure 16.

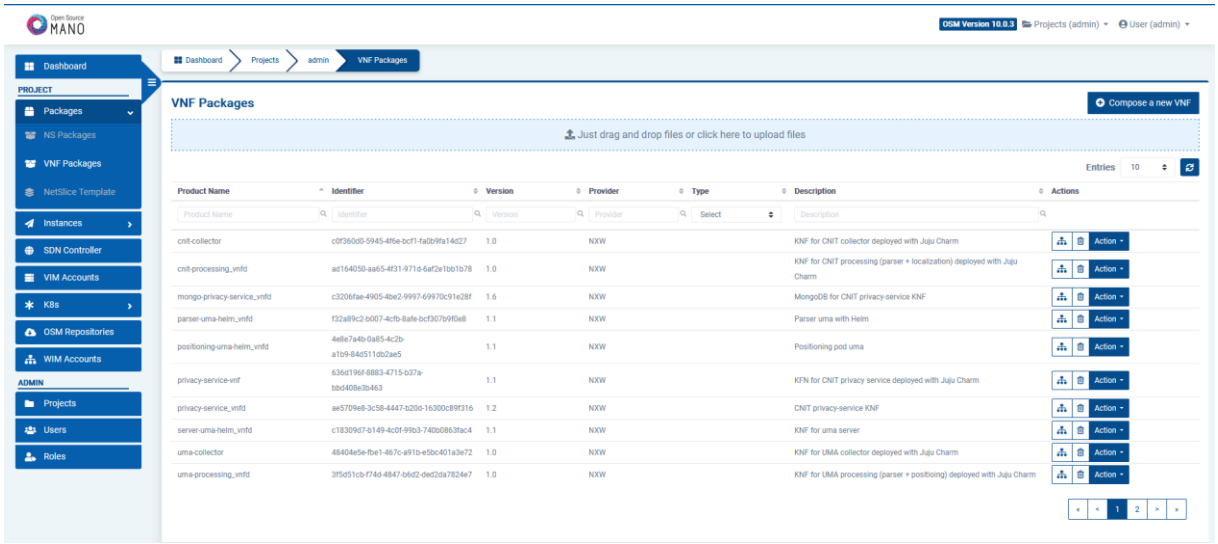

Figure 16: List of VNF/CNF localization and analytics functions for PoC#1

The localization and analytics services are packaged as network service packages (NS Packages). As their corresponding VNF Package, the LOCUS services can be found under the Package tab, in the NS Package page.

Once both VNF and NS packages are onboarded, the services can be instantiated allowing LOCUS platform components to interact with these services. The lifecycle management of the LOCUS services, as already mentioned above, is handled by ETSI OSM, through the resource orchestration module. All the running services can be checked in the Instances tab, in the NS Instances page, as shown in Figure 17.

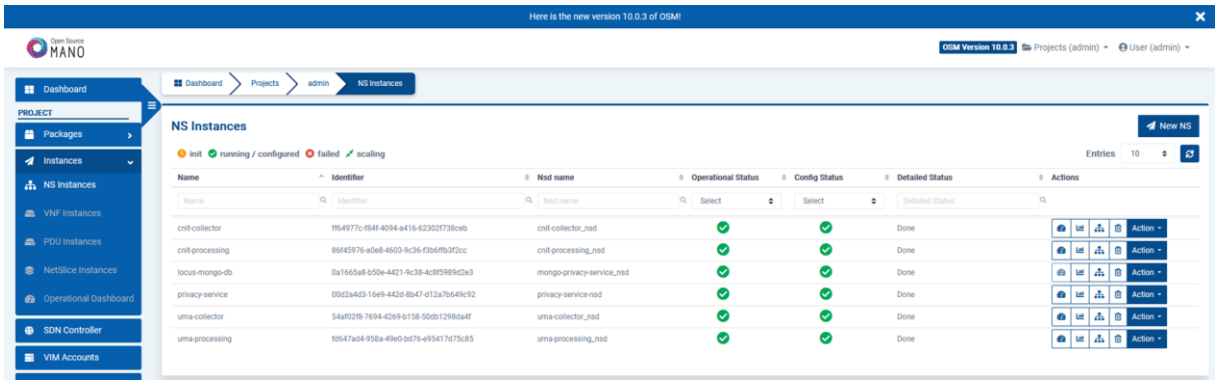

Figure 17: PoC#1 Running NS Instances

3.2.2 Monitoring platform

The monitoring platform is running in the Virtualization Platform Kubernetes cluster, as a dedicated management service. Following the design and functional decomposition specified in D4.3 [1], it is based on Prometheus [20], that is in charge of the collection of the metrics on the different LOCUS functions, and on Grafana [21], that offers the possibility to have a graphical view, over specific dashboards, of the gathered metrics. Figure 18 shows an example of the dashboards available in the monitoring platform. In this particular case, it shows the CPU utilization percentage, memory utilization, network traffic and network input/output requests related to the SI-based localization service (see section 4.3).

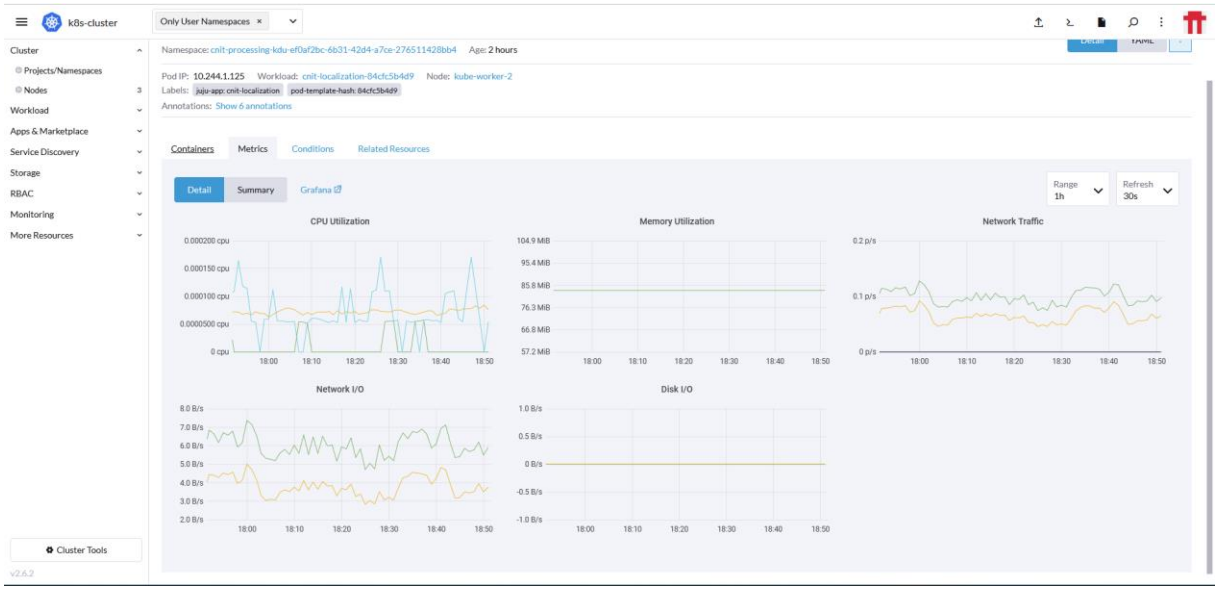

Figure 18: Example dashboard with metrics of a LOCUS Service

3.3 Packaging of LOCUS Functions and Services

The use of LOCUS functions and services within the LOCUS Virtualization Platform requires a packaging process to prepare these respectively as VNF packages and NS packages. This process is divided in two main steps summarized below:

- Creation of the virtualized image of the function composed of all the requirements to properly run the function into the LOCUS Virtualization Platform;
- Creation of the VNF and NS packages to be used for the instantiation of the network service into the LOCUS NFV MANO

3.3.1 LOCUS functions image packaging

The first step of packaging for LOCUS functions is the creation of a containerized image, containing the virtualization of functions, as source or binary code ready to be deployed into the LOCUS Virtualization Platform. As specified in deliverable D5.1 [22] and D4.3 [1], it is possible to virtualize the LOCUS function using two different techniques: Virtual Machine or docker container. The choice between those two possibilities is based on the requirements provided by the LOCUS function provider, through the function template defined in deliverable D4.3 [1].

A preliminary phase, before the function is virtualized, may require the update of the source code, adding to the function the opportunity to accept configuration parameters during the runtime execution of the service. This requirement is necessary for the dynamic configuration and reconfiguration of the function, known also as day-1/day-2 configurations. Once the function code is ready to be containerized, the process starts with the creation of a docker

file, that will subsequently create the virtualized docker image of the LOCUS function. The docker file contains all the information related to the LOCUS function (i.e.: the list of ports for all the services exposed by the function). After the creation of the docker file, the Docker image can be generated. This concludes the generation phase of the function image, and from this point it is possible to start the process of the creation of the VNF package.

3.3.1.1 LOCUS functions and services packaging for LOCUS NFV MANO

The next step on the virtualization of the LOCUS functions and services is the creation of the VNF and NS packages, respectively for LOCUS functions and LOCUS services. The first one will contain the reference to the LOCUS function, while the second one will contain the description of the service in terms of function and interconnection requirements.

3.3.2 VNF packaging

After the creation of the docker images of the containerized LOCUS functions, it is possible to start the MANO packaging process. This process includes the packaging of these into Kubernetes Network Functions (KNFs) that can be orchestrated from the LOCUS Management and Orchestration tool (i.e., ETSI OSM). As specified in D4.3 [1], a KNF is a special case of VNF (in the context of ETSI OSM), where the VNF itself is a containerized network function, thus developed and packaged as a cloud-native application (compatible with k8s deployments). It is possible to package the LOCUS functions using two techniques: i) Helm Chart and ii) Juju Charm [23]. The choice to select one or the other is based on the different characterization of the technique itself, and is described in the following paragraphs.

3.3.2.1 Helm Chart

Helm Chart is a Kubernetes standard deployment. Figure 19 shows how a Helm Chart package is composed, with all the files that describe the service.

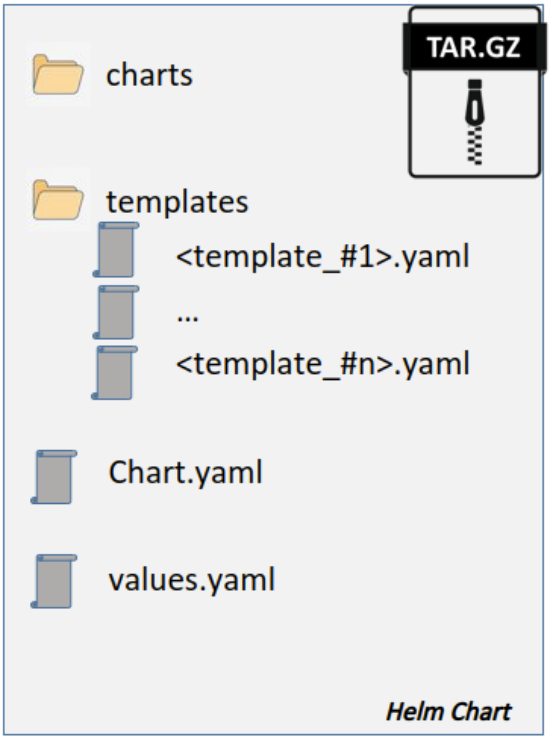

Figure 19: Helm Chart structure

Each Helm Chart package contains the following elements:

- ***charts***: is an auto-generated directory;
- ***templates:*** is a directory that contains all the templates that describe the function to be virtualized and its deployment;
- ***Chart.yaml*** contains the metadata of the chart (i.e., name, version, type of the chart, etc.);
- ***values.yaml*** contains the values of the parametrized field into the templates.

The package in its whole is referred to as a chart. A chart is deployed on the basis of its templates folder. The YAML files within this directory describe the type of the service, the type of deployment for the service and also specify the virtualized image that has to be used.

3.3.2.2 Juju Charm

The alternative approach of the Helm Charts for the MANO packaging as KNFs, as mentioned above, is Juju Charm. A Juju Charm package is composed of YAML descriptor and python script files that describe the behaviour of the function to be deployed. This package can be referenced as charm. A peculiarity of Juju Charm is that the behaviour of the function is handled by intercepting events. Some of these are default events (e.g., *on_start*, *on_config_changed*) and handle the creation of the charm, including the creation of the docker container, while other events are implemented to trigger specific day-1/day-2 configuration actions.

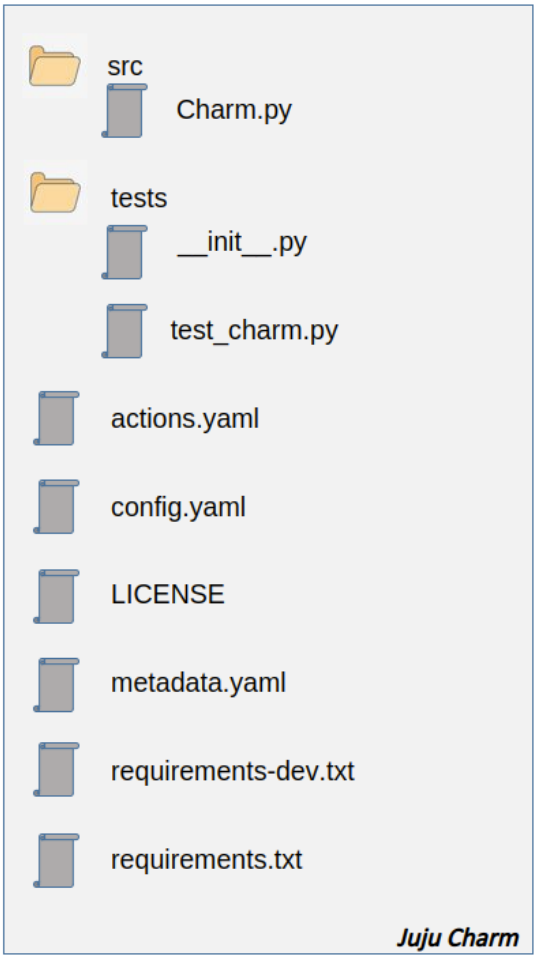

Figure 20: Juju charm structure

The structure of a Juju Charm is depicted in Figure 20. The main components composing it are described below:

- ***charm.py*** file represents the source code of the Juju Charm, where the following are defined:
  - the behaviour of the Charm when an action event occurs,
  - the containerized image of the function that the charm implements;
- ***actions.yaml*** contains the actions that the charm can perform and describes the default parameters required to execute the action;
- ***config.yaml*** is an optional file that contains settings to be applied to the charm. These settings can be also applied at runtime through the source code;
- ***metadata.yaml*** contains the information about the service and how it has to be deployed. It defines the type of deployment (i.e., stateful, stateless or daemon) and the type of service (i.e., loadbalancer, cluster, external or omit). In the LOCUS function virtualization process only a subset of deployment and service types are used.
  - Deployment:

- **stateful**: with which the service will be deployed as a stateful set, i.e., a service that can manage the deployment and the scaling of K8s Pods [24].

- Service:
  - **loadbalancer**: The service has two different IP addresses. The first one is an internal IP, limited to the cluster space. It can be used by other services inside the Kubernetes Cluster to reach the LOCUS function, while the second one has an external IP, allowing external entities to reach the function from outside the cluster;
  - **cluster**: The service has an internal IP address assigned, that makes the container reachable only by services defined into the same cluster.

In the context of the LOCUS Virtualization Platform, both alternatives of packaging have been used with a preference to Juju Charm that allows dynamic configuration through the day-1/day-2 configurations.

3.3.2.3 VNF Descriptor

Once the VNF/KNF is packaged, it is possible to define its VNF Descriptor, to be used into LOCUS MANO, in compliance with ETSI NFV SOL006 [25].

The VNF/KNF Descriptor contains the information about the external connection and the reference to the Helm Chart or Juju Charm that is used to deploy the function. A Kubernetes Deployment Unit (KDU) is the basic part of KNF and corresponds to a Kubernetes POD.

A *kdu* object can specify with which type of deployment the service has to be instantiated.

In case of Helm Chart, the *kdu* object references a Helm Chart repository where the chart is located and from where it can be downloaded. Whilst, for Juju Charms the *kdu* object references a juju-bundle that specifies the location of the juju charm. For KNF that uses Juju Charm to virtualize the function the descriptors also include the declaration of the parameters to be passed to the charm source code and to be used into day-1 and day-2 configurations.

Figure 21 depicts the contents of a VNF package that uses Juju Charm. In addition to the descriptor, the juju-bundle folder is present and it contains the *bundle.yaml* file and the directory where the built Charm is located and referenced into the bundle file.

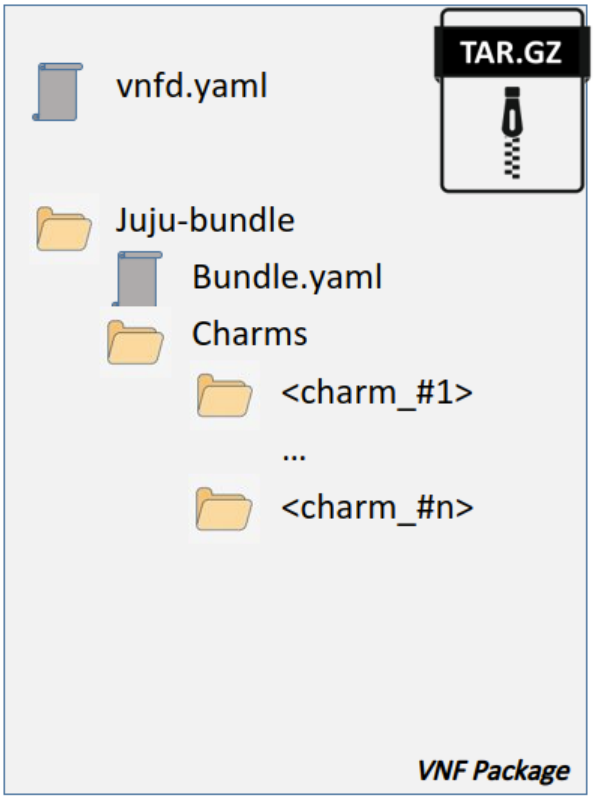

Figure 21: VNF Package structure

3.3.3 NS packaging

Once the VNF package is created, it is possible to create the NS package that contains the Network Service Descriptor (NSD). The NSD is a YAML file complaint with ETSI NFV SOL006 [25] and contains the description of the service mapping the virtual link where the service is exposed with the virtualized function that implements the service through the VNF Descriptor. The virtual link is referred to by the virtual link identifiers and maps a virtual link that is present into the Kubernetes Cluster network. The virtualized function is referred to with the VNF Descriptor identifier of the VNFD where it is defined.

4 Integration of LOCUS PoC Functions and Services

This section describes how some initial LOCUS localization analytics functions and services have been implemented, packaged, virtualized, deployed and integrated in the LOCUS Virtualization Platform through the LOCUS MANO. Specifically, the section covers only part of the LOCUS functions and services in scope of the project PoCs, concentrating on those ready and available at the time of writing this deliverable. More will be integrated as part of the WP6 activities.

It is worth to mention that the goal of this section is to mostly provide details on the packaging and virtualization aspects of these LOCUS PoC functions and services, describing how they have been integrated in the LOCUS Virtualization Platform and orchestrated through the LOCUS MANO, rather than how they integrate and fully map with the overall LOCUS Platform architecture. This latter aspect is mostly considered in WP6, where the integration of the whole LOCUS Platform components and functional boxes (i.e., with reference to the D2.5 architecture diagram shown in Figure 1) is carried out.

4.1 High precision accuracy service

As described in D6.2 [4], the high precision accuracy service developed by UMA uses ranges (distances to reference points) to perform trilateration and obtain a location estimation. Normally, these ranges would come from a single technology (e.g., LTE, 5G, etc.), but in this service, ranges from different technologies can be fused. The service will listen to any range provider that makes data available in the LOCUS Message Queue Module within the LOCUS platform, implemented, as previously described, through a RabbitMQ message broker. In the current scenario, UMA has tested fusion of UWB, WiFi FTM and LTE. Using the fusion of the above technologies, a great improvement of the location precision is observed. The following figure (Figure 22) shows the service within the context of the LOCUS Virtualization Platform.

In this diagram, the UE measures a magnitude (i.e., the RSRP) of a reference point (i.e., a 5G gNB), and reports it to a Data Collection Module (i.e., the Collector orange box) of the LOCUS Platform. This component publishes the raw reports of the UE in RabbitMQ message broker. A parser element will use the measured magnitude to estimate the range to the reference point. With ranges to several reference points from a UE, the Positioning service will then estimate the location using fusion techniques. This location can then be used by other blocks; for instance, to calculate a coverage map, using the estimated location with other measurements obtained from the terminal.

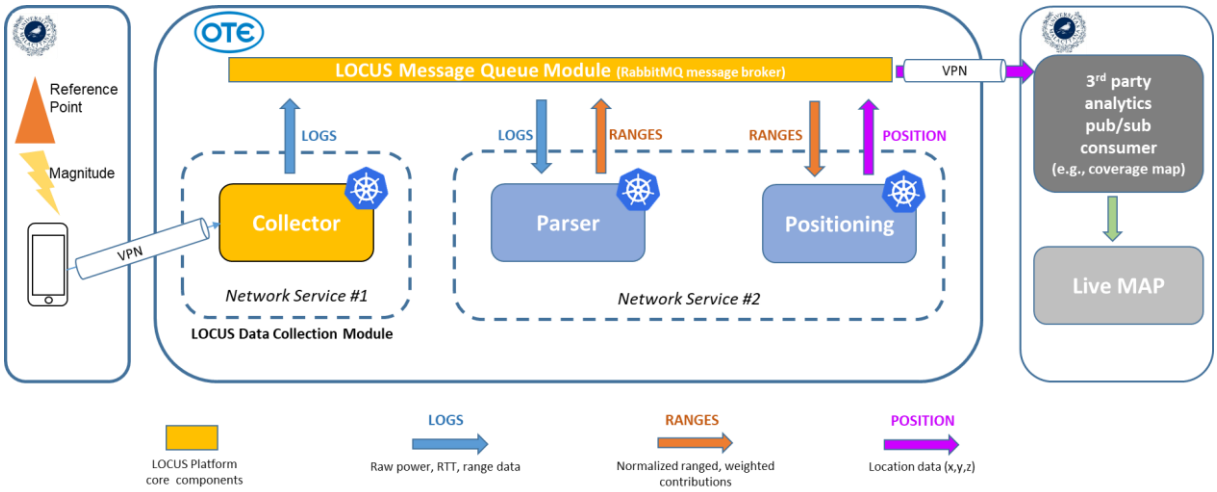

Figure 22: High precision accuracy service architecture

Apart from enhancing the end-user location precision, the multi-technology fusion used by the positioning service also provides a seamless navigation between areas served by different technologies (e.g., UWB and WiFi deployments using other ranging technologies such as LTE to cover for the missing ranges in the borders). In the following Figure 23, a map of the testbed is shown, with the location of the different reference points.

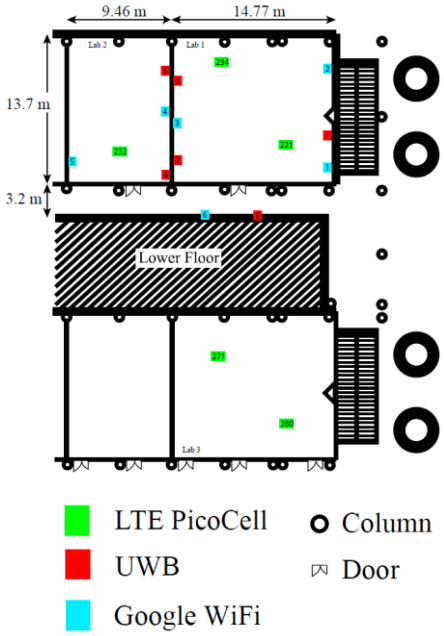

Figure 23: UMA testbed map

As Figure 24 depicts, the precision accuracy service is composed of two different NFV Network Services, collector and processing, both of them deployed using Juju Charm.

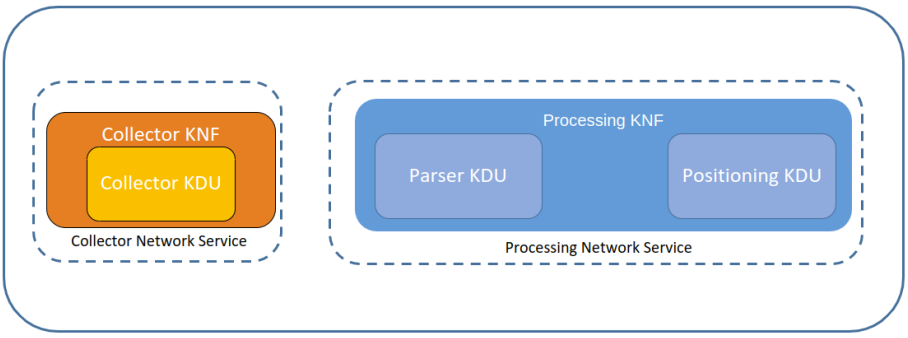

Figure 24: High precision accuracy service deployment

The two network services, that will be described in detail in the paragraphs below, both contain a single KNF, respectively called collector KNF and processing KNF. These Kubernetes Network Functions represent the deployments that the two services require. Within the collector KNF it is deployed a single KDU, collector KDU, that deploys the collector function to collect the UEs' data. Otherwise, in the processing KNF two different KDUs are deployed, the parser KDU and the Positioning KDU. The first one elaborates data taken from the collector and the second one estimates the position and produces the output.

4.1.1 Collector service

The collector service allows the UEs connected to the UMA infrastructure to send their location data to the LOCUS Platform by pushing it on the message broker.

The NS package required to instantiate this service is composed of a single file, the collector NSD.

```
nsd:
    nsd:
    -   description: NSD with UMA collector KNF
        designer: NXW
        sapd:
        -   id: cp_ext
            virtual-link-desc: provider
        df:
        -   id: default-df
            vnf-profile:
            -   id: '1'
                virtual-link-connectivity:
                -   constituent-cpd-id:
                    -   constituent-base-element-id: '1'
                        constituent-cpd-id: cp-ext
                    virtual-link-profile-id: provider
                vnfd-id: uma-collector_vnfd
        id: uma-collector_nsd
        name: uma-collector_nsd
        version: '1.0'
        virtual-link-desc:
        -   id: provider
            mgmt-network: true
        vnfd-id:
        - uma-collector_vnfd
```

Figure 25: Collector's NS Descriptor

Figure 25 shows the content of the collector's NSD with focus on the mapping between the virtual link identified by the ID ***provider*** and ***uma-collector_vnfd***, that is the identifier of the VNF Descriptor where the collector function is defined.

```
collector-uma_vnf
├── juju-bundles
│   ├── bundle.yaml
│   └── charms
│       └── collector-uma
└── uma-collector_vnfd.yml
```

Figure 26: Collector VNF package

As Figure 26 depicts, the collector VNF package follows the structure presented previously in section 3.3. It is possible to observe how the collector service is composed of a single charm, named **collector-uma,** whose location is defined into the bundle file and referred into the **uma-collector_vnfd.yml.**

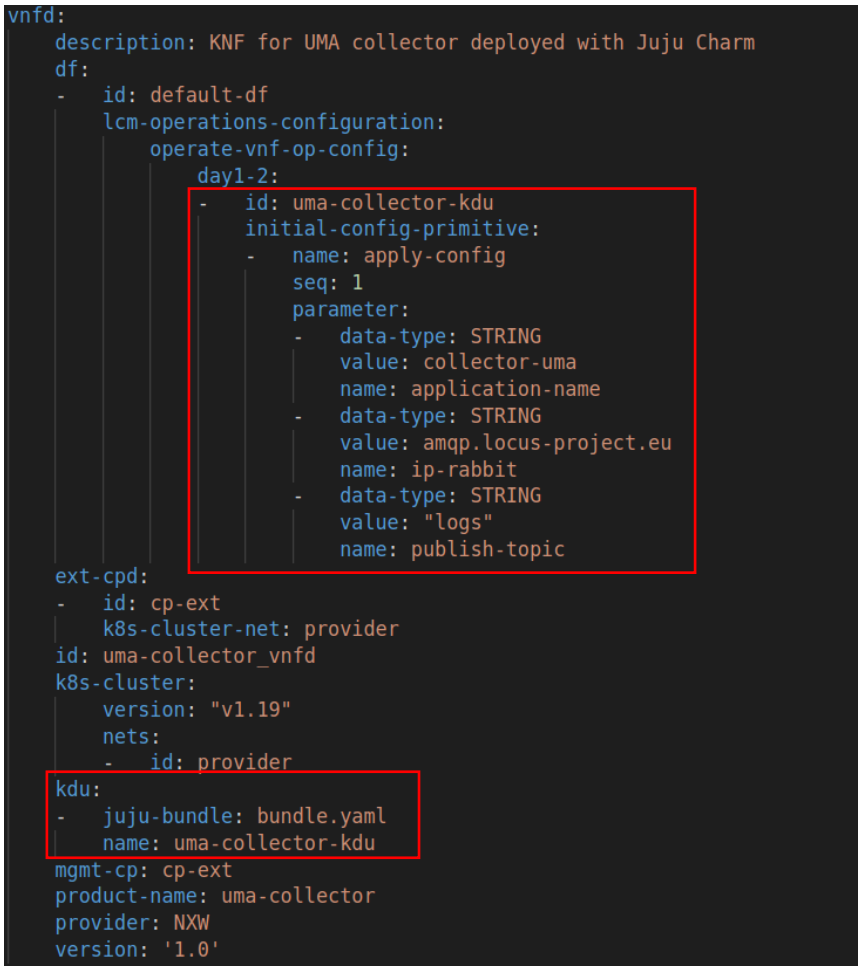

Figure 27: Collector VNF Descriptor

Figure 27 describes the content of the collector VNF Descriptor. It is possible to observe what has been specified in the Section 3.2, the ***kdu*** object refers to the bundle file where the charm is located. The key ***initial-config-primitive*** identifies the object that contains the parameters to be passed to day-1 configuration, and in this specific case to the action called ***apply-config***. The required parameters are described below:

- **application-name:** the value assigned to the application into the bundle file, i.e., *collector-uma*;
- **ip-rabbit**: the IP address of the message broker, the value assigned is the IP address of the RabbitMQ [11] instance present into the LOCUS Virtualization Platform;
- **publish-topic**: the topic where the collector function publishes the data collected from UEs (i.e., logs).

The collector charm to be deployed uses the containerized image that virtualizes the collector function available in the LOCUS Container Registry. This step of the deployment is executed when the default event on_config_changed occurs.

Figure 28 depicts the ***_on_config_changed*** function of the collector charm where the specification of the container is implemented. The specification contains the location of the virtualized image and the mapping with the local container port where the service will be reachable.

```
def _on_config_changed(self, _):
    if not self.unit.is_leader():
        self.unit.status = ActiveStatus()
        return
    try:
        self.unit.status = MaintenanceStatus("Applying pod spec")
        # Initializing pod
        containers = [
            {
                "name": self.framework.model.app.name,
                "image": "registry.locus-project.eu/locus_uc1/locus_uma:collector",
                "ports": [
                    {
                        "name": "flask-port",
                        "containerPort": 5000,
                        "protocol": "TCP",
                    }
                ],
            }
        ]

        self.model.pod.set_spec({"version": 3, "containers": containers})

        self.unit.status = ActiveStatus()
        self.app.status = ActiveStatus()

    except OCIImageResourceError:
        self.unit.status = BlockedStatus("Error fetching image information")
        return
```

Figure 28: Collector's virtualized image specification

4.1.2 Processing Service

The processing service is composed of two different functions, parser and positioning, respectively in charge of cleaning the data sent by the UE and calculating the positioning of the user.

The NS package contains, also in this case, only the processing NSD. The structure of the descriptor, shown in Figure 29, is the same of collector's one, with only a difference: the mapping between the virtual link identifier ***provider*** will be with the provisioning VNF Descriptor identifier, ***uma-processing_vnfd.***

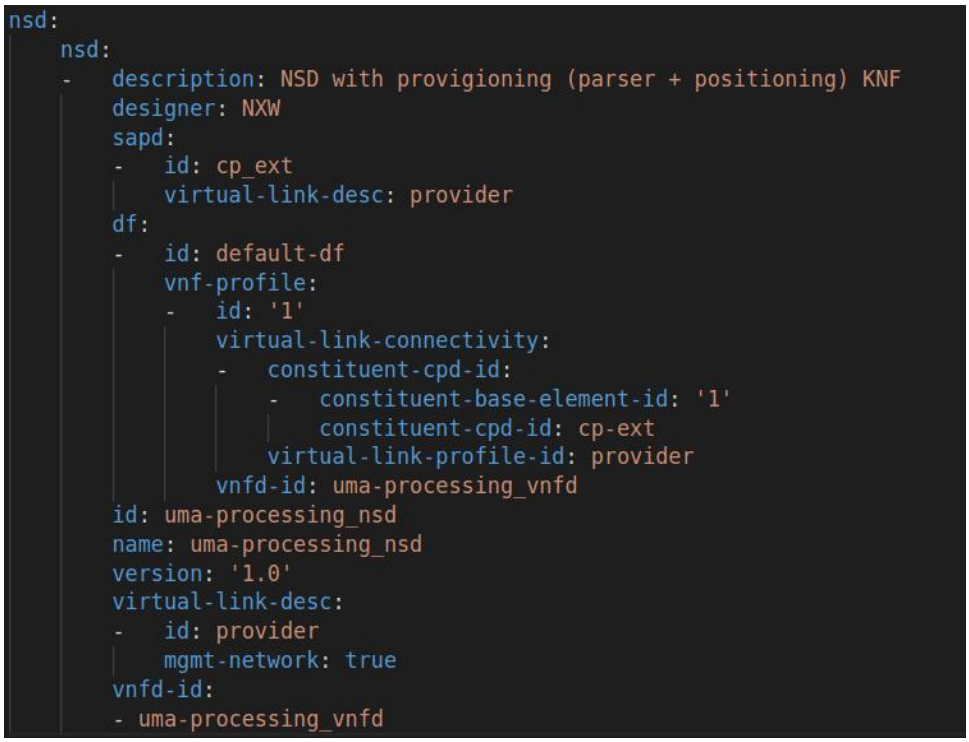

Figure 29: Processing NSD

Figure 30 depicts the content of the processing VNF package that follows the baseline shown in Section 3.3.2.

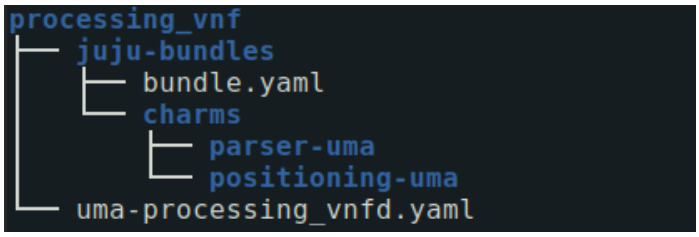

Figure 30: Processing VNF package

The *charms* directory for the processing service contains two charms, the first one named **parser-uma**, that implements the *parser* function, and the second one named **positioning-uma**, that implements the *positioning* function.

The processing VNF Descriptor differs from the collector ones for the vnfd-id ***uma-processing_vnfd*** and for the object identified by the key ***initial-config-primitive***, shown in Figure 31 and Figure 32.

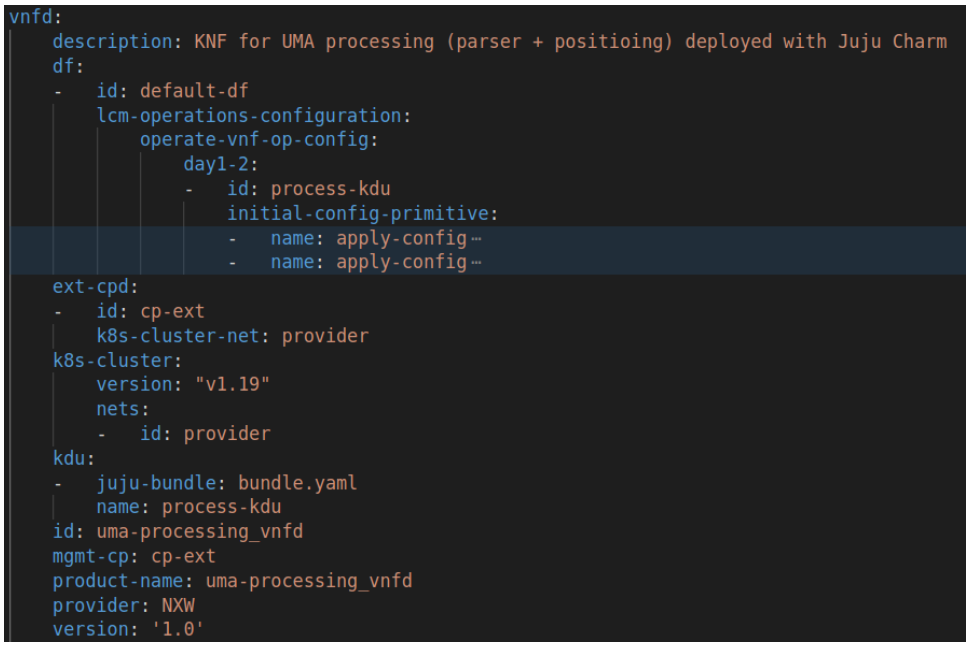

Figure 31: Processing VNF Descriptor

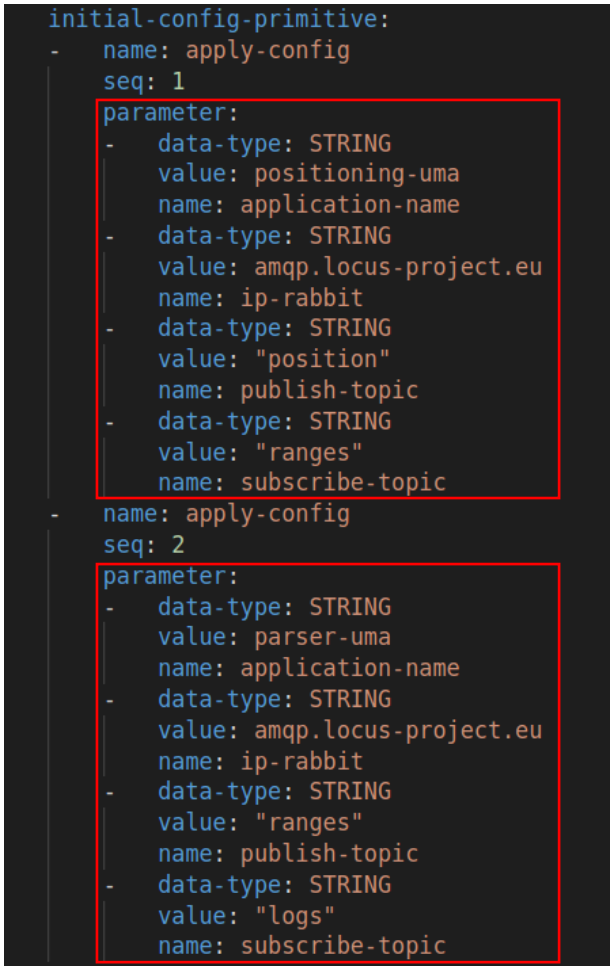

Figure 32: Processing initial config primitive parameters

Both the functions, positioning and parser, have been configured with day-1 configuration. The description of the required parameters follows:

- Positioning function
    - **application-name**: the value assigned to the application into the bundle file, i.e., positioning-uma;
    - **ip-rabbit**: the IP address of the message broker, the value assigned is the IP address of the RabbitMQ instance present into the LOCUS Virtualization Platform;
    - **publish-topic**: the topic where the positioning function publishes the calculated position data, i.e., position;
    - **Subscribe-topic**: the topic where the positioning function takes the data from the message broker, i.e., ranges;
- Parser function
    - **application-name**: the value assigned to the application into the bundle file, i.e., parser-uma;
    - **ip-rabbit**: the IP address of the message broker, the value assigned is the IP address of the RabbitMQ instance present into the LOCUS Virtualization Platform;
    - **publish-topic**: the topic where the parser function publishes the data after it have cleaned it, i.e., ranges;
    - **Subscribe-topic**: the topic where the parser function takes the data from the message broker, i.e., logs;

Parser and positioning charms follow the same approach shown in Figure 28 to deploy their function. The differences between the collector and these two are the tag of the containerized image, respectively *parser* and *positioning*, and the ports mapped to expose the TCP service, port 80 for parser and port 100 for positioning.

4.2 Privacy Service

The privacy service supports the positioning use case scenario where users execute queries with position and content request to Location Based Service (LBS). The privacy service employs k-anonymity and results aggregation functions. k-anonymity processes the data to make the output related to one user indistinguishable from at least k-1 other individuals. While result aggregation uses aggregate responses to respond to a 3rd party LBS application, the aggregate request is pushed to the LBS.

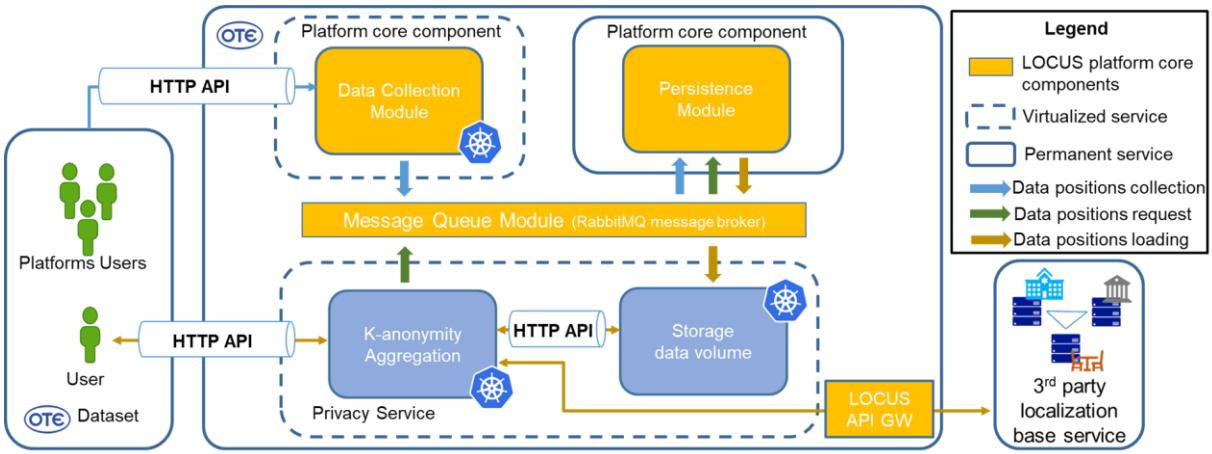

Figure 33: Architecture of the deployed privacy service

Figure 33 depicts the architecture of the privacy service and the interaction with the LOCUS platform core components (with reference to the LOCUS architecture shown in Figure 1). The service is composed of two different blocks. The first one, the k-anonymity and aggregation function receives users' requests, apply the proposed algorithms, and makes the results available to the external 3rd party LBS through the LOCUS APIs functionalities (i.e., Analytics Northbound API Gateway, as shown in Figure 1). The second block maintains the data structure used from the first function. Both the blocks are virtualized in the platform and use HTTP APIs to communicate. The integration with the LOCUS API Gateway functionalities are still work in progress and reported here for properly mapping the privacy service with the LOCUS Platform architecture.

Moreover, Figure 33 shows the Data Collection Module and the Persistence Module. Both are LOCUS platform core components, the first one allows the loading of users' data to the LOCUS Platform, and the second maintains the complete users' data available by other LOCUS functions and services functionalities. The Data Collection Module is virtualized in the platform, while the Persistence Module is permanently instantiated in the platform.

The privacy service has been integrated with the LOCUS Virtualization Platform that is hosted in the OTE infrastructure and can be deployed on-demand through the LOCUS MANO. Data from user real-world mobility traces are connected to the platform.

The connection between privacy service and other platform modules has been addressed by the LOCUS Message Queue Module, implemented through the RabbitMQ message broker in the LOCUS Virtualization Platform. More specifically, as shown in Figure 33, three topic messages are contemplated in the architecture:

- the “data positions collection” topic, it is used from Data Collection Module to publish data that will be stored into the Persistence Module, this latter subscribes the topic;

- the “data positions request” topic, it is used from privacy service to publish request of position data to the Persistence Module, that therefore needs to subscribe to this topic;
- finally, the “data positions loading” topic is used from the Persistence Module to publish request position received on the “data positions request” topic. Privacy service subscribes to the “data positions loading” topic.

Figure 34 depicts the Privacy Service deployment, which is composed of a single NFV NS. This NS contains a single KNF named Privacy Service KNF, which deploys a single KDU named Privacy Service KDU. As shown in Figure 33, and according to its deployment, both K-anonymity Aggregation and Storage data volume modules are implemented within the same KDU.

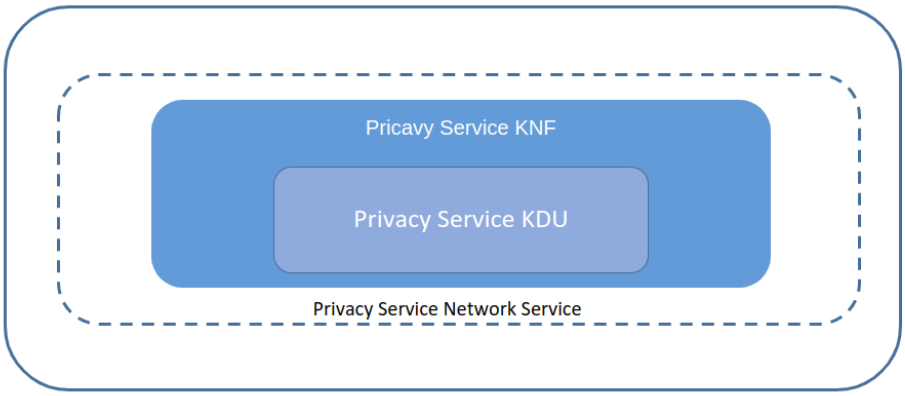

Figure 34: Privacy Service deployment

4.2.1 Privacy Service

The Privacy Service is composed of a single function, privacy service, that is composed itself of two different module, K-anonymity aggregation and Storage data module, respectively in charge of calculating the aggregation while preserving the privacy of the user and take a selected portion of users’ data from the LOCUS Persistence Module.

The Privacy Service NS package contains the Privacy Service NS Descriptor. Figure 35 shows the content of the NSD with a focus on the mapping between the virtual link identifier, ***provider***, and the VNF Descriptor identifier, ***privacy-service-vnf***.

```
nsd:
  nsd:
  - description: NSD for KNF CNIT privacy service deployed with Juju Charm
    designer: OSM
    df:
    - id: default-df
      vnf-profile:
      - id: '1'
        vnfd-id: privacy-service-vnf
        virtual-link-connectivity:
        - constituent-cpd-id:
          - constituent-base-element-id: '1'
            constituent-cpd-id: cp-ext
          virtual-link-profile-id: provider

    id: privacy-service-nsd
    name: privacy-service-nsd
    version: '1.1'
    virtual-link-desc:
    - id: provider
      mgmt-network: true

    vnfd-id:
    - privacy-service-vnf
```

Figure 35: Privacy Service NS Descriptor

Figure 36 depicts the content of the Privacy Service VNF package. This package follows the specification shown in Section 3.3.2.2.

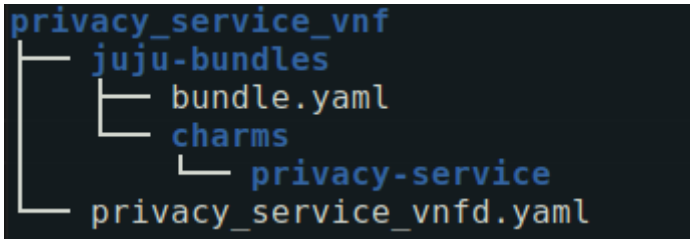

Figure 36: Privacy Service VNF package

From the above descriptor, depicted in Figure 36, it is possible to observe the *privacy-service* charm that corresponds to the single KDU that is defined within the Privacy Service KNF, according to what is previously specified.

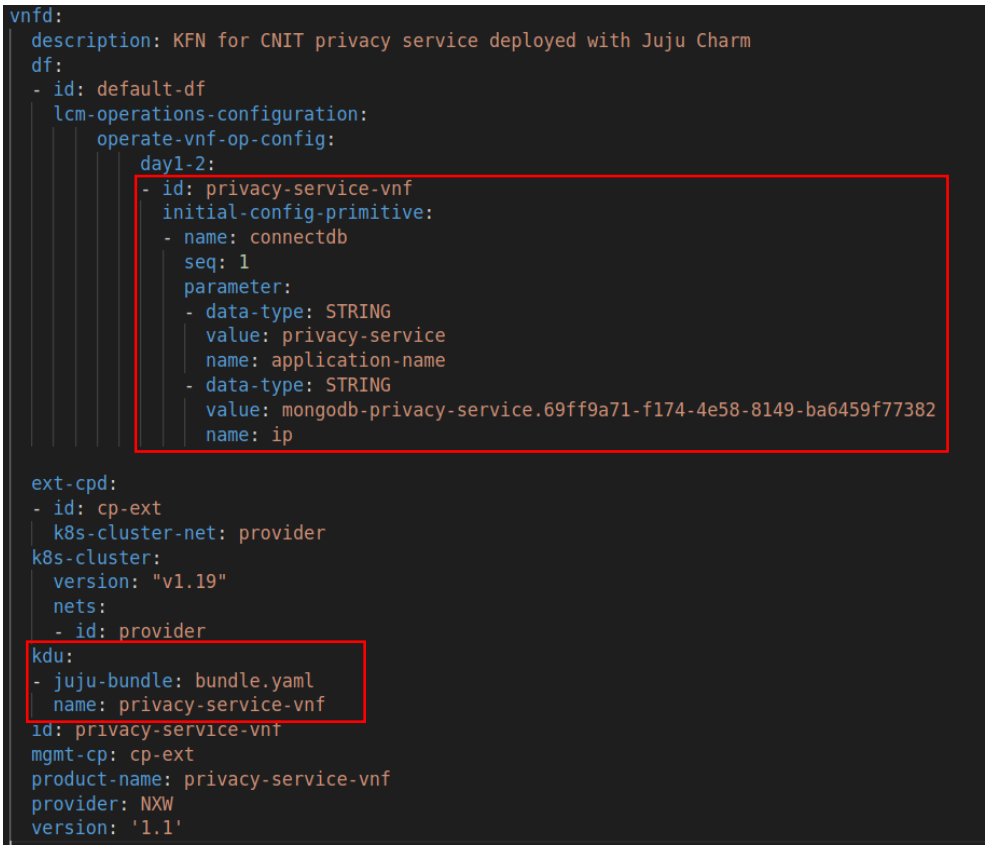

Figure 37: Privacy Service KNF Descriptor

Figure 37 describes the content of the Privacy Service KNF Descriptor with a focus on two different objects. The first one is the ***kdu*** object that specifies the name of the KDU and the location of the charm in order to implement the Privacy Service function. The second one is the ***initial-config-primitive*** object that identifies the day-1 configurations that have to be applied to the Privacy Service charm during its instantiation. The only action that has to be executed on the charm is the one named ***connectdb*** that allows the Privacy Service to reach the LOCUS Data persistence module.

The description of the required parameters is presented below:

- **application-name**: the value assigned to the application into the bundle file, i.e., privacy-service;
- **Ip:** the IP address where it is possible to reach the LOCUS Data persistence module.

The privacy service charm to be deployed uses the containerized image that virtualized the collector function available in the LOCUS Container Registry. This step of the deployment is executed when the default event on_config_changed occurs.

The following figure (Figure 38) depicts the ***_on_config_changed*** function of the privacy service charm where the specification of the container is implemented. The specification

contains the location of the virtualized image and the mapping with the local container port where the service will be reachable.

```
def _on_config_changed(self, _):
    if not self.unit.is_leader():
        self.unit.status = ActiveStatus()
        return
    try:
        self.unit.status = MaintenanceStatus("Applying pod spec")
        # Initializing pod
        containers = [
            {
                "name": self.framework.model.app.name,
                "image": "registry.locus-project.eu/locus_uc1/cnit-privacy-service:privacy-service",
                "ports": [
                    {
                        "name": "flask-port",
                        "containerPort": 5000,
                        "protocol": "TCP",
                    }
                ],
            }
        ]

        self.model.pod.set_spec({"version": 3, "containers": containers})

        self.unit.status = ActiveStatus()
        self.app.status = ActiveStatus()
    except OCIImageResourceError:
        self.unit.status = BlockedStatus("Error fetching image information")
        return
```

Figure 38: Privacy Service's virtualized image specification

4.3 SI-based Localization Service

Soft information (SI)-based localization has been developed in WP3 for providing accurate location estimates in 5G networks, exploiting both 5G radio access technology (RAT)-dependent and RAT-independent measurements [26][27][28][29][30][31]. In particular, SI-based localization relies on generative models (i.e., machine learning models) learned from the environment in order to probabilistically characterize the relation between the measurements and the UE positional features (e.g., distance, direction, and velocity) [32]. In D6.2 [4] a first implementation of the NFV NS providing position estimates based on SI-based localization algorithm has been presented and tested in a local environment. As detailed in D6.2 [4], the service is composed of three VNFs/KNFs (namely, *Collector*, *Parser,* and *Localization*) communicating through the LOCUS Message Queue Module, as anticipated above implemented in the Virtualization Platform Advanced through a RabbitMQ message broker [11]. The Collector VNF/KNF is an implementation of the LOCUS Data Collection Module, and provides similar functionalities to the one described in section 4.1. At the time of writing, they are implemented and packaged as two different VNFs/KNFs, but they will be integrated in a single LOCUS Data Collection Module implementation as part of the upcoming activities in WP6. With respect to the D6.2 [4], SI-based localization service has been fully

integrated in the LOCUS Virtualization Platform and several improvements to the VNFs/KNFs have been implemented:

- The *Collector* is now able to process real-time data (or validation data) from sources external to the LOCUS Platform through a REST interface. The communication with sources external to the LOCUS Platform is possible through the VPN used to access the OTE cloud infrastructure. As anticipated, this virtual function is similar to the Collector described in section 4.1, and will be available as a single VNF/KNF package for the final LOCUS PoCs
- The configuration of the VNFs/KNFs is accomplished through configurations files which are provided to the virtual functions during the deployment phase. Different configuration files are used to control the parameters related to the service deployment and the configuration of the localization algorithm parameters.
- The implementation of the algorithms in the virtualized functions has been improved and optimized for reducing the inference latency.

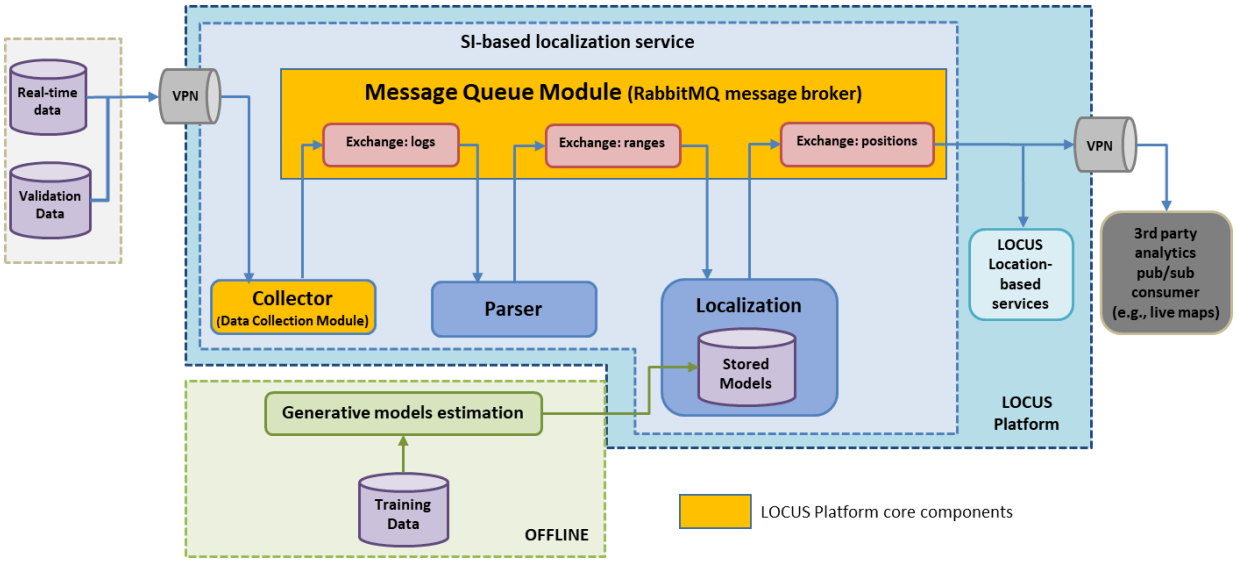

Figure 39: Architecture of the SI-based localization service.

Figure 39 shows the updated architecture of the SI-based localization service. The positions estimated by the SI-based localization service are forwarded to the RabbitMQ exchange called *positions* and can be accessed by different types of location-based services internal to the LOCUS Platform (e.g., analytics services). In addition, services external to the LOCUS Platform such as live navigation using digital maps services can access to the positions provided by the SI-based localization service via the message bus, in line with the approach supported by LOCUS and described in D2.5 [10].

4.3.1 Collector Service

The collector service allows the UEs connected to the infrastructure to send their location data to the LOCUS Platform pushing it on the message broker.

The NS package required to instantiate this service is composed of a single file, the collector NSD.

```
nsd:
    nsd:
    -   description: NSD with CNIT collector KNF
        designer: NXW
        sapd:
        -   id: cp_ext
            virtual-link-desc: provider
        df:
        -   id: default-df
            vnf-profile:
            -   id: '1'
                virtual-link-connectivity:
                -   constituent-cpd-id:
                    -   constituent-base-element-id: '1'
                        constituent-cpd-id: cp-ext
                    virtual-link-profile-id: provider
                vnfd-id: cnit-collector_vnfd
        id: cnit-collector_nsd
        name: cnit-collector_nsd
        version: '1.0'
        virtual-link-desc:
        -   id: provider
            mgmt-network: true
        vnfd-id:
        - cnit-collector_vnfd
```

Figure 40: Collector NS Descriptor

Figure 40 depicts the content of the collector NSD, with a focus on the mapping between the virtual link identifier, ***provider***, and the VNF Descriptor identifier, ***cnit-collector_vnfd***.

The VNF Descriptor, identified by the ID ***cnit-collector_vnfd***, is contained into the VNF package, as shown in following Figure 41.

```
cnit-collector_vnf
├── cnit-collector_vnfd.yaml
└── juju-bundles
    ├── bundle.yaml
    └── charms
        └── cnit-collector
```

Figure 41: Collector VNF package

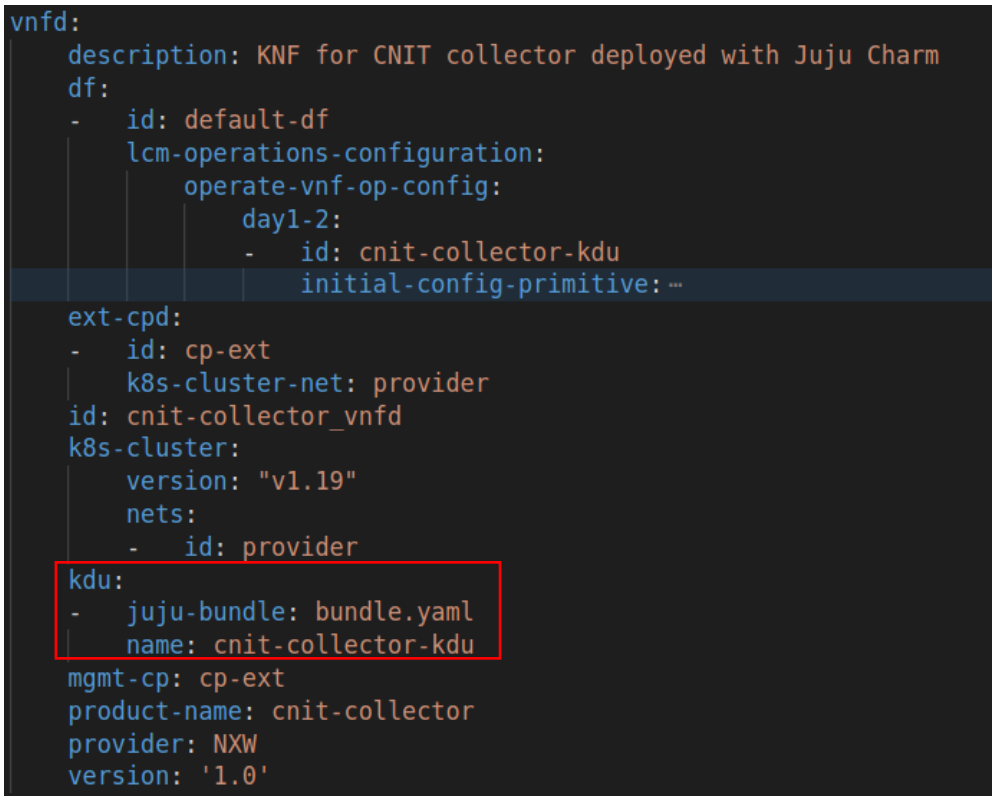

Figure 42: Collector VNF Descriptor

Figure 42 shows the content of the Collector VNF Descriptor with a focus on the ***kdu*** object that refers the ***cnit-collector*** charm within the bundle file.

A focus on the ***initial-config-primitive*** object is shown in Figure 43 and in the following description of the required parameters to be passed to the charm to execute the day-1 configurations.

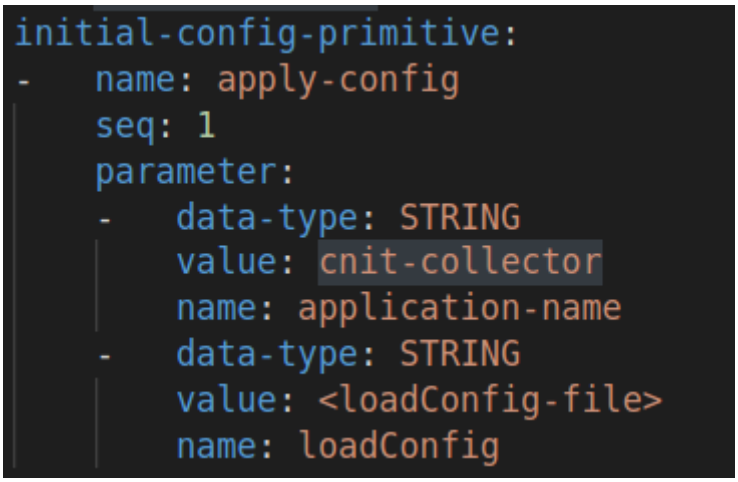

Figure 43: Collector initial config primitive parameters

- **application-name:** the value assigned to the application into the bundle file, i.e., cnit-collector;

- **loadConfig**: the JSON file that contains the configuration parameters required by the collector's function

To deploy the collector KDU, Juju Charm is used and the virtualized image required to implement this function is available on the LOCUS Container Registry.

The creation of the container that deploys the function is handled by the **on_config_changed** of the **cnit-collector** charm as shown in Figure 44.

```
def _on_config_changed(self, _):
    if not self.unit.is_leader():
        self.unit.status = ActiveStatus()
        return
    try:
        self.unit.status = MaintenanceStatus("Applying pod spec")
        # Initializing pod
        containers = [
            {
                "name": self.framework.model.app.name,
                "image": "registry.locus-project.eu/locus_uc1/cnit-si-localization-service:collector-v2",
                "ports": [
                    {
                        "name": "flask-port",
                        "containerPort": 5000,
                        "protocol": "TCP",
                    }
                ],
            }
        ]

        self.model.pod.set_spec({"version": 3, "containers": containers})

        self.unit.status = ActiveStatus()
        self.app.status = ActiveStatus()

    except OCIImageResourceError:
        self.unit.status = BlockedStatus("Error fetching image information")
        return
```

Figure 44: Collector's virtualized image specification

4.3.2 Processing Service

The processing service is composed of two different functions, parser and localization, respectively in charge of cleaning the data sent by the UE and calculating the positioning of the user based on SI technique.

The NS package contains, also in this case, only the processing NSD. The structure of the descriptor is shown in Figure 45 focusing the mapping between the virtual link identifier ***provider*** will be with the processing VNF Descriptor identifier, ***cnit-processing_vnfd.***

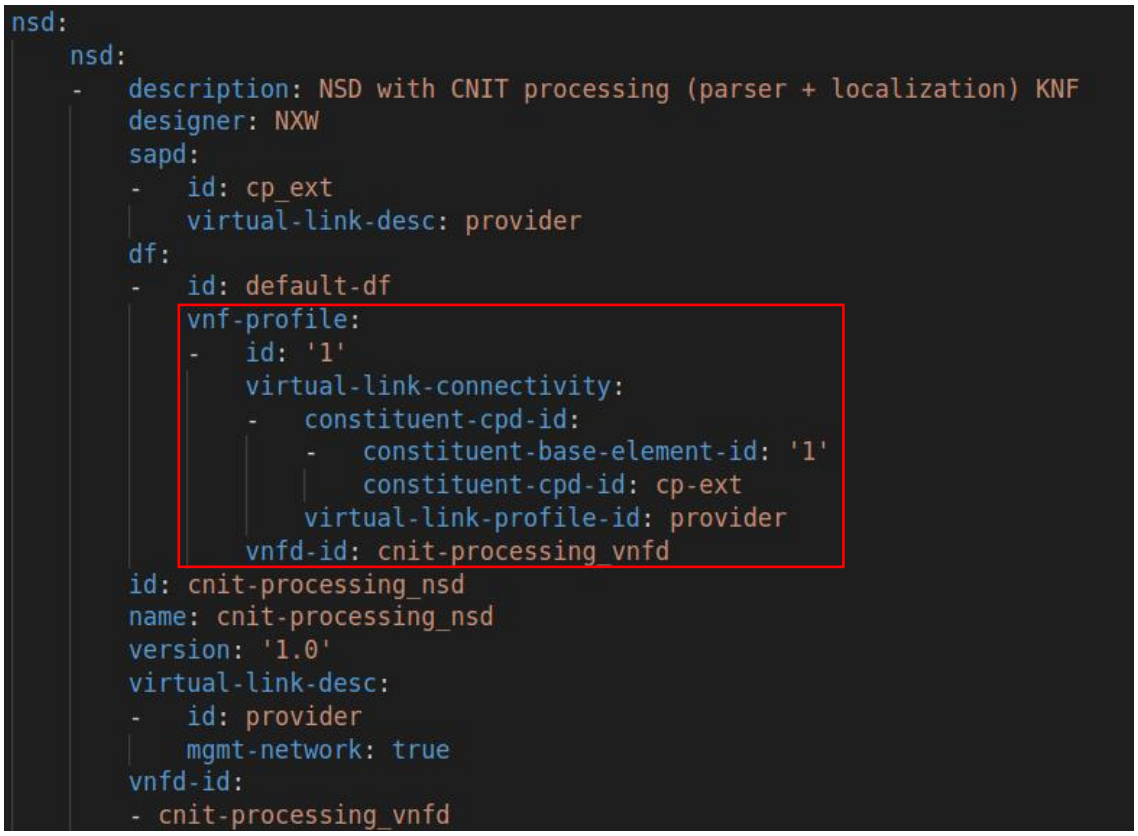

Figure 45: Processing NS Descriptor

Figure 46 depicts the content of the processing VNF package and as shown the charms directory contains two different charms, ***cnit-parser*** and ***cnit-localization***.

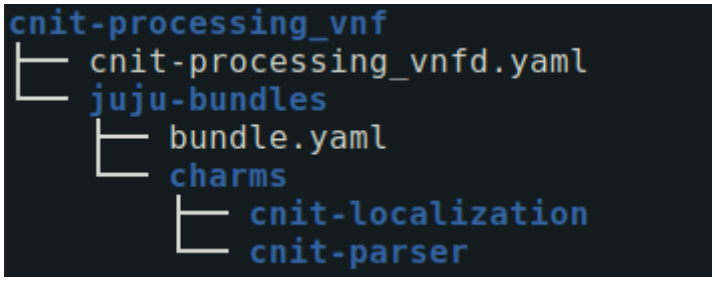

Figure 46: Processing VNF package

The VNF Descriptor within the processing VNF package follows the same baseline of the collector's one with the main difference represented by the day-1 configurations depicted in Figure 47 and Figure 48.

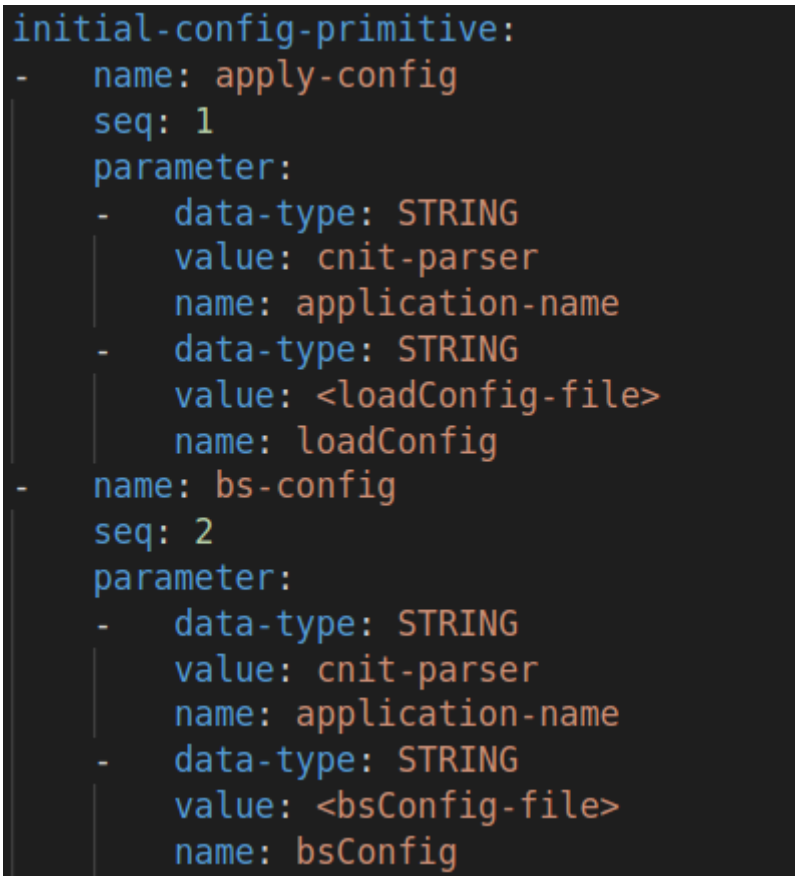

Figure 47: Parser initial config primitive

Figure 47 depicts the parameters required by the actions executed to configure the **cnit-parser** charm. The parameters of ***apply-config*** action are the same of the collector **loadConfig** that in this case will contain all the parameters required by the parser to be configured.

The ***bs-config*** action is required to configure the positions of the base stations and anchors used by SI-based localization algorithm. Following a description of the parameters.

- **application-name**: the value assigned to the application into the bundle file, i.e., cnit-parser;
- **bsConfig**: the JSON file that contains the configuration parameters for the base-station.

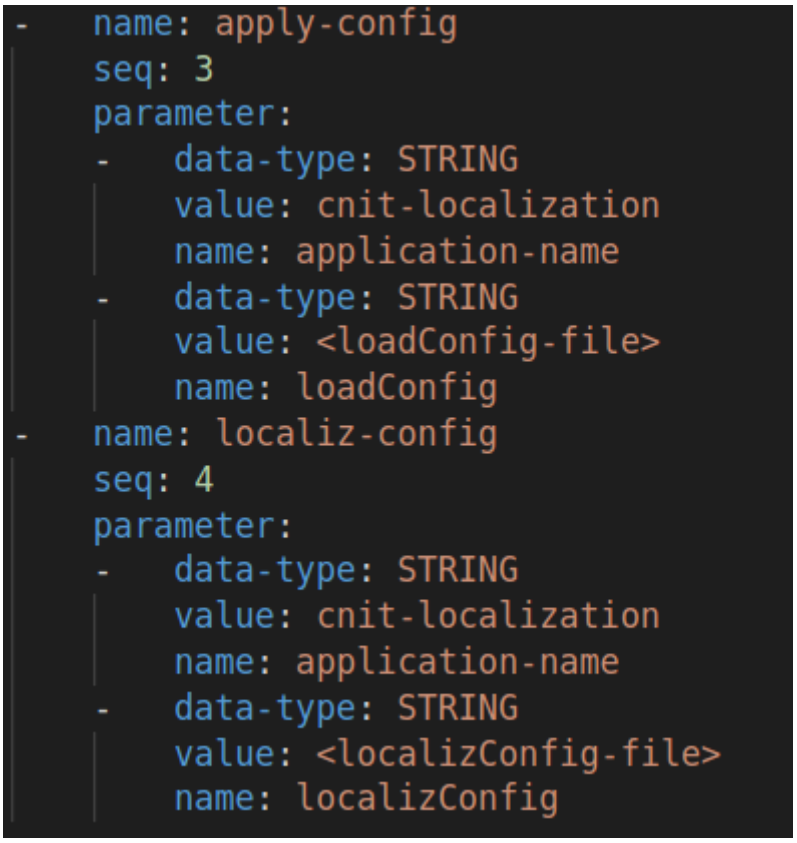

Figure 48: Localization initial config primitive

Figure 48 depicts the parameters required by the actions executed to configure the **cnit-localization** charm. The parameters of ***apply-config*** action are the same of the parser.

The **localiz-config** action is required to set the localization technologies and to define the boundaries of the device position. Following a description of the parameters.

- **application-name**: the value assigned to the application into the bundle file, i.e., cnit-localization;
- **localizConfig**: the JSON file that contains the configuration for the Localization function.

Similar to the collector, the creation of the containers that deploy the parser and localization functions is handled by the **on_config_changed** of the two charms, **cnit-parser** and **cnit-localization**, as shown in Figure 44 for the collector function. The containerized images of the parser and localization function are available in the LOCUS Container Registry at the following address:

- *registry.locus-project.eu/locus_uc1/cnit-si-localization-service:parser-v2*
- *registry.locus-project.eu/locus_uc1/cnit-si-localization-service:localization-v2*

5 Conclusions

The Virtualization Platform and the MANO framework represent two key aspects and building blocks in the LOCUS Platform. They enable the implementation of the localization analytics services platform, as they provide high degree of automation and flexibility in the management and operation of localization analytics functions in cloud-native virtualized environments.

This deliverable has presented their final prototype and integration in the project "production" testbed deployed at the OTE premises. Moreover, details on how LOCUS PoC functions and services are virtualized and made ready to be operated by the LOCUS MANO on top of the Virtualization Platform were provided.

The LOCUS Virtualization Platform is implemented following a hybrid approach to leverage on existing virtualization technologies and support distributed edge/core cloud-native deployments. This allows to match the requirements imposed by the 5G network architecture, which is based on a high degree of network function virtualization that can be deployed at different computing locations. The LOCUS Virtualization Platform is available at the OTE testbed, and integrates a Kubernetes cluster with an Openstack infrastructure, with the aim of supporting both traditional virtual machine-based services, as well as cloud-native applications more suitable to run at the edge as containerized services. A set of additional services for data distribution, software image repository and registry, datastores, and others, have been also integrated to fulfil the architecture and operation requirements imposed by the LOCUS Platform and localization analytics services.

On the other hand, the LOCUS MANO is implemented leveraging on the opensource platform ETSI OSM, which allows to support automated lifecycle management of the localization analytics functions and services in cloud-native environments. It fully exploits the NFV principles for the deployment and operation of VNFs and NFV Network, and provides automated configuration and runtime operation mechanisms for containerized functions which ease the integration of the localization analytics functions within other LOCUS platform services (e.g., data distribution, datastores, etc.). A monitoring platform has also been implemented as part of the LOCUS MANO to collect and visualize performance metrics of the deployed localization analytics functions and services, with special emphasis on those running as cloud-native Kubernetes containerized functions.

As part of the next steps, the work on the LOCUS Virtualization Platform and MANO will be continued in WP6, to support the various PoC activities in terms of localization analytics functions and services packaging, automated deployment and operation, and integration with other platform services, following the approach covered in this document.

References

[1] H2020 LOCUS Deliverable D4.3 “Implementation of the Virtualization platform for network control and management: preliminary version”

[2] Kubernetes, https://kubernetes.io/

[3] Openstack, https://www.openstack.org/

[4] H2020 LOCUS Deliverable D6.2 “Network management, network-assisted self-driving vehicles, people mobility and flow monitoring applications, integrated with geolocation mechanisms”

[5] Openstack Queens version, https://www.openstack.org/software/queens/

[6] Docker, https://www.docker.com/

[7] Rancher OS, https://rancher.com/

[8] Harbor, https://goharbor.io/

[9] ETSI OSM, https://osm.etsi.org/

[10] H2020 LOCUS Deliverable D2.5, “System Architecture: final version”

[11] RabbitMQ, https://www.rabbitmq.com/

[12] AMQP, https://www.amqp.org/

[13] MQTT, https://mqtt.org/

[14] STOMP, https://stomp.github.io/

[15] PostgreSQL, https://www.postgresql.org/

[16] MongoDB, https://www.mongodb.com/

[17] BIND 9, https://www.isc.org/bind/

[18] Apache2, https://httpd.apache.org/

[19] Helm Charts, https://helm.sh/docs/topics/charts/

[20] Prometheus, https://prometheus.io/

[21] Grafana, https://grafana.com/

[22] H2020 LOCUS Deliverable D5.1 “Design and implementation of virtualization technologies and pattern recognition mechanisms for physical analytics”

[23] Juju Charms, https://juju.is/

[24] Stateful deployments in Kubernetes, https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[25] ETSI GS NFV-SOL 006, “Network Functions Virtualisation (NFV) Release 3; Protocols and Data Models; NFV descriptors based on YANG Specification”, August 2020

[26] H2020 LOCUS Deliverable D3.1 “5G-based localization solutions: preliminary version”

[27] H2020 LOCUS Deliverable D3.2 “5G-based localization solutions: intermediate version”

[28] H2020 LOCUS Deliverable D3.3 “Integrated localization technologies: preliminary version”

[29] H2020 LOCUS Deliverable D3.4 “Integrated localization technologies: final version”

[30] H2020 LOCUS Deliverable D3.7 “5G-based localization solutions, final version”

[31] A. Conti et al., "Location Awareness in Beyond 5G Networks," IEEE Communications Magazine, vol. 59, no. 11, pp. 22-27, November 2021.

[32] A. Conti, S. Mazuelas, S. Bartoletti, W. C. Lindsey, and M. Z. Win, “Soft Information for Localization-of-Things,” Proc. IEEE, vol. 107, iss. 11, pp. 2240-2264, 2019.

[33] Consul, https://www.consul.io/