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 D5.4

"Prototype of the localization & analytics as a service solution"

Deliverable Type:	R/OTHER
Dissemination Level:	Public
Contractual Date of Delivery to the EU:	31/07/2022
Actual Date of Delivery to the EU:	04/08/2022
WP contributing to the Deliverable:	WP5 – Localization & Analytics for New Services
Editor(s):	**Nextworks**, Giacomo Bernini
Author(s):	**Nextworks**, Michael De Angelis, Nicola Venturi, Giacomo Bernini **Incelligent**, Athina Ropodi, Aristotelis Margaris, Yannis Filippas, Kostas Tsagkaris **VIAVI**, Takai Eddine Kennouche **IBM**, Faisal Ghaffar **CNIT**, Flavio Morselli, Andrea Conti, Gianluca Torsoli

	NEC, Giuseppe Siracusano, Gurkan Solmaz
Internal Reviewer(s):	**CNIT**, Stefania Bartoletti, Flavio Morselli, Nicola Blefari Melazzi
	Samsung, Tomasz Mach
Short Abstract:	This deliverable describes the software prototype of the LOCUS localization analytics as a service solution. In particular, the document details on the implementation of the LOCUS API layer functionalities and the LOCUS platform control components. These enable the execution and exposure of agile analytics services and pipelines in support of external smart network management and vertical applications. The report also describes the technologies selected (and related prototypes) for the LOCUS data platform, which enable data stream exchange and data persistence within the platform in support of the LOCUS virtualized analytics services.
Keyword List:	Analytics as a Service, API gateway, Kubernetes, Localization, Machine Learning, Virtualization

Executive Summary

LOCUS provides a unified and generalized platform for the deployment of localization analytics functions and services, and their exposure towards Smart Network Management and 3rd party vertical applications that require (geo)location-awareness and analytics for their purposes. More specifically, LOCUS implements a localization analytics as a service solution on top of a flexible, scalable and virtualized infrastructure that allows to deploy and execute analytics services and functions as virtualized elements across edge and core compute locations of the 5G network.

This deliverable provides implementation details of the LOCUS localization analytics as a service solution in the form of a software prototype, which is built by two main macro-components, i.e., the LOCUS API layer and the LOCUS platform control. The LOCUS API layer represents the access point of the LOCUS platform, and it offers a flexible and open northbound interface to interact with the localization, analytics and ML functions available in the whole platform. External applications (such those for third party vertical applications or smart network management) and users can consume analytics services data on-demand, without considering the complexity of the analytics service deployment, configuration and operation. On the other hand, the LOCUS platform control complements the API layer functionalities and provides a coordinated control of data operations, while acting as bridge towards the LOCUS Management and Orchestration for the automated and on-demand deployment of functions and services as virtualized elements.

The LOCUS localization analytics as a service solution prototype reported in this deliverable is based on the specifications provided in D5.3 for the solution approach, as well as for the identification of main components and workflow operations. On the other hand, the overall LOCUS platform system architecture and (above all) technology selection scouting carried out in D2.5 also provides a key reference baseline. Indeed, the LOCUS localization analytics as a service solution prototype relies on existing opensource technologies. In particular, analytics APIs exposure functionalities are based on Consul, Zuul, and Swagger among others, while support of analytics service and Machine Learning management and virtualization rely on Apache Airflow and Kubeflow.

The deliverable also reports on the LOCUS data platform software prototype, which is implemented as a combination of opensource tools which enables both real-time data streams exchange (i.e., RabbitMQ) and data persistence (i.e., Hadoop, Hive, Trino) in support of the communications among the analytics service components and functions.

VERSION CONTROL TABLE			
VERSION N.	**PURPOSE/CHANGES**	**AUTHOR (S)**	**DATE**
0.0	First draft of ToC	NXW	30/05/2022
0.1	LOCUS API layer, LOCUS Platform control	INCE	24/6/2022
0.2	SI-based localization service	CNIT	27/6/2022
0.3	LOCUS Platform control	NXW	1/7/2022
0.4	LOCUS Data Platform	VIAVI, INCE	6/7/2022
0.5	Transportation UC	INCE	8/7/2022
0.6	Mobility patterns UC	IBM	11/7/2022
0.7	ML Pipeline Optimization, Crowd mobility UC	NEC	15/7/2022
0.8	LOCUS API layer	NXW	19/7/2022
0.9	Final version for the internal review	NXW	25/07/2022
1.0	Final version for EC submission	NXW	02/08/2022
1.1	Final revision by the coordinator	CNIT	04/08/2022

Table Of Contents

List of Abbreviations

ABBREVIATION	FULL NAME
5G	Fifth generation technology standard for cellular networks
AI	Artificial Intelligence
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DAG	Directed Acyclic Graph
ETSI	European Telecommunications Standards Institute
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
LEN	Localization Enabler
ML	Machine Learning
NFV	Network Function Virtualization
NSE	New Services
MANO	Management and Orchestration
POI	Point of Interest
REST	REpresentational State Transfer
RSSI	Received Signal Strength Indicator
RSRP	Reference Signal Received Power
RSRQ	Reference Signal Received Quality
SDK	Software Development Kit
SNM	Smart Network Management
SSO	Single Sign-On
UI	User Interface
UC	Use Case
UE	User Equipment

VNF	Virtual Network Function
VM	Virtual Machine
WP	Work Package
WP5	Work Package 5

Figure Index

Table Index

1 Introduction

1.1 Document scope and objectives

This deliverable presents the latest and final results of Task T5.3 “Virtualized platform for localization & analytics as a service” activities. As part of the T5.3 objectives, the LOCUS API layer as well as the LOCUS analytics services control functionalities have been implemented and deployed in the project testbed available at OTE premises.

Specifically, the API layer and platform control functionalities represents two key assets within the LOCUS platform. They provide an advanced and innovative integrated solution for the execution and exposure of localization and analytics services running on top of virtualized infrastructures integrated in the 5G network.

In particular, this deliverable reports on the software prototype details of the different components and modules providing the LOCUS API layer and the LOCUS platform control functionalities, together with their deployment and integration in the LOCUS testbed provided by OTE (which is described in D6.2 [51]).

The work reported in this deliverable is based on the design and specifications provided in D5.3 [1], where the LOCUS localization analytics as a service paradigm has been formulated and formalized, introducing the concept of an API layer. On the one hand, this API layer is responsible to provide access to the virtualized analytics functions, Machine Learning (ML) pipeline services and ML model predictions when they run in the LOCUS edge/core virtualized infrastructure. On the other hand, it exposes the data they generate as services that can be consumed by external applications (e.g., Smart Network Management and 3rd party vertical applications). In addition, dedicated data operations control functionalities are integrated with this proposed localization analytics as a service solution to coordinate the execution of both analytics functions and services pipelines. This is implemented through the pipeline orchestration functionalities that decouple the API layer exposure and data consumption from the internal analytics service details and virtualization aspects (thus including its deployment and management through the LOCUS Management and Orchestration - MANO).

While D5.3 [1] provided the specification of the localization analytics as a service solution with the related functional decomposition, D2.5 [2] as the final LOCUS architecture deliverable described how all these functionalities are integrated into the overall platform at the system level. In summary, with reference to the LOCUS system architecture depicted in Figure 1, the work described in this document covers the implementation of: (i) the LOCUS API blocks; (ii) the LOCUS platform control blocks; and (iii) the LOCUS data platform components. the LOCUS data platform components refer to the message queue and persistence modules which enable the various localization analytics functions and service to exchange positioning and analytics

data (as specified in D5.3 [1] and D2.5 [2]) and thus facilitate the exposure of analytics services results towards the external applications.

In addition to the pure software prototype description for the LOCUS API blocks, the LOCUS platform control blocks, and the LOCUS data platform components, this deliverable describes the changes related to the LOCUS virtualized analytics services implementation, following up what was designed and preliminary developed as part of D5.1 [3], D5.3 [1] and D5.2 [4] for the WP5 use cases (NSE-UCs). In this context, this deliverable describes a set of implemented prototypes which are candidates for integration in the final LOCUS Proof-of-Concepts (PoCs) that will be delivered with D6.3 [5].

Despite this deliverable represents the final outcome of task T5.3, it is worth highlighting that the activities related to the LOCUS platform control blocks are expected to be continued in the context of the final project sprint for PoCs developments, integration and demonstration in WP6. This is planned to be realized in the form of support for integrating the final PoC functionalities within the LOCUS platform and enable a comprehensive implementation and demonstration of the localization analytics as a service solution.

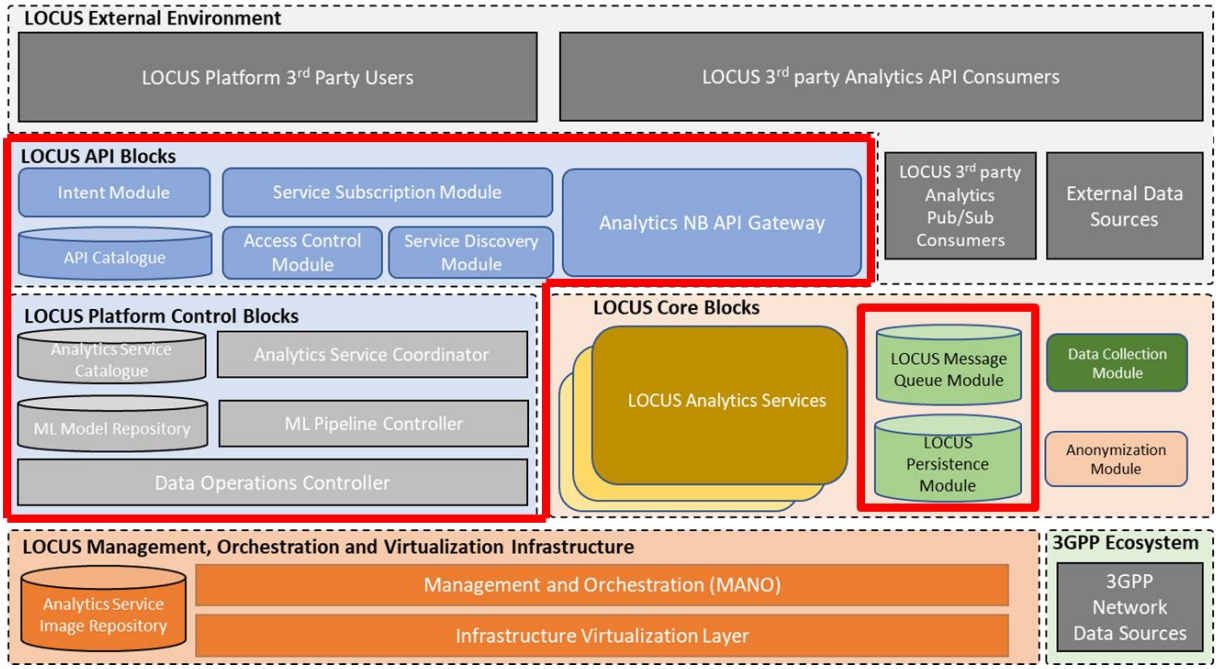


Figure 1: LOCUS System Architecture (ref. D2.5 [2])

1.2 Document structure

The rest of this deliverable is structured into four main sections:

- Section 2 describes the software prototype that has been implemented to realize the functionalities of the LOCUS localization analytics as a service solution. In particular, the section reports on the implemented components building the LOCUS API layer and

on those providing the LOCUS platform control. Details about their deployment and integration in the OTE testbed are also provided;

- Section 3 describes the LOCUS Data Platform prototype, which is composed of the message queue module for real-time data streams exchange within the LOCUS platform, and the persistence module. Both of them are deployed and integrated in the OTE testbed;
- Section 4 provides updates on the implementation of the LOCUS analytics service (mapped to the WP5 NSE-UCs), following up what was reported in previous deliverables (mostly D5.3 [1] and D5.2 [4]). Specifically, these virtualized analytics service prototypes are candidates to be integrated in the LOCUS PoCs and the related final demonstrations;
- Section 5 provides concluding remarks for this document.

2 LOCUS localization analytics as a service prototype

This section describes how the LOCUS localization analytics as a service solution specified in D5.3 [1] and D2.5 [2] has been developed in the form of a software prototype integrating several components and modules.

Figure 2 shows the LOCUS system architecture, with an highlight of the reference technologies selected to implement the various functional components and modules. The localization analytics as a service solution is implemented by the integration of the API layer (for exposing the available services and enabling their consumption) with the platform control (for taking care of the internal analytics service data operations control). They rely on the LOCUS MANO to automate the deployment of the LOCUS analytics services as virtualized services (and thus mapped to NFV Network Services).

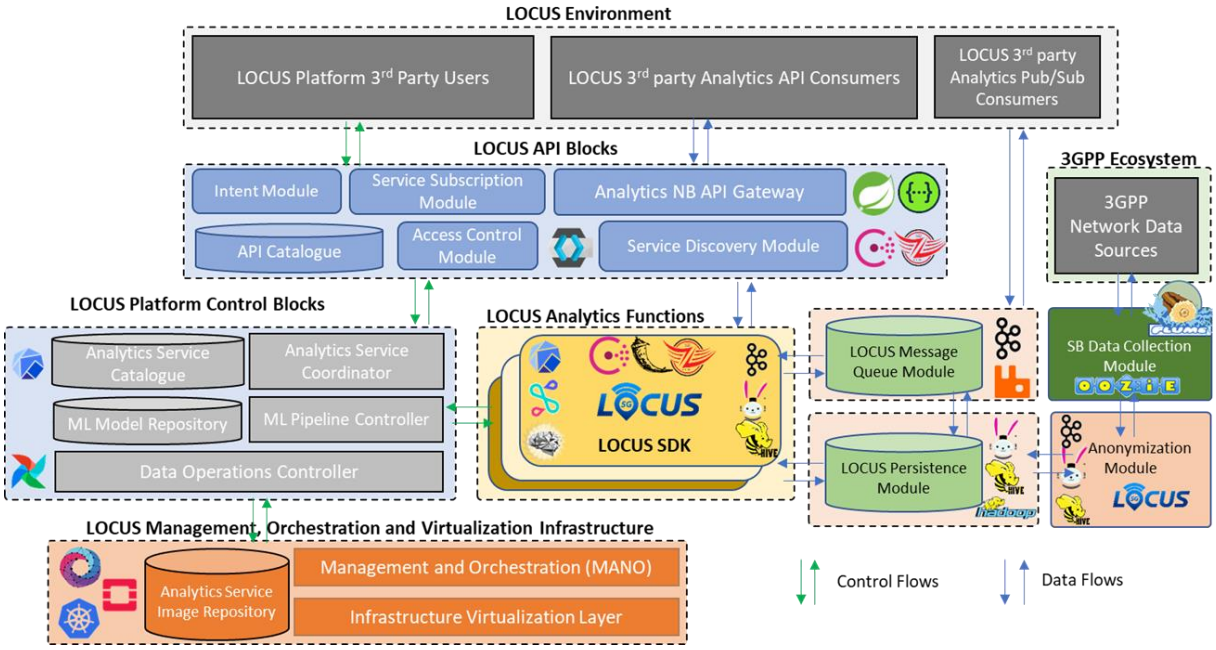


Figure 2: LOCUS system architecture functional blocks – technology selection (ref. D2.5 [2])

The LOCUS API layer represents the northbound interface of the LOCUS platform, and it offers a flexible and open interaction with the localization, analytics and ML functions available in the whole platform. In particular, it enables the external applications and users to consume localization analytics and ML predictions on-demand, while interacting with the LOCUS MANO and virtualization platform for automated deployment and operation of the various offered services. With the combination of the LOCUS platform control, it is possible to decouple the exposure of the analytics services from the internal analytics service data operations constraints and complexities. This includes not exposing the details of the implementation of the offered analytics services as NFV Network Services and VNFs. As a result, the external

applications and users can focus on querying the available analytics services and consuming the produced analytics data and ML model predictions that suites their requirements.

From an implementation perspective, most of the components and modules identified in D2.5 [2] have been developed and deployed in the OTE testbed, where the LOCUS virtualization platform is realized. Table 1 and Figure 3 detail how the LOCUS API layer and LOCUS platform components have been deployed as services in the OTE testbed available Virtual Machines (VMs).

Table 1: LOCUS API layer and platform control services in the OTE testbed

Testbed VM	**IP**	**LOCUS component**
API Layer VM	172.16.12.53	Access Control (*port: 9080*) API Gateway (*port: 8080*) API Catalogue (port: *8090*) Service Discovery (port: *8500*)
Platform Control VM	172.16.12.46	Analytics Service Coordinator (*port: 9090*) Data Operations Controller (*port: 8088*)
Kubernetes (master node)	172.16.12.65	ML Pipeline Control (*port: 31080*)

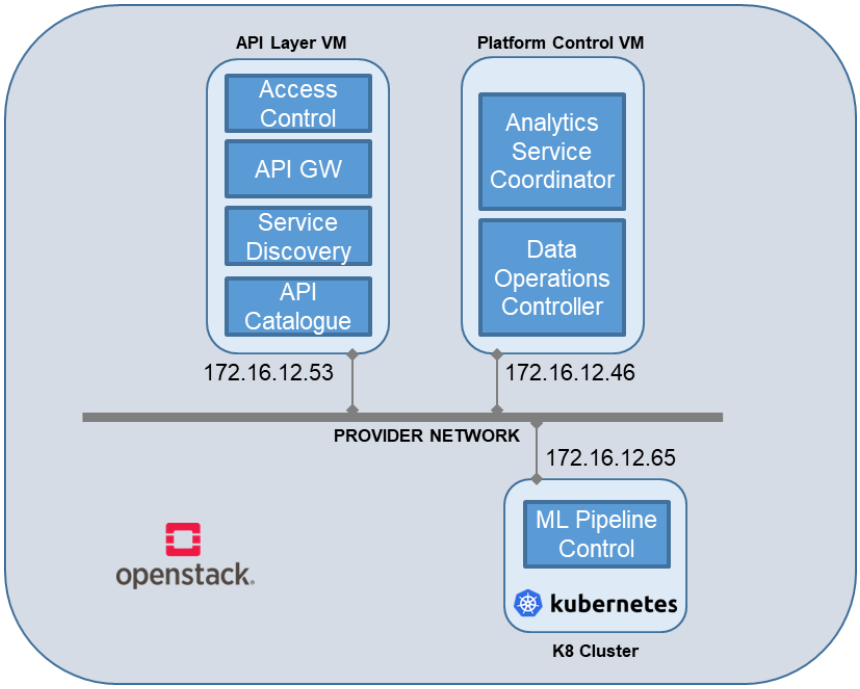


Figure 3: Deployment of the LOCUS API layer and platform control services

On the other hand, as anticipated in D5.3 [1], the intent functionalities for intent-based localization analytics as a service is left at design concept level, thus the Intent Module shown in Figure 2 is not delivered as part of the software prototype described in this document.

The rest of this section is organized in two main sub-sections, one dedicated to the LOCUS API layer prototype, and the other to the LOCUS platform control prototype.

2.1 LOCUS API layer prototype

The LOCUS API layer has been implemented as the integration and combination of several components and modules, following D5.3 [1] and D2.5 [2] specifications and system level requirements, which resulted into overall decomposition and mapping to reference technologies shown in Figure 4.

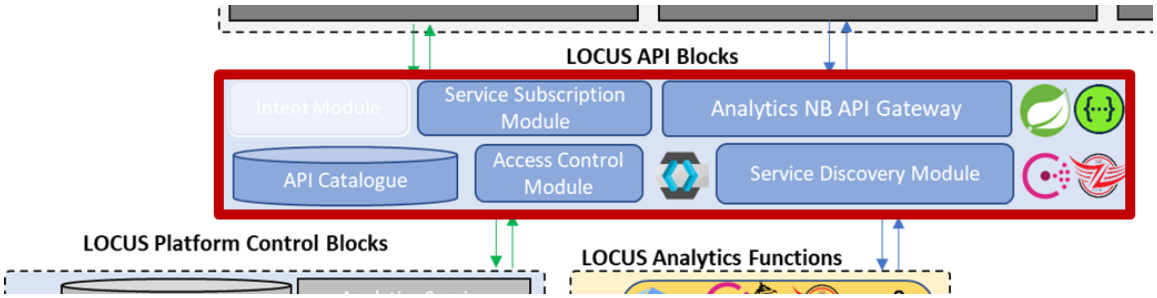


Figure 4: LOCUS API layer software prototype components

Figure 4 highlights the LOCUS API layer software prototype components. As main northbound interface of the LOCUS platform. It allows to discover the available analytics services, that can be retrieved by external applications and users through the API Catalogue. Thanks to the automation in the LOCUS analytics services deployment and operation, the API layer also enables these external entities to subscribe to the catalogued services and either directly consume the related data (if the service is already activated and running in the platform) or trigger its deployment and later consume the data. This has been developed according to the API layer capabilities and workflows described in D5.3 [1]. Specifically, the implemented API layer separates the analytics service exposure and access (to/from external applications and users) from the LOCUS platform internal logics for service data operations and automated deployment of virtualized services and functions on top of the virtualization platform.

The following subsections provide details about each component of the LOCUS API layer software prototype.

2.1.1 Access Control

In order to allow access to the LOCUS Platform analytics, security aspects must be taken into account, i.e. permitting access based on the platform user profile/access rights. The Access Control module allows for:

- user generation, administration and assignments rights
- enforcement of credential verification or API key-based access on all the internal, sensitive resources.

Single sign-on (SSO) is an authentication process that enables users to securely authenticate for continuous use of multiple independent and/or related systems/services. In this context and based on the technology scouting taken place during the design phase, Keycloak [6] was selected as an open-source authentication and identity checking provider following the SSO-architecture. Figure 5 displays the Keycloak homepage, in which -though the administration console- users and groups and relevant access rights (among others) can be configured. More specifically, in Figure 6a "locus" realm is set up, i.e. an object including a set of users along with their credentials, roles and groups. Further details on users and groups can be set through the menu on the left side.

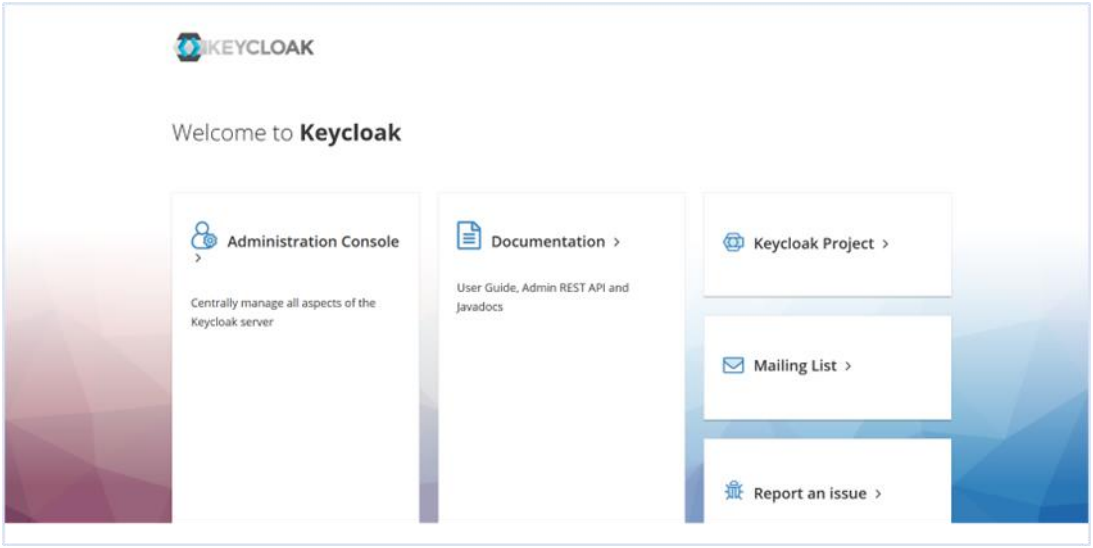


Figure 5: LOCUS Access Control Module - Homepage

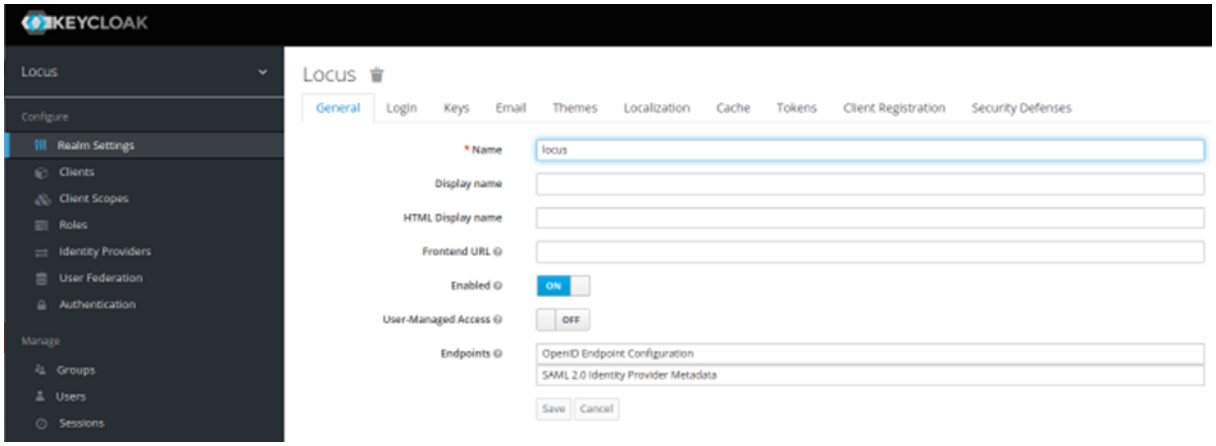


Figure 6: LOCUS Access Control Module – Setting up the "locus" realm

It should be noted that the enforcement of the authentication and authorization rules of the platform is done via the LOCUS API Gateway (see also section 2.1.3) through its integration with the Keycloak system. More specifically, Java Spring [7] and Python Flask [8] applications (technologies used for development/deployment of services) can use the Keycloak third party system as a single source of authentication. Furthermore, Spring security [9], i.e. an extension on the Sprint REST framework (utilized for the API Gateway purposes), allows for implementation of filtering and security rules that is based on the Keycloak rules.

The Access Control module implemented through Keycloak is deployed and integrated in the OTE testbed infrastructure, available and accessible in a common LOCUS API layer VM at:

IP: 172.16.12.53

Port: 9080

2.1.2 API Catalogue and Service Subscription

The API catalogue is the actual front-end component of the LOCUS API layer for what concerns accessing the capabilities offered by the platform in terms available analytics services. Indeed, it maintains the list of available localization analytics services that can be either activated on-demand in the platform and then consumed through the API gateway (see section 2.1.2), or just consumed in the case the service is already deployed and available in the platform. In particular, these two possible approaches allow for both pre-scheduled (including platform admin-based execution of services) and on-demand localization analytics services. Following the D5.3 [1] and D2.5 [2] specifications, the API catalogue stores and exposes information about the available localization analytics services, with additional metadata that allows to identify the main characteristics of the service, including type of analytics data offered as well as format and schema of the data, and how to consume it (i.e. from which endpoints exposed by the API gateway).

From a software prototype perspective, it is implemented through a postgreSQL database [10], which is intended to maintain all the analytics service information required by external applications and users. In particular, for each analytics service available in the LOCUS platform (e.g., Flow Monitoring Service, Transportation Optimization Service, etc.), a dedicated entry in the database is kept following the information model specified in D5.3 [1] (including service ID, type, name, output data format, etc.).

While this database service represents the back-end of the API catalogue, the front-end is implemented as a custom Python application (i.e., developed from scratch for LOCUS) that provides access to the database. Specifically, this front-end application provides a set of REST APIs and enable Create Read Update Delete (CRUD) operations on the API catalogue database. The Python FastAPI framework [10] has been used in order to have native integration with the Keycloak framework [6] implementing access control in the LOCUS platform, as well as for native integration with Swagger tools [12] and related API documentations.

Beyond the actual CRUD operations on the content of the API catalogue, the front-end application also takes care to manage the external subscriptions to activate and consume the analytics services, thus offering the related REST APIs. For this reason, the API catalogue and service subscription components depicted in Figure 4 are actually developed as a single module, embedding the database service and its front-end for querying the catalogue content, and at the same time exposing dedicated REST APIs allowing external applications and users to subscribe to LOCUS analytics services.

Based on this, the LOCUS API layer software prototype follows the high-level design depicted in Figure 7. As shown in the figure, the subscription engine implemented in the front-end application is responsible for the interaction. This is done by invoking the Analytics Service Coordinator within the LOCUS control platform by using the appropriate “serviceID” attribute (please refer to D5.3 [1]), which is known to both components as unique identifier of the given LOCUS analytics service.

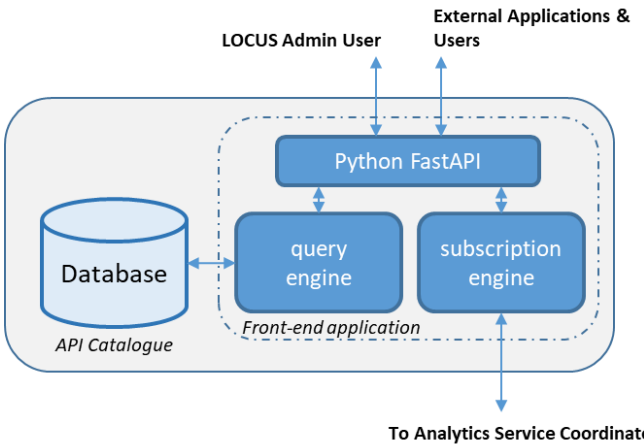


Figure 7: API Catalogue (and Subscription Service) software prototype

As said the API catalogue and service subscription software prototype exposes a set of REST APIs to interact with the LOCUS platform (as kind of platform northbound interface). Figure 8 shows these APIs in the form of a Swagger UI, as a result of the automated generation of API documentation from the Python FastAPI front-end. These are classified in: Service Discovery APIs to query the content of the API catalogue (and related available services in the platform), Service Management APIs to manage the content of the API catalogue, Service Subscription APIs to activate and subscribe to new services to be deployed in the platform (as per D5.3 [1]).

The API catalogue (integrated with the service subscription) is deployed and integrated in the OTE testbed infrastructure, available and accessible in a common LOCUS API layer VM at:

IP: 172.16.12.53

Port: 9090

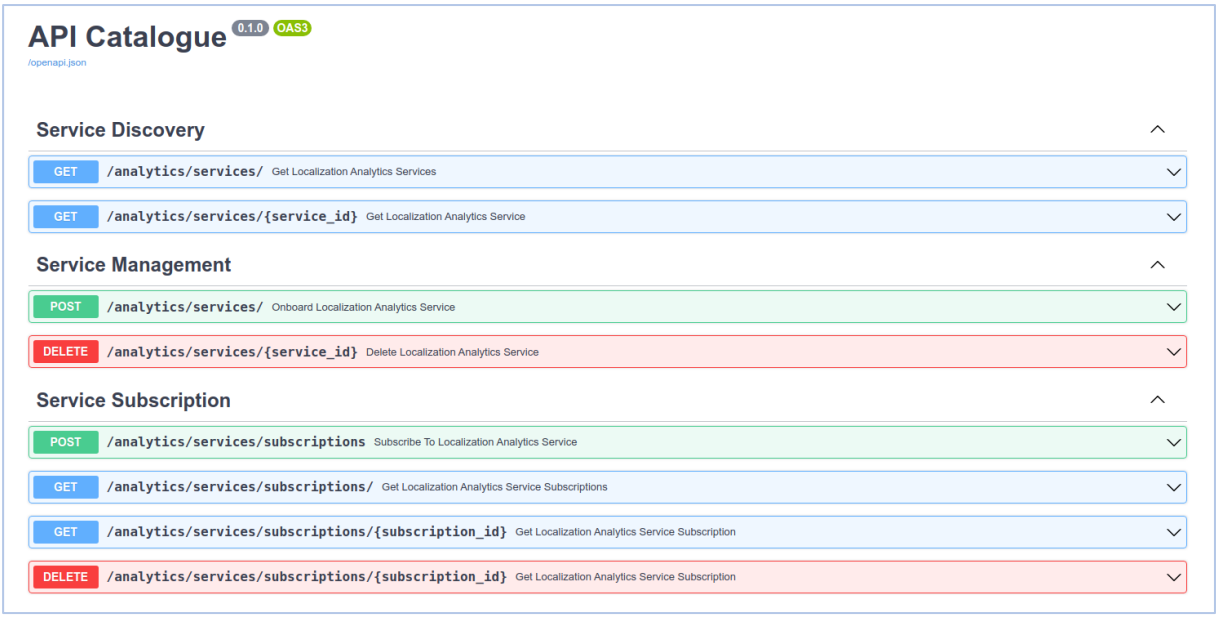


Figure 8: API Catalogue (and Service Subscription) exposed REST APIs

It is worth to mention that, as specified in D5.3 [1] (with the related workflow), the API Catalogue content is populated through offline onboarding procedures performed by the LOCUS platform manager/administrator. In practice, the Service Management APIs shown in Figure 8 are visible and usable by platform managers/administrators only, while all the others are exposed and offered to the external applications and users.

2.1.3 Analytics Northbound API Gateway

The Analytics Northbound (NB) API Gateway is the block that allows exposing the LOCUS services to 3rd parties and external applications in general, (see D2.5 [2]), based on the microservices principle acting as a system that combines all the analytics resources in one

place. It connects external applications to the LOCUS environment with the analytics results that are generated and stored in the LOCUS Platform via reverse-proxy communication. Furthermore, the gateway system includes load balancing capabilities, as well as https-based (Transport layer Security - TLS) transport protocols to ensure standard internet security.

In addition to the above, though its communication with the Access Control module (see 2.1.1), user access control is enforced.

The NB API Gateway is developed using the LOCUS Software Development Kit (SDK) (described in D2.5 [2]), i.e. as part of the Java Spring framework [7] which enables the use with the Spring-Swagger integration [12]. This generates automatically a front-end for building REST requests and parsing/ formatting the responses in a way that web developers are accustomed to work and build applications with. More specifically, Spring Boot is a Java-based web framework for the development of production-ready REST applications. Since it is flexible, scalable, light-weight and includes security features for the access control, it can be used for the deployment of a public API gateway using also extensions/ integration libraries of the Spring ecosystem. More specifically, it allows for seamless integration with the Swagger REST interface which automatically generates meta-data from the LOCUS analytics functions acting as a user manual, input and output data type communication and testing the operation of the public API. To enable automated exposure of runtime analytics and data APIs (i.e. those exposed by analytics service and functions deployed in the platform as virtualized elements), the API gateway also integrates Zuul [13], an opensource gateway service that provides dynamic routing and together with the Service Discovery (see section 2.1.4) allows to dynamically retrieve API documentations metadata from the various registered APIs and thus expose at runtime only the actual available analytics and data APIs.

Figure 9 presents an exemplary screenshot of the designed gateway where in green some direct analytics APIs for the LOCUS PoC#3 are presented. In blue, the APIs that enable authentication via the communication with the Access Control module are also presented.

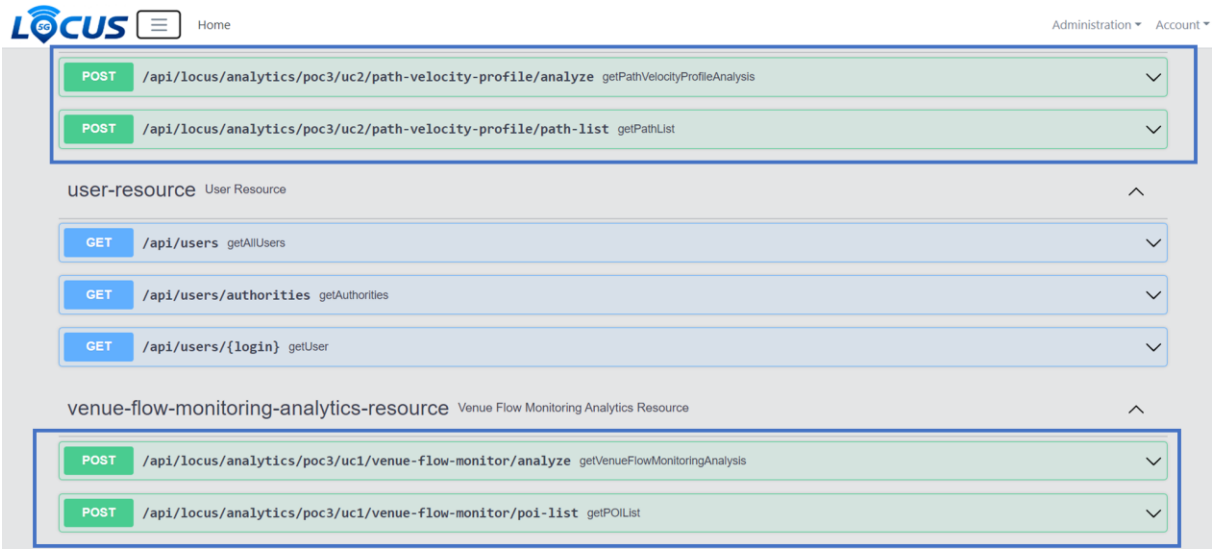


Figure 9: LOCUS NB API Gateway – Exemplary list of direct analytics/services APIs for PoC#3 (in green) and authentication APIs (in blue)

Similarly, an exemplary set of data consumption APIs from the LOCUS PoC#3 are presented in Figure 10. Lastly, in Figure 11, an exemplary API description for an API as shown in the gateway UI is presented.

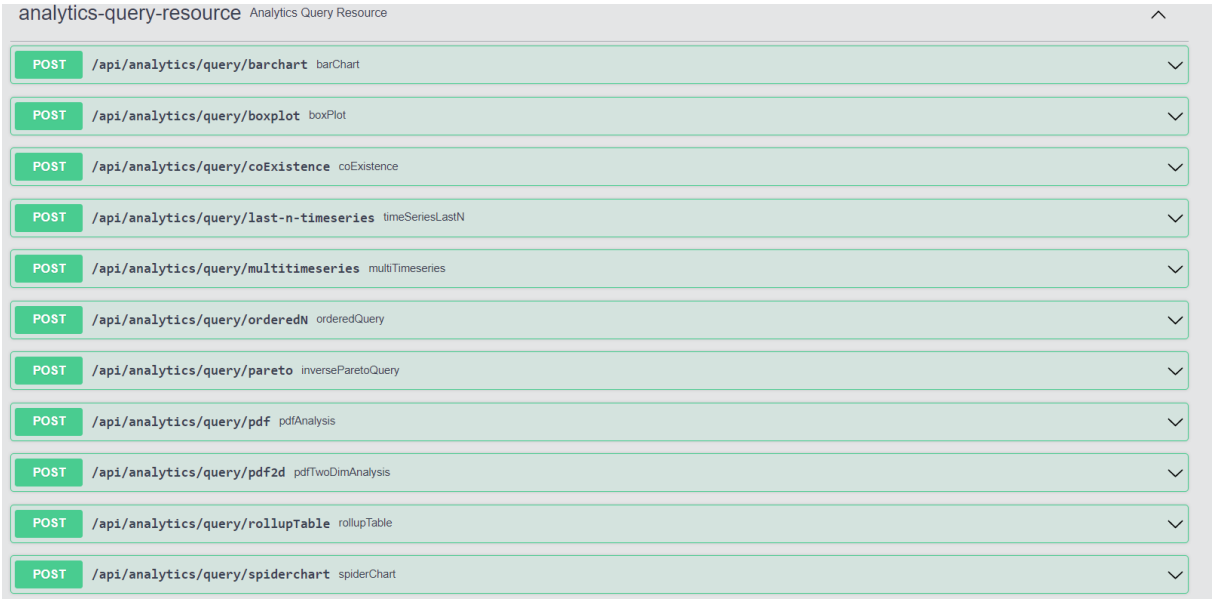


Figure 10: LOCUS NB API Gateway – Exemplary list of data APIs for PoC#3

The API gateway is deployed and integrated in the OTE testbed infrastructure, available and accessible in a common LOCUS API layer VM at:

IP: 172.16.12.53

Port: 8080

2.1.4 Service Discovery

As designed in D2.5 [2] as part of the LOCUS system architecture, and first presented in the D4.4 [14] dedicated to the Virtualization Platform, the Service Discovery Module is part of the LOCUS APIs block (realizing the LOCUS API layer functions).

The platform requires a logically centralized component that stores all the available analytics services and functions that expose analytics and data APIs to be consumed, mapping them to a specific service name. Thus, the Service Discovery module provides a centralized repository for microservices to register their IP/ port pairs, avoiding issues with the static IP/ port configuration and the dynamic spawning of containers that occur when Kubernetes [15] is used as cloud-native solution to execute the LOCUS analytics functions and services. In detail, the internal analytics function instances are generated on demand, allowing for flexibility and scalability, and their reachability information is registered through a naming convention.

This way:

a) The internal functions can be accessed by client libraries through their network addresses for direct communication within the LOCUS analytics functions internal network;

b) The Service Discovery Module keeps track of the service's status, links the external API and the internal services, providing information to users on the status of the requested API access.

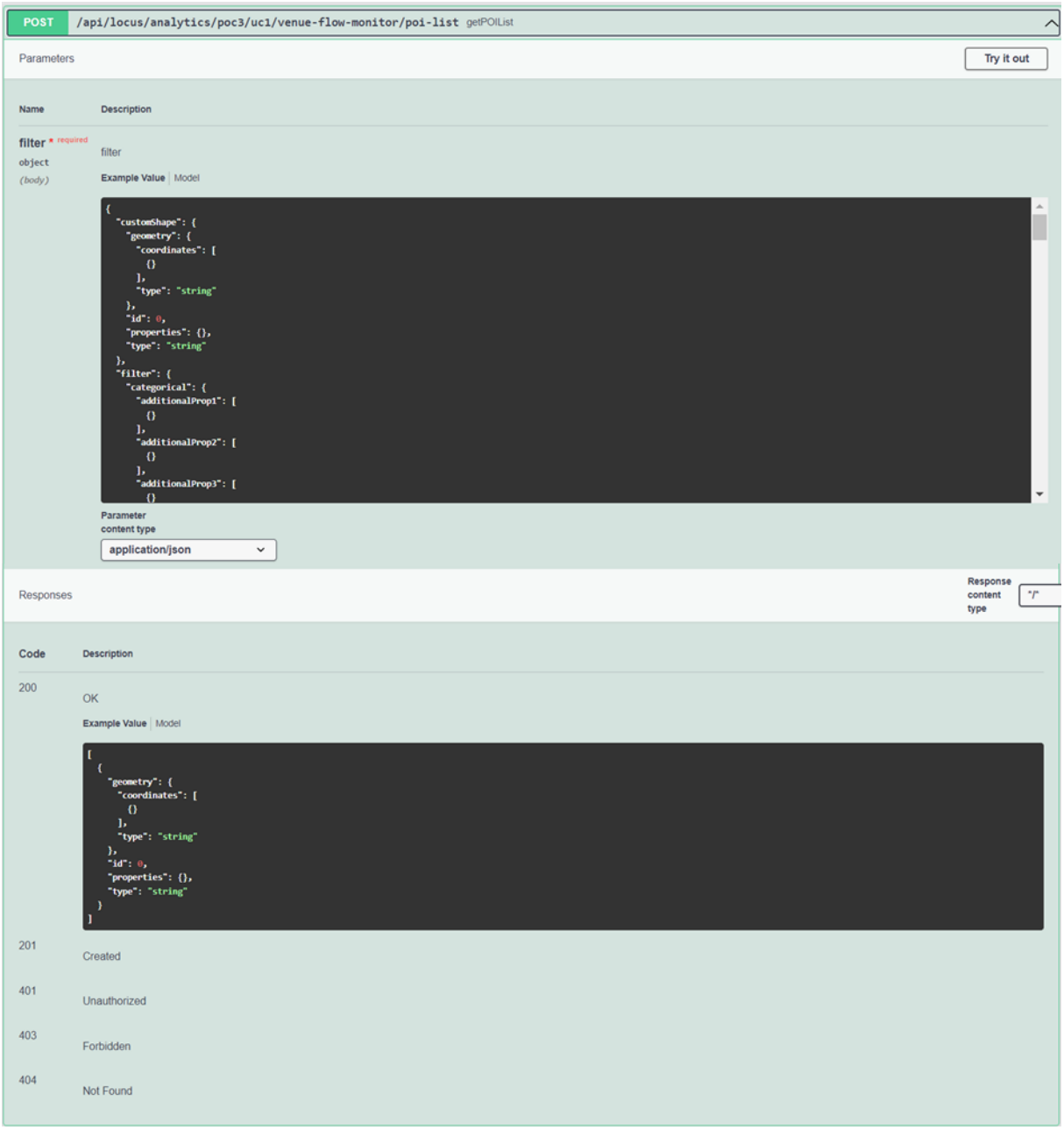


Figure 11: LOCUS NB API Gateway – Exemplary API description for a direct analytics API (top) and the response (bottom)

The aforementioned module is currently deployed successfully in the OTE testbed for the purposes of the final demonstrations. Furthermore, it has been a part of the LOCUS Project demonstration booth in the 2022 EuCNC & 6G Summit conference [16].

It is based on the open-source technology Consul [17] (which is also part of the initial technology scouting that took place in Deliverable 2.5 [2]) allowing for REST service discovery functions, such as service registration and service network address reverse lookup.

Consul offers a common REST Service registry that helps with the physical discovery of existing instantiated LOCUS functions. Figure 12 presents such a service overview through the Consul

user interface (UI) in which a list of registered services is presented for the case of the LOCUS PoC#3 demonstration.

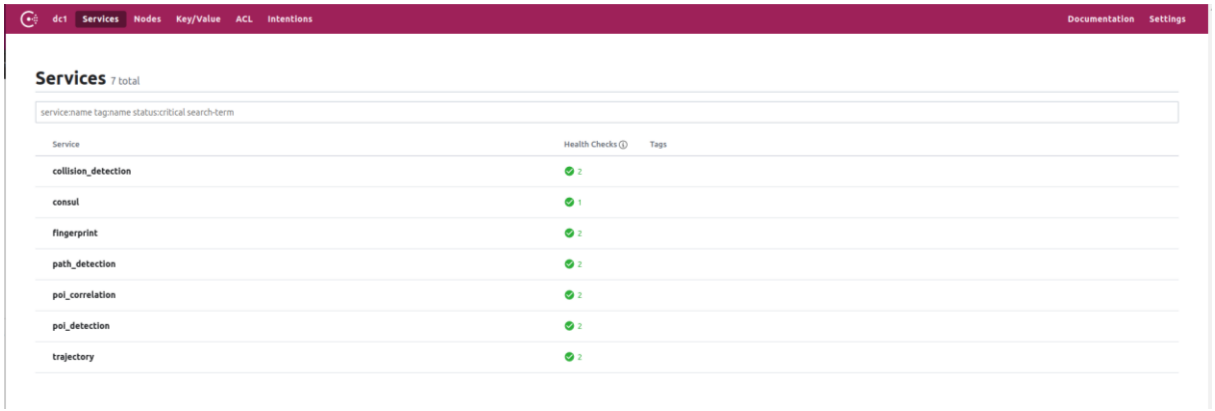


Figure 12: LOCUS Service Discovery module – Indicative list of services

Additionally, each function contains its own health checks in order to check its operational status, as shown in Figure 13 for the case of a Localization Enabler (LEN) function called "fingerprint" utilized in LOCUS PoC#3. Lastly, it should be noted that the Service Discovery module is then utilized in the Data Operations Controller (see 2.2.3) to find the location of each service and invoke its appropriate action.

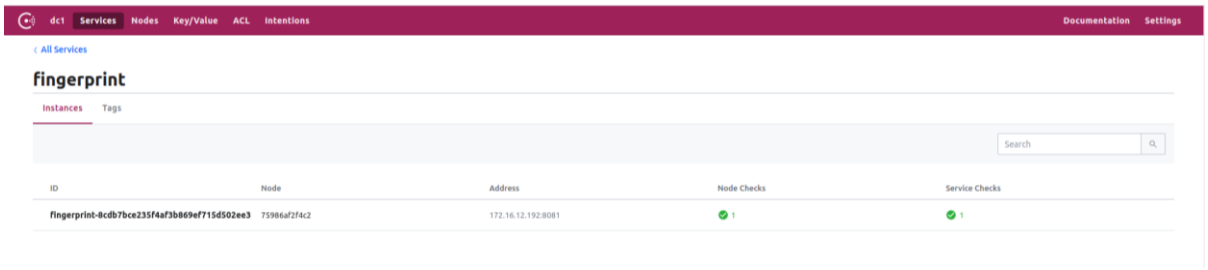


Figure 13: LOCUS Service Discovery module – Operational status/ health check via Consul UI

The Service Discovery implemented through Consul is deployed and integrated in the OTE testbed infrastructure, available and accessible in a common LOCUS platform VM at:

IP: 172.16.12.53

Port: 8500

2.2 LOCUS Platform control prototype

The LOCUS platform control represents a key functional macro-block within the LOCUS platform, as it enables the various data operations involving the various functions that compose the analytics service and pipelines. In particular, it provides a coordinated control of data operations, while acting as a bridge towards the LOCUS MANO for the automated and on-demand deployment of functions and services as virtualized elements.

As for the API layer, the LOCUS platform control has been implemented as the combination of different components, according to the specifications reported into D5.3 [1] and D2.5 [2], which resulted into the overall decomposition and mapping to the reference technologies shown in Figure 14.

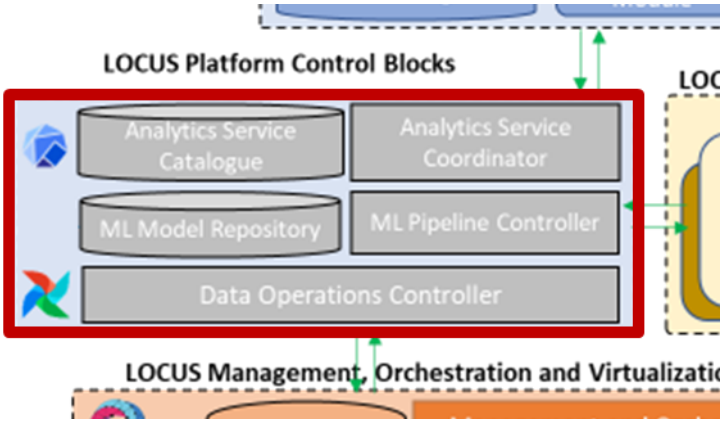


Figure 14: LOCUS control platform software prototype components

The LOCUS platform control software prototype has been developed according to the capabilities, components decomposition and workflows described in D5.3 [1]. It allows to hide the internal analytics details of functions and services hosted in the LOCUS platform, in terms of data operations control, and in terms of deployment of the involved functions as VNFs in the virtualized platform. At the same time, it is fully integrated with the API layer to provide seamless and comprehensive localization analytics as a service solution.

The following subsections provide details on each component of the LOCUS platform control software prototype.

2.2.1 Analytics Service Coordinator and Catalogue

The Analytics Service Coordinator is the bridge component between the LOCUS API layer and the LOCUS MANO, as it translates the requests coming from the API layer for the activation of new analytics services in the platform into actual operations in the platform infrastructure. A given analytics service activation (upon a subscription requested by an external application or user) is translated into a set of operations that include:

- identification of the virtualized services (i.e., NFV Network Services) required to implement the overall analytics service;
- deployment and automated configuration of the NFV Network Services through the LOCUS MANO;
- if needed, additional virtualized service configuration (i.e., “Day-2” configurations in the LOCUS MANO terminology [14]) to satisfy specific operation constraints;

- interaction with the Data Operation Controller for the execution of required data control operations, according to the constraints of the specific analytics service and involved functions.

The Analytics Service Coordinator provides therefore a workflow engine which is logically based on the information maintained in the Analytics Service Catalogue. Indeed, as specified in D5.3 [1] the Analytics Service Catalogue maintains for each analytics service exposed a set of information required to model its constraints in terms of decomposition into more granular services and functions deployable in the platform as VNFs or NFV Network Services, as well as requirements in terms of internal data operations and data exchange. Specifically, it keeps information that is not suited for the LOCUS MANO (as it is too analytics oriented) and that shall not be exposed to the API layer (for complexity reason), but is required to properly deploy and operate the various analytics services.

For these reasons, the Analytics Service Coordinator and Catalogue are tightly integrated, and therefore are provided as a single software prototype, as shown in Figure 15. More specifically, the catalogue component is implemented as a PostgreSQL database service [10], which maintains internal analytics services and functions according to the information models specified in D5.3 [1]. On the other hand, the Analytics Service Coordinator is implemented as a custom Python application, which provides the workflow engine logics, as well as the access (also for other LOCUS components, such as the API layer) to the Analytics Service Catalogue. It provides a set of REST APIs to manage the analytics service operations (i.e. invoked by the API layer) and enable CRUD operations on the catalogue database (i.e. invoked by administrator users and other LOCUS components). For these APIs, the Python FastAPI framework [10] has been used to have a direct integration with Keycloak [6] and the Swagger tools [12].

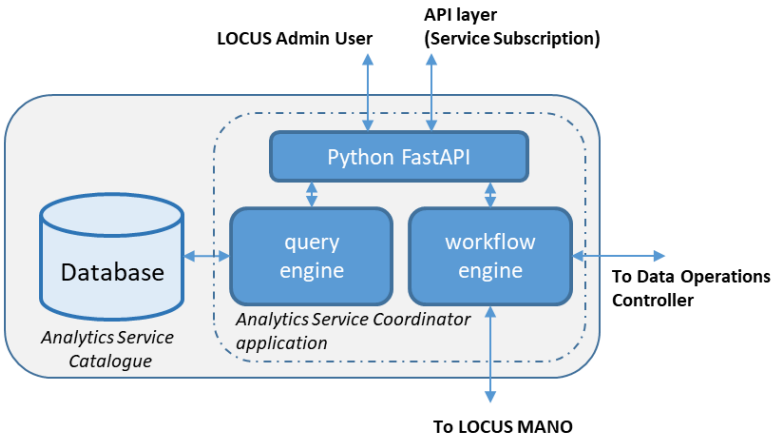


Figure 15: Analytics Service Coordinator software prototype

Figure 16 shows the APIs exposed by the Analytics Service Coordinator application in the form of a Swagger UI, as a result of the automated generation of API documentation from the Python FastAPI front-end. These are divided in: (i) Service Management APIs to manage the content of the Analytics Service Catalogue (with CRUD operations on the internal analytics services according to the information model of D5.3 [1]); and (ii) Service Operation APIs to activate and deactivate new analytics services to be first decomposed into NFV Network Services (as shown in the example of Figure 15) and then deployed and configured in the platform. These latter set of APIs can be either invoked by the API layer (as part of the automated activation of analytics services), as well as by LOCUS platform administrator users to deploy pre-scheduled services not activatable on-demand by external applications and users.

The API catalogue (integrated with the service subscription) is deployed and integrated in the OTE testbed infrastructure, available and accessible in a common LOCUS platform VM at:

IP: 172.16.12.46

Port: 9090

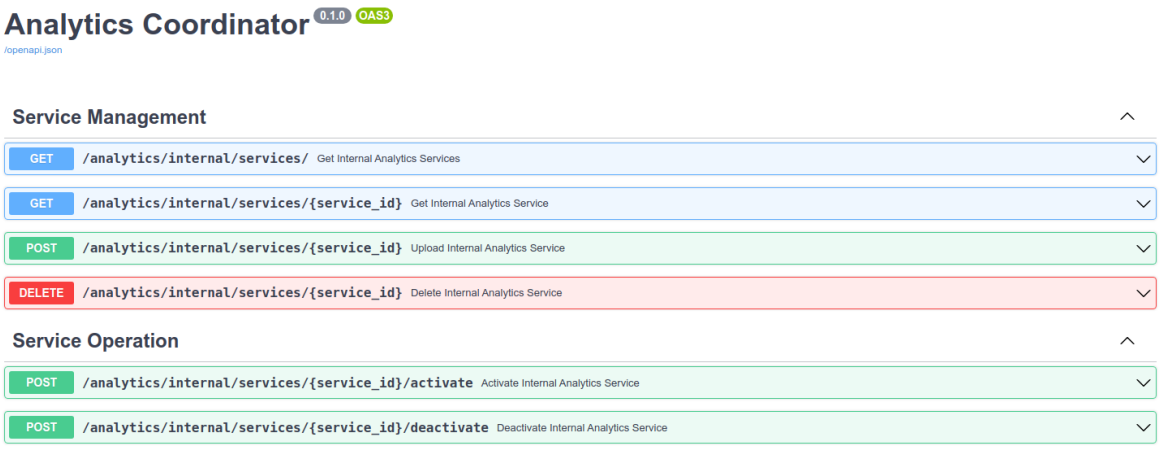


Figure 16: Analytics Service Coordinator exposed REST APIs

Similar to the Analytics Service Catalogue, it is worth to mention that, as specified in D5.3 [1] (with the related workflow), the onboarding of internal analytics services information (as per Service Management APIs above) is performed through offline procedures performed by the LOCUS platform manager/administrator.

2.2.2 ML Pipeline Control and Model Repository

The ML pipeline control functionalities have been implemented on top of the Kubeflow platform [19], following the technology selection exercise carried out in D2.5 [2]. This allows to implement the required functionalities and facilitate the deployment and operations of

virtualized ML solutions. It enables virtualized ML pipeline operations, in a portable and scalable way. Specifically, the implementation of Kubeflow [19] in the LOCUS platform enables a set of functionalities for the control and management of virtualized ML pipelines and, more generally, for the ML features, reported below, realized through the LOCUS platform itself. Among the ML features and workflows available and achievable with Kubeflow, the ones that have been implemented, tested and integrated (i.e., the ones that most suits the needs of the LOCUS platform) are described in the following paragraphs.

Pipelines on-demand execution

Pipelines developed and on-boarded in Kubeflow can be executed from the Kubeflow UI (Figure 17) or by leveraging the REST-based APIs exposed by the Kubeflow services (Figure 18).

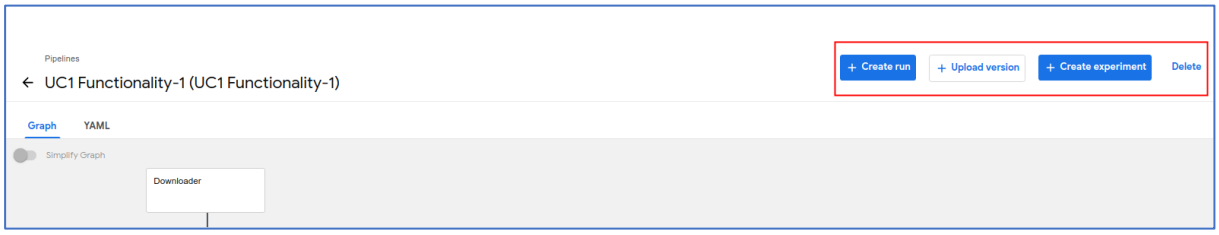


Figure 17: Kubeflow UI, available options for on-boarded pipeline

```
pipeline_args = {
    'dataset_url': DATASET_URL,
    'minio_url': MINIO,
    'minio_access_key': MINIO_ACCESS_KEY,
    'minio_secret_key': MINIO_SECRET_KEY,
    'model_filename': str(int(time.time())) + ".joblib"
}

kfp_client.run_pipeline(experiment_id = KFP_EXPERIMENT_ID, job_name = str(uuid.uuid4()), params = pipeline_args, pipeline_id = KFP_PIPELINE_ID)
```

Figure 18: Kubeflow Pipeline client, new pipeline execution request

On-demand training of Machine Learning Models

Pipelines developed with the specific task to train ML Models can be also on-boarded in Kubeflow and executed in the same ways reported in the “Pipelines on-demand execution” paragraph above; in these pipelines, all the steps required, from the data pre-processing to the training of the ML Model, as highlighted in Figure 19, are defined as reusable Kubeflow Pipeline components.

The implementation of Kubeflow and Kubeflow Pipelines and their capabilities to run, on-demand, ML Pipelines in the LOCUS platform makes suitable a seamless integration with AirFlow based Data Operations Controller through a dedicated Directed Acyclic Graph (DAG) providing a Kubeflow client leveraging the Kubeflow Pipeline Client library. Specifically, this on-demand training of ML models can be exposed as a dedicated internal platform service available for administrator users (i.e., internal use), exposed either through the Analytics Service Coordinator or the API gateway.

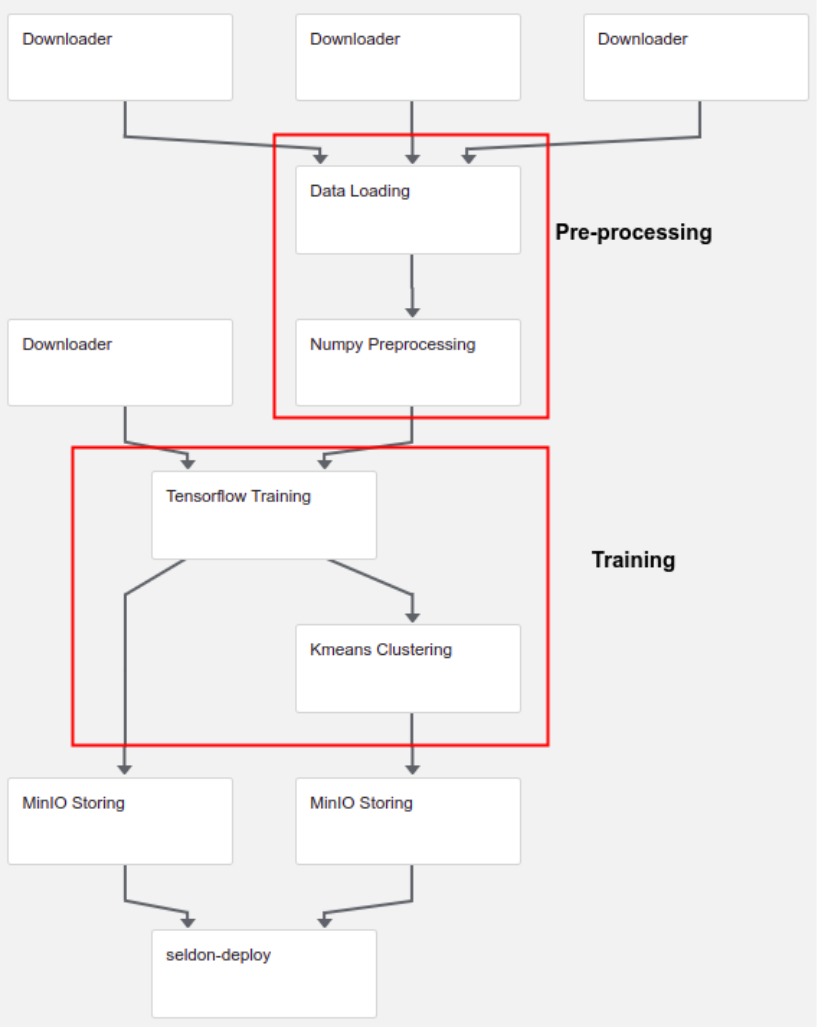


Figure 19: Context-Aware UC1 Functionality-1 ML Pipeline, pre-processing and training steps

Machine Learning Models Persistence

ML Models trained in the execution of a ML Pipeline can be saved (i.e., persisted) in the LOCUS ML Models repository (Figure 20) in order to be later used by other components of the platform and deployed by the means of the LOCUS MANO (and thus ETSI OSM [21] as described in D4.4 [14]) in order to be used to perform predictions on new data.

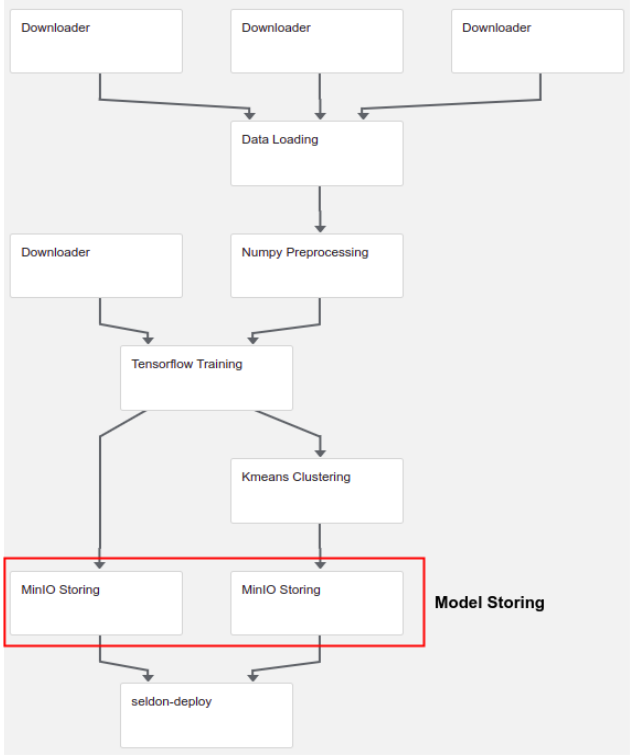


Figure 20: Context-Aware Crowd mobility ML Pipeline, Storing of the Encoding (Tensorflow Training) and Clustering models

The LOCUS ML Models repository has been implemented as a MinIO Object Storage [21] where the ML Models trained by the execution of the Kubeflow Pipelines are stored, by reusable Kubeflow Pipeline components that makes use of a MinIO client, for further use with a fixed name nomenclature: *pipelineId-modelName-timestamp*.

Deployment of trained Machine Learning Models

In a ML Pipeline, optionally, a model that has been trained and stored in the LOCUS ML Models Repository, can also be deployed in an underlying Kubernetes cluster in order to be used to perform predictions on new incoming data exposing a RESTful northbound interface. The deployment of a ML Model is performed through the use of Seldon and Seldon-Core [20] as the last step of a ML pipeline (Figure 21) referring to the ML model saved in the LOCUS ML Models repository previous steps of the pipeline. The main functionalities and characteristics of Seldon have been already reported and described in D5.3 [1]and D.5.2 [4].

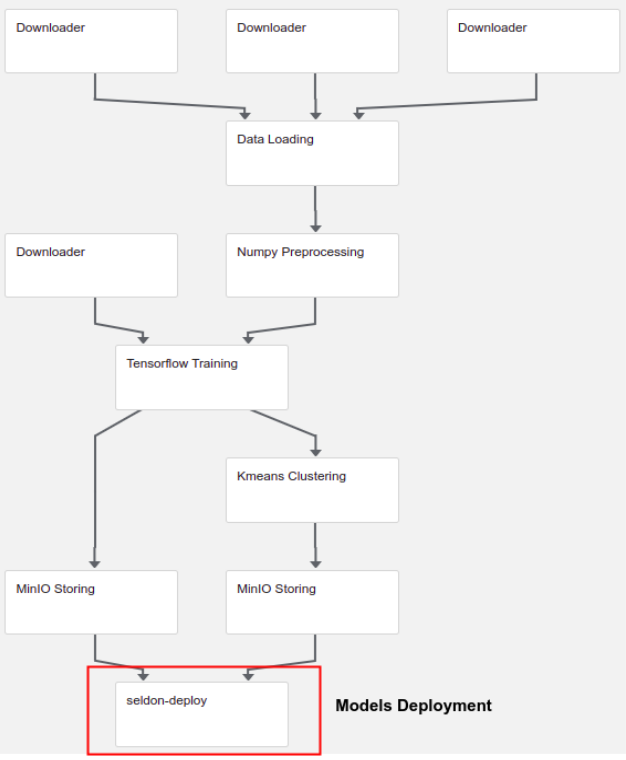


Figure 21: Context-Aware UC1 Functionality-1 ML Pipeline, deployment of a chain of ML Models: encoding (Tensorflow Training) and clustering

2.2.2.1 Deployment details

The Kubeflow component has been deployed in the OTE testbed to validate and test it against the other modules of the LOCUS architecture and the LOCUS platform itself. Kubeflow has been deployed in the Kubernetes cluster available in OTE, composed of two worker nodes and one master node (as described in D4.4 [14]), following the quick start guidelines available for the charmed-Kubeflow deployment, which leverage on Juju [23] to deploy and manage the components which make up Kubeflow using the provided collection of Juju Charms (i.e., structured bundles of YAML configuration files and scripts for a software component that enables Juju to deploy and manage the software component as a service according to best practices). Once Juju has been installed on the machine from which we want to deploy Kubeflow and used to bootstrap the targeted Kubernetes cluster, Kubeflow has been deployed simply using the:

juju deploy kubeflow-lite –trust

command. For more details the charmed Kubeflow quick start guide is available [23]. The Kubeflow UI is exposed with a Kubernetes NodePort service and it is available at:

IP: 172.16.12.65

Port: 31080

2.2.2.2 Optimization of Machine Learning workloads using SOL acceleration middleware

SOL, as described into D5.3 [1], provides state-of-the-art optimization on top of well-known Deep Learning frameworks, where optimization includes performance and energy consumption. The ML Pipelines created and on-boarded for/into Kubeflow can leverage the SOL acceleration middleware simply importing the appropriate library in the code that describe the various steps of the pipeline (Kubeflow Components) where the SOL optimization can be performed. The following paragraph describes in detail how the SOL acceleration middleware works.

SOL Acceleration middleware

The LOCUS ML pipelines (as the exemplary one reported in for Figure 22 for NSE-UC1 flow monitoring service and crowd mobility patterns functionality [3]) can leverage on the SOL acceleration middleware introduced to implement ML acceleration of ML workload, as introduced and described in D5.3 [1].

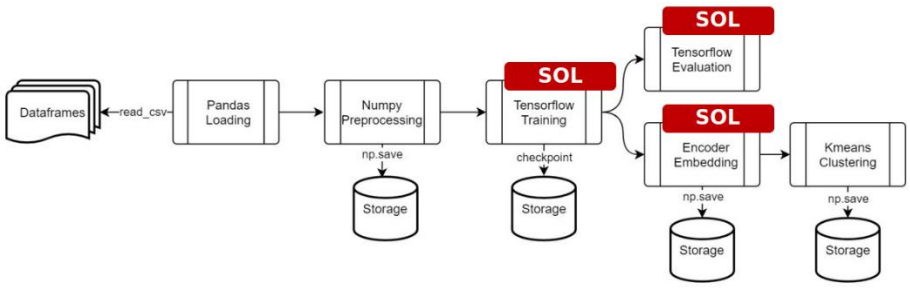


Figure 22: Exemplary LOCUS ML pipeline

SOL acceleration can be applied to several blocks of the LOCUS ML pipeline such as the Training and Inference (as well as Evaluation); moreover, also the Embedding encoder block can be accelerated in case the Embeddings are obtained through Neural Network inference. For instance, this is the case of embeddings generated as output of an autoencoder's hidden layer. The application of accelerations to these modules happens transparently to both users and control plane.

The key to transparent integration is to tap into existing ML frameworks without disrupting the surrounding ecosystem. Indeed, all the ML serving frameworks (e.g., Kubeflow [19], Seldon [20]) are already designed to interact with ML frameworks (e.g., TensorFlow [25], PyTorch [26]), so integrating SOL acceleration directly inside the ML framework allows to reuse all the API toward users and Control plane. The approach we used in LOCUS to integrate SOL acceleration middleware in TensorFlow (TF) [25] is shown in Figure 23; a similar approach is used for other ML frameworks (e.g., PyTorch [26]).

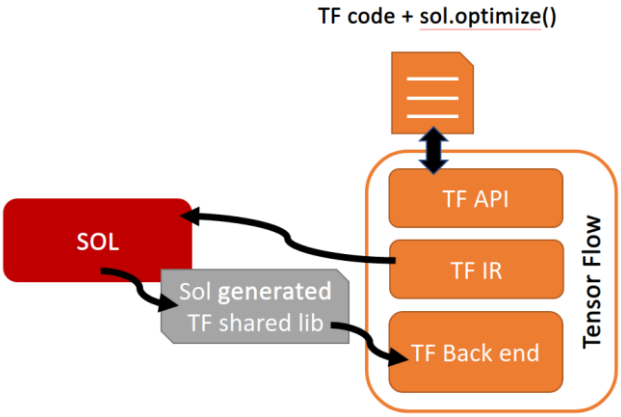


Figure 23: SOL Acceleration middleware TensorFlow integration

SOL takes as input the TF Intermediate Representation of the code describing the model and its operations, it then performs the optimization described in D5.3 [1] (e.g., memory layout optimization, deep first parallelism, layer fusion). Finally, it generates a set of shared libraries that are reinjected into TF and can be directly used without the need to recompile the ML framework itself. TF will then see the optimized model as a single Super Layer that internally implements the original model. There are three types of generated shared libraries each one with a different scope:

- Connector: used to connect the TF runtime system to the SOL runtime system. It contains the definition of the SOL super layer.
- Compute libs: these libraries contain the implementation of the forward passes for inference/training and backward pass for training. The actual computation happens inside these libraries.
- Model wrapper libs: a Python wrapper, generated per each optimized model, which mimics a TF model but internally calls SOL. It calls the SOL super layer, which then triggers the execution of the Compute libs.

Figure 24 shows a TF model before (left) and after (right) the SOL optimization pass. As described above, TF sees the models as a single custom layer (sol_layer_65038aac in figure) that contains the optimized model. All the APIs remain unmodified and they can be invoked normally.

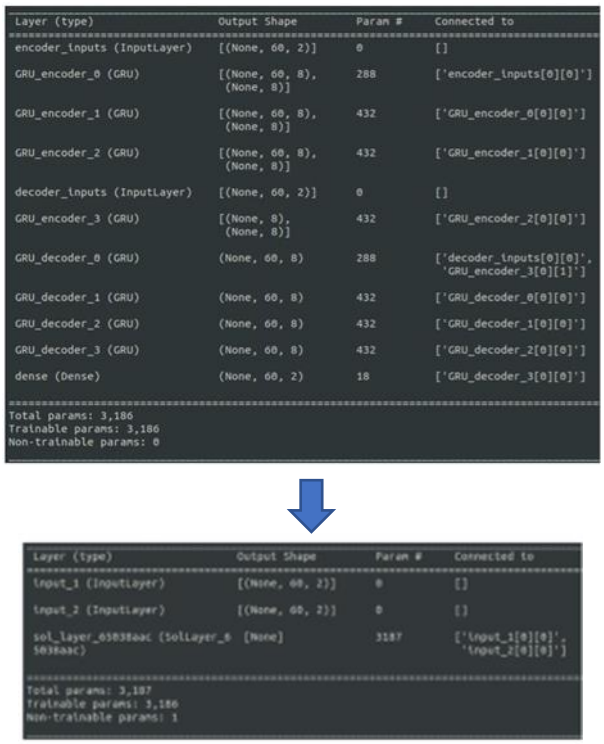


Figure 24: TensorFlow model before and after SOL optimization pass

From the user perspective, the optimization step is almost transparent, in fact, it has only to import the SOL library and then call the optimize function on the model. All the operations (e.g., train(), test()) performed through the ML framework APIs will be automatically executed on the optimized model. Running an optimized model in the LOCUS ML pipeline is also easy. Indeed, since the model is in fact running inside the ML framework, the Control plane can interact with the model using the ML framework interfaces. The only required step is to have SOL in the pipeline component container (training, evaluation, or encoding) that is operating on the model. This can be easily done by installing SOL along with the other requirements of the component (e.g., TensorFlow [25] or Pytorch [26]) or by directly using a container image with SOL preinstalled (Figure 25).

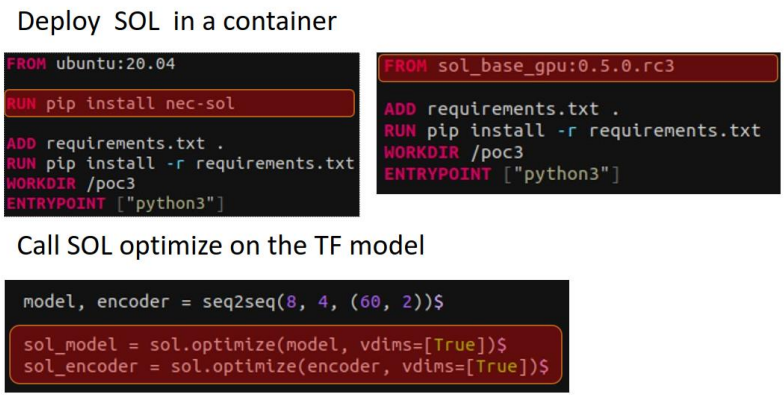


Figure 25: Operations required to optimize and execute a model in the LOCUS ML pipeline

SOL results

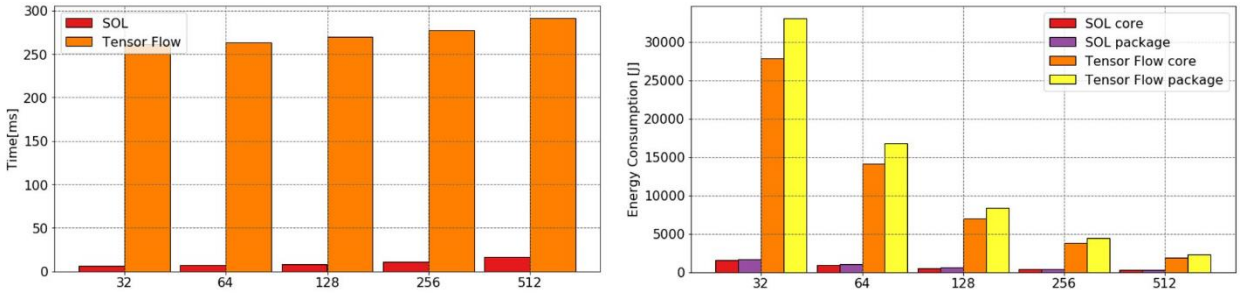


Figure 26: People mobility model performance w/o and w/ SOL optimization

In Figure 26 (left) we present the inference latency of the people mobility model when using plain TensorFlow [25] and when using Tensorflow plus SOL optimization. The model is composed of eight Gated Recurrent Unit (GRU) layers plus two Dense layers and is executed on the CPU (Intel Core i7-9700K @ 3.60GHz). For this use case, SOL provides remarkably faster inference times, up to 40x with a batch size of 32 and up to 17x with a batch size of 512. We also measured the energy consumption of the CPU while performing the inference with TensorFlow and SOL plus Tensorflow. We did not measure the PCIe bus and PCIe cards since we do not rely on GPUs in our experiment and the energy consumption related to input/output data transfers (i.e., network or disk) is constant in both scenarios (w/o and w/ SOL acceleration). We also measured the energy consumption related to RAM access, but we do not report these results since its overall energy consumption is negligible compared to CPU, thus the energy saving will not have impact on the overall system consumption. Figure 26 (right) shows the results of the test. In this case, SOL uses from 19x to 7x less energy than TensorFlow. Execution time and energy savings are directly correlated, indeed the more a system is on the more it will consume energy, but faster execution time comes with a cost. In fact, we observe that this is an expected outcome since some of the optimizations provided by SOL improve memory access with the goal of reducing stall cycles in the CPU [1]. Thus, the CPU will spend fewer cycles waiting for the memory and will be able to work at a higher P-state (i.e., higher frequency and voltage) for more time, increasing the overall energy consumption.

2.2.3 Data Operations Controller

LOCUS analytics services have complex dependencies in terms of data, other analytics services, message events and resource demands. The Data Operations Controller allows for a high-level pipeline coordination, working as a centralized and stable scheduler that will ensure the timely outcomes for end-to-end analytics results.

In more detail, as described in the LOCUS platform architecture documented in D2.5 [2], this control block is responsible for the following:

- Based on the input configuration request of the specific analytics service and the internal dependencies of the service on LOCUS functions (service pipeline), it requests their instantiation, passing on the configuration parameters to each children service instance.
- Uses parallel and sequential analytics service subtask execution in the form of a DAG. Given all the necessary requirements are met, the pipeline is being executed via the designated interfaces.
- Tracks the status of the instantiated analytics service sub-tasks; the spawned services must provide health information to the Data Operations Controller to start the chain execution, similarly in the case of a completed or failed task (in which case appropriate actions take place).

Based on the technology scouting that took place, the LOCUS Data Operations Controller is developed based on Apache Airflow [18]. Apache Airflow is a multi-purpose high level data operations orchestrator for a very wide variety of underlying systems and/or task allowing for a smooth execution of the various combined models.

Apache Airflow DAGs are used to design parallel, serial, dependant and independent jobs to build high level architectures of data pipelines and also various "Operators" (e.g. ShellOperator, HttpOperator, PythonOperator, JavaOperator) to perform tasks. From its UI, the various pipelines are catalogued and monitored. Indicatively, Figure 27 presents the menu with the various pipelines, listing an example pipeline related to PoC#3 scenarios and its status (i.e. "Active" in the mentioned figure).

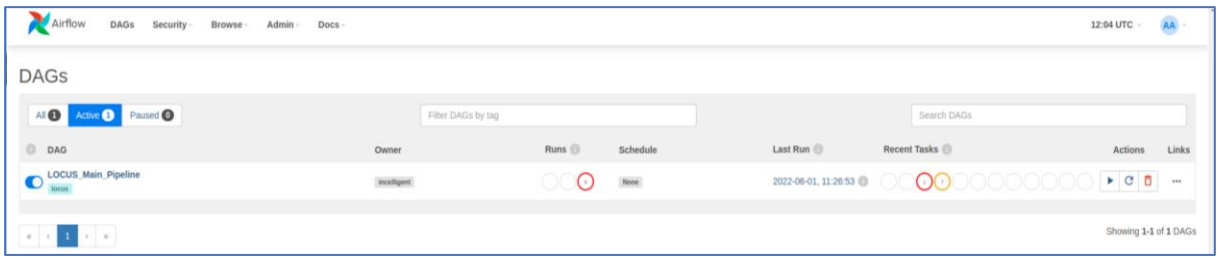


Figure 27: LOCUS Data Operations Controller – Pipelines & status

The sequence of invocation for these jobs is critical for their healthy operation, which is why each LOCUS function has to provide its own, respective DAG for executing its internal step-based tasks. Figure 28 presents an indicative DAG of the various functions used for different PoC#3 scenarios related to flow tracking, transportation optimization and collision detection, starting from the fingerprinting function which uses ML to estimate the position of the User Equipment (UEs) in a given area. After the training stage the prediction (position estimation) stage occurs. Then points of interest (POIs) and Paths are extracted as part of the flow monitoring and transportation scenarios and the extracted geometric objects will be correlated real-time or near real-time by finding geometric intersections through the relevant

correlation function. In parallel, previous trajectories are used for training and then prediction of the future UE trajectories as input for the collision detection functionality.

Lastly, Airflow allows access to the appropriate logs and monitoring information through its UI as shown in Figure 29.

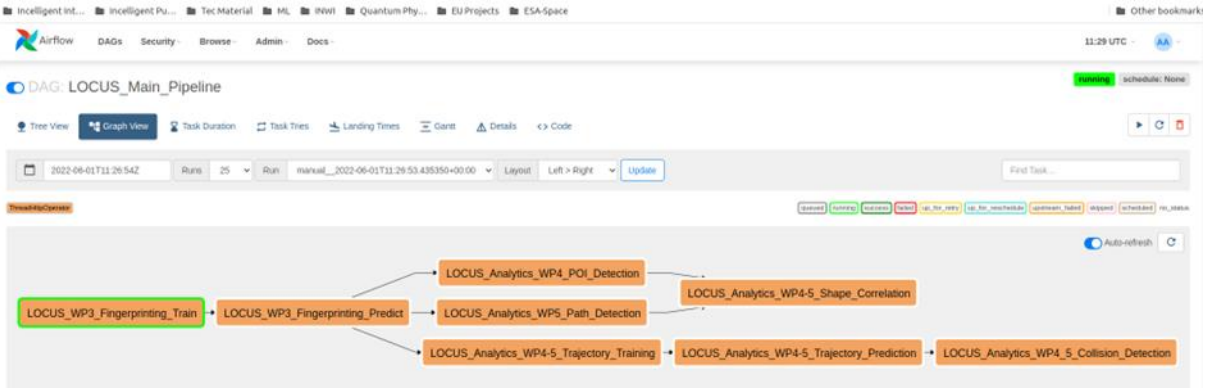


Figure 28: LOCUS Data Operations Controller – Indicative DAG for PoC#3-related scenarios

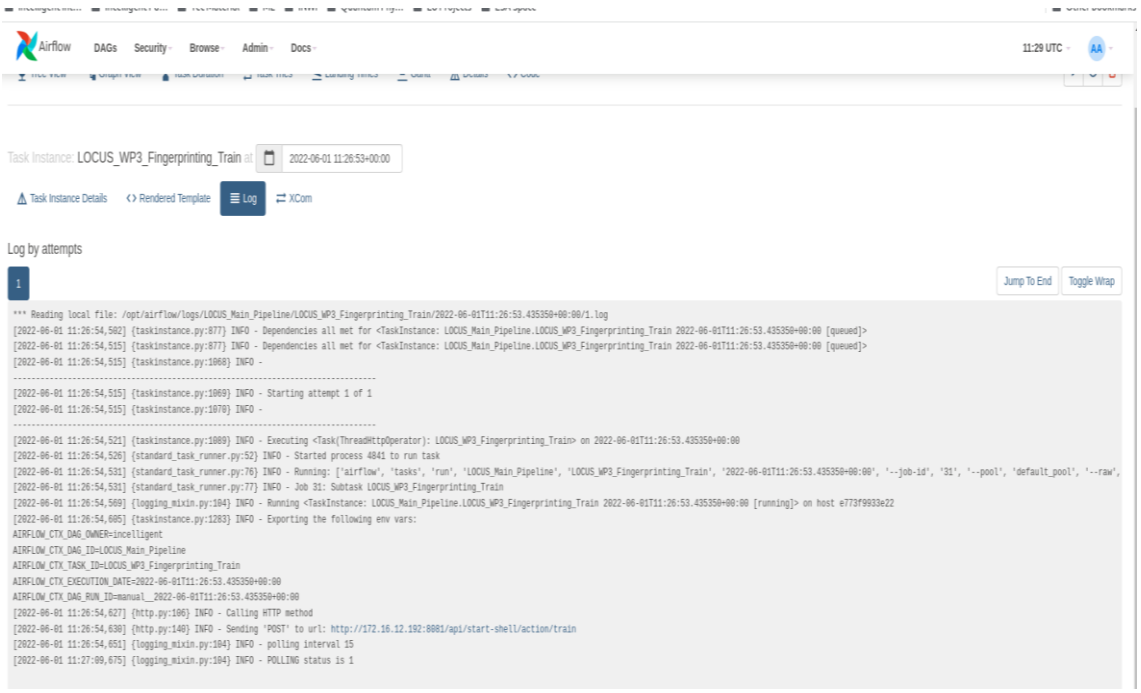


Figure 29: LOCUS Data Operations Controller – Logging & monitoring

The Data Operations Controller implemented in Airflow is deployed and integrated in the OTE testbed infrastructure, available and accessible in a common LOCUS platform VM at:

IP: 172.16.12.46

Port: 8088

3 LOCUS Data Platform prototype

This section focuses on the LOCUS data platform software prototype. In particular, it describes how data movement, data exchange and data persistence capabilities have been implemented in the LOCUS platform, and then deployed and integrated in the reference virtualized infrastructure running at the OTE testbed.

More specifically, starting from the design and approaches envisaged in D5.3 [1], this section describes how the various functionalities for real-time data stream exchange and data persistence have been implemented, to support the communications among the LOCUS analytics service components and functions.

Figure 30 highlights the LOCUS data platform software prototype components with reference to the technology selection performed in D2.5 [2] for the overall platform architecture. It is implemented by the integration of the LOCUS Message Queue module and the LOCUS Persistence module.

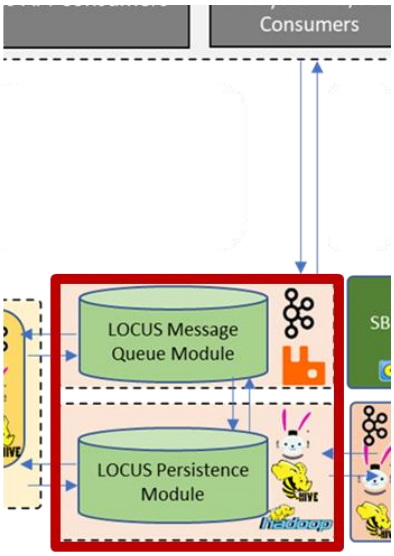


Figure 30: LOCUS Data Platform software prototype components

3.1 Message Queue module

The LOCUS Platform implemented the Message Queue Module based on RabbitMQ [27]. RabbitMQ provides a publish/subscribe framework, which allows to decouple services while enabling asynchronous communications among them. It provides a broker component, whose main job is to filter and distribute all the produced messages to the subscribers. It supports the advanced message queue protocol (AMQP) [28], and - in the latest releases - it implements other protocols like MQ Telemetry Transport (MQTT) [29] and streaming text-oriented message protocol (STOMP) [30].

In D5.3 [1] we proposed a data movement layer, to allow efficient data streaming and flexibility in deploying and pipelining microservices, the initial design proposed the reliance on Kafka capabilities to enable the microservices based approach of LOCUS.

The capabilities targeted by initially proposing Kafka [31] consist of: **a)** allowing for a high level of separation of concern in designing the microservices; **b)** allowing the developers to build efficient services that consume and produce data, while delegating data movement and all the necessary data operations (replication, queuing, routing, etc) to the data broker. An important aspect we emphasised in the previous deliverable is the ability to adapt to a distributed environment where services might be deployed at various physical location, and that this distribution should not affect services operations and should be handled seamlessly by the data broker.

During the development of LOCUS PoC#1 in WP6, RabbitMQ was relied on as the message broker for the LOCUS data movement platform service, instead of Kafka, for technical and non-technical reasons. The non-technical reasons can be summarized as developer experience within the consortium that made it more convenient to deploy RabbitMQ, and with less development effort than deploying Kafka.

The technical reasons behind this choice can be summarized as:

- The smart broker/dumb consumer model better fits the identified use-cases. The advanced routing and queuing capabilities of RabbitMQ compared to Kafka, allows for better reliability and guarantees message delivery. With RabbitMQ, consumers can leverage the routing features for subscribing and consuming from the relevant producers, which enables more flexibility in a microservices environment where multiple services could produce data on the same topics using the same data model.
- Both Kafka and RabbitMQ are tailored for data-intensive applications, and even if there is a common perception that Kafka could be more suited for handling high volume throughput applications, RabbitMQ offers lower latency messaging capabilities, the Smart broker model, in addition to the capability of handling high data volumes. Hence RabbitMQ is deemed a better choice for the targeted use cases in LOCUS.
- RabbitMQ implements the Advanced Message Queuing Protocol AMQP [28], a standardized and open protocol. With mature support for major languages, and through third-party clients implementing the AMQP protocol.

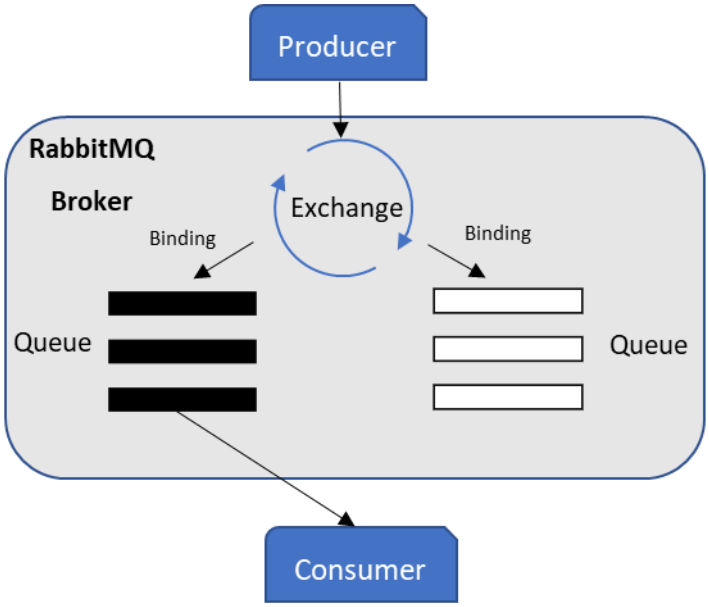


Figure 31: RabbitMQ message flow

In RabbitMQ, the message broker receives the messages from the producer, first at the exchange level where they are routed to different queues (see Figure 31). The exchange type defines the parameters that dictate which queues will receive the message. RabbitMQ defines four exchange types:

- Fanout Exchange: all the queues that are bound to an exchange will receive the message.
- Direct Exchange: a routing key provided by the producer is used by the exchange to choose the queues with an equal binding key to route the message to.
- Topic Exchange: allows the use of wildcard matching, where a routing key can define a pattern that matches multiple queues.
- Header Exchange: the message header attributes and values are used for routing.

Based on the exchange type, the message is forwarded to a single or multiple queues, where it stays until a consumer subscribes to the exchange and binds to the queue to receive the message. The message is deleted from the queue upon acknowledgement by the consumer.

The different exchange types allow for the deployment of complex data flows: unicast, broadcast, and multicast. Services can receive generic broadcast updates and can also selectively consume messages based on specific patterns of producers.

As mentioned above, an additional requirement of the LOCUS applications and distributed services is the ability to efficiently access the data broker. RabbitMQ supports clustered and distributed deployments, where the broker itself is distributed. This is achieved using two different approaches: Clustering and Federation.

In a clustering deployment, compatible versions of RabbitMQ can be inter-connected, and clients can access the broker as in a single node setting, without worrying about the underlying

inter-node communication between RabbitMQ nodes that involves replication of users, exchanges, queues, etc. The interconnection links for a clustered deployment must be sufficiently reliable and provide good latency.

In a federated deployment, brokers are logically separated and connected with point-to-point links. This type of deployment is robust with respect to unreliable links such as connections established over the Internet. Point-to-point connection must be defined for specific exchanges and queues.

3.1.1 Deployment details

Version 3.9.8 of RabbitMQ is used to implement the Data movement service of the LOCUS platform as described in D4.4 [14], which reports the implemented LOCUS virtualization platform. It is deployed in a single node configuration, as a VM on top of the Openstack [32] infrastructure implementing the LOCUS Virtualization Platform in the OTE testbed.

The message bus is accessible to deployed containerized and virtualized services at:

IP: *172.16.12.27:5672,*

port: 52672

while the RabbitMQ web interface is accessible at:

http://172.16.12.27:15672

A use-case of the Data movement platform service is illustrated in Figure 33. Where 2 location services publish geolocated network information with two different positioning techniques, the messages are published on the same exchange *PositionAndNetworkInfo.* Two different queues are bound to the exchange using two different binding keys. The consumers of this exchange can choose whichever location estimate use for their analytics functions based on the corresponding routing key. Two analytics VNFs are deployed in this case, one VNF comprising a single Kubernetes pod, and a second VNF comprising 3 Kubernetes pods. All the exchanges among the pods rely on a direct exchange type and the use of routing keys to bind to a specific producer's messages

RabbitMQ™ RabbitMQ 3.9.8 Erlang 24.1.4

Overview | Connections | Channels | Exchanges | Queues | Admin

Exchanges

All exchanges (17)

Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D	0.00/s	0.00/s	
Clear_Data	fanout		0.00/s	0.00/s	
ContextIndUpdate	fanout		0.00/s	0.00/s	
Diagnosis_topic	fanout		0.00/s		
PositionAndNetworkInfo	direct		0.00/s	0.00/s	
Raw_Data	fanout		0.00/s	0.00/s	
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
covcorrs	direct		0.00/s		
covholes	direct		0.00/s	0.00/s	
krigingout	direct		0.00/s	0.00/s	
nMedAllAreas_topic	fanout		0.00/s	0.00/s	
rsrpAllAreas_topic	fanout		0.00/s	0.00/s	

Add a new exchange

Figure 32: RabbitMQ interface

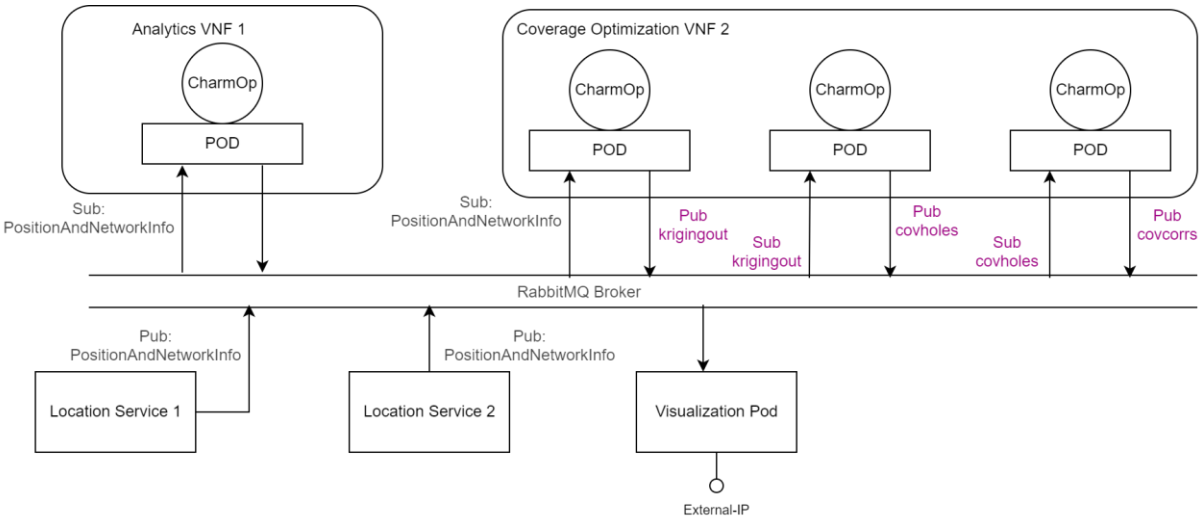


Figure 33: Messages Exchange on the Datamovement Platform Function

3.2 Persistence Module

As described also in D2.5 [2], the LOCUS Persistence Module is an important block in the LOCUS Platform as it allows the storage of both structured (tabular) and unstructured data. Due to the nature of the platform, i.e., aiming to accommodate multiple services, multiple tables and schemas etc., the LOCUS Persistence Module should be a distributed filesystem that includes mechanisms for raw data transformations into structured and big data query engines for fast read operations, as well as direct analytic execution via aggregation functions, window functions and predicate pushback for filtering and selection. Furthermore, it is essential to offer scale-out capabilities, and therefore the selected technologies to make it easy to increase/ decrease nodes (even if we are talking about replica VMs or Primary/ Secondary nodes).

3.2.1 Deployment details

In order to maximize input-output (IO) speeds, this module was not deployed as a container (containers use raw socket network interfaces with relatively slower performance), but instead was deployed on top of bare metal VMs in the OTE testbed. The technologies acting as the LOCUS Platform's Big Data Lake are the de-facto standard for multi-VM multi-disk, high availability, multi-node environments and revolve around the Apache Hadoop project [33] that were also described in D2.5 [2].

More specifically, in the node available at:

IP: 172.16.12.33

of the OTE testbed, under the:

/usr/HDC

directory the following have been installed:

- Apache Hadoop:
    - Apache Hadoop Distributed File System (HDFS), which is the storage layer for multiple cloud persistence deployments, allowing for addition/ removal of storage nodes and high throughput (HTTP-based) read and write operations. In Figure 34, the relevant UI is indicatively presented, i.e. a high-level overview of HDFS (left) and a datanode overview (right). The UI is available at:

      http://172.16.12.33:9870/dfshealth.html

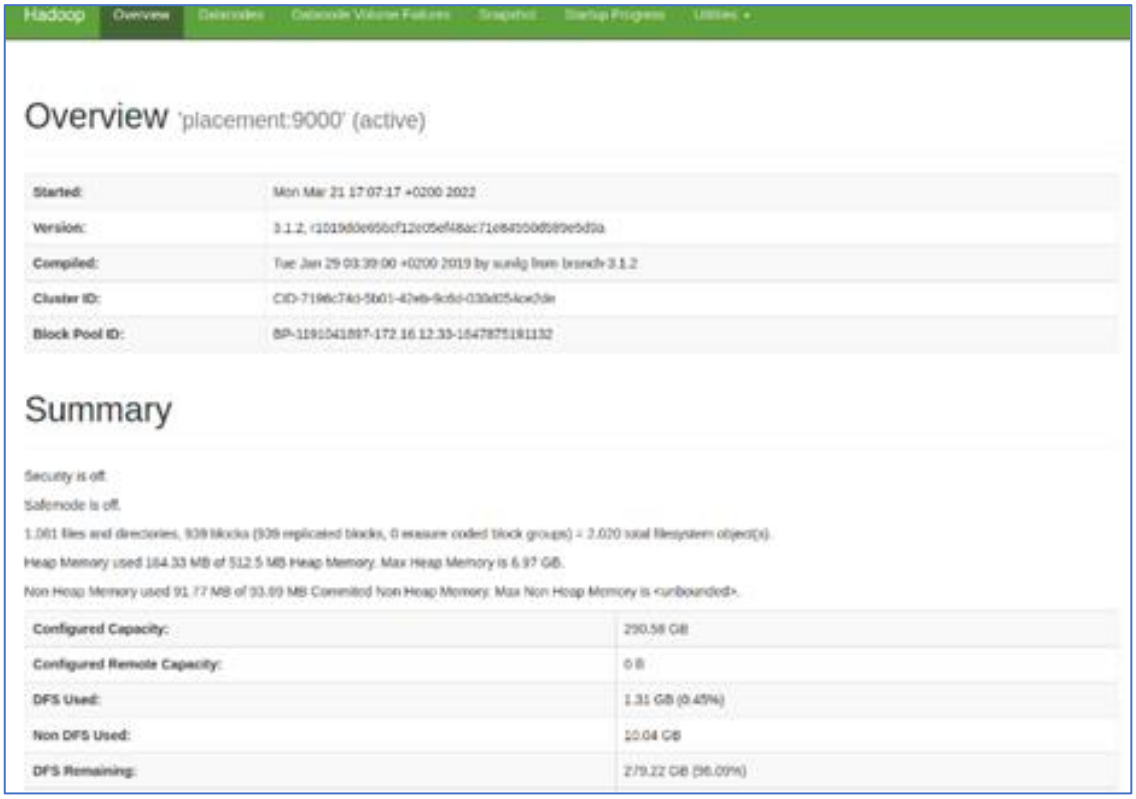
    - Apache Hadoop YARN [34] responsible for resource management (managing resources in clusters) and job scheduling/monitoring (for user application). An indicative screenshot of the UI containing configuration details of the cluster and a list of the executed applications is shown in Figure 35. The UI is available at:

http://172.16.12.33:8088/

- Apache Hive [35]: Built on top of Apache HDFS, is basically an SQL tabular data storage layer that facilitates reading, writing and managing large datasets in HDFS. It utilizes Hadoop map-reduce binaries to transform various data sources (that exist within the HDFS) into different file formats that are more optimized for compression or analytics, while it allows other query engines and SQL interfaces to perform SQL-like operations.
- Trino (Former Presto SQL) [36]: Trino is a distributed query engine for big data using the SQL query language. It is optimized for multi-machine scaling on top of SQL data storage catalogues such as Apache Hive. By splitting the read and write operations between those two technologies, we take advantage of the speed of Trino and the stability of Hive. Figure 36 shows the Trino UI, and specifically an overview with a list of SQL queries. The UI is available at:

  http://172.16.12.33:8080

(a)



(b)

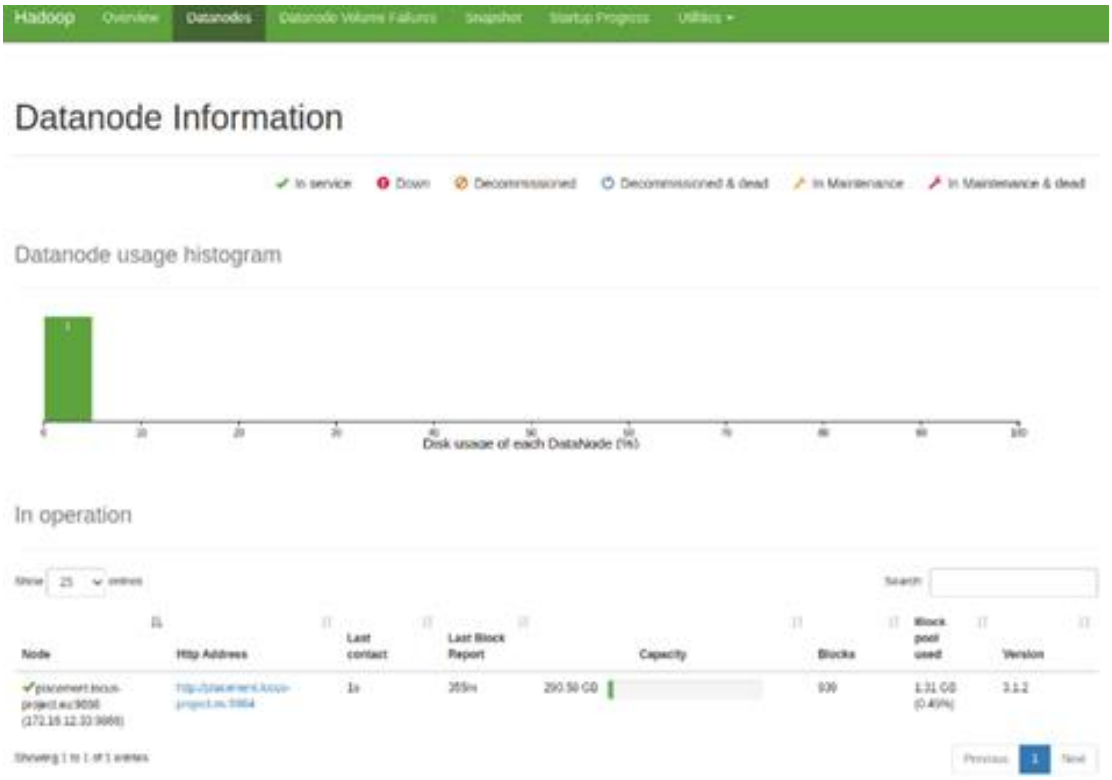


Figure 34: Examples of Hadoop user interface: (a) HDFS overview and (b) Datanode information menu.

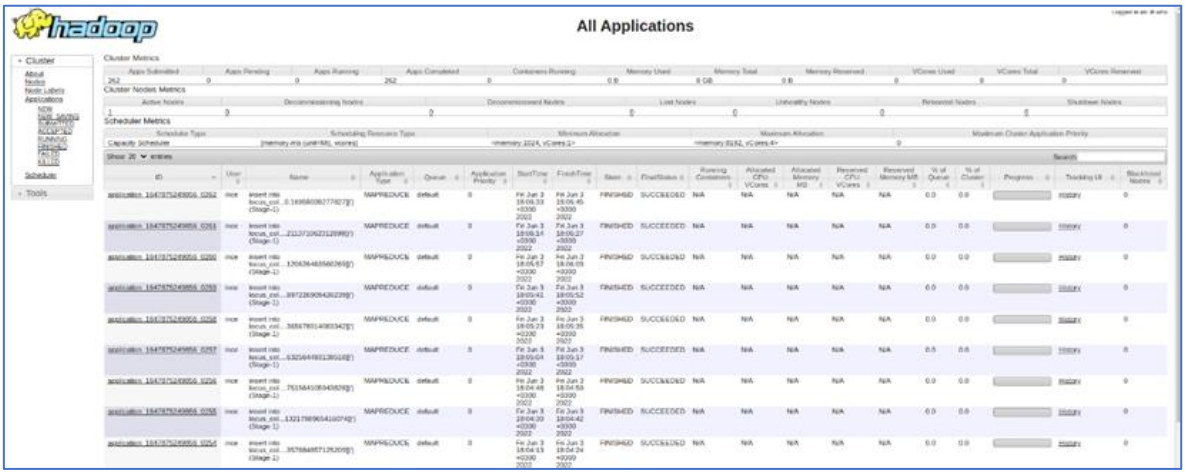


Figure 35: YARN UI: Cluster configuration details and executed applications.

Although not part of the LOCUS Platform design and not necessary for the LOCUS services' deployment, Apache Zeppelin [37] has also been installed and exposed in the OTE testbed at:

http://172.16.12.33:9995

for the purposes of data visualization and monitoring of changes in data/tables. It offers the possibility of creating web-based notebooks that enable interactive data analytics with Hive, Trino and others. Figure 37 presents the UI welcome page (top) and an example notebook exploring tables in the LOCUS Persistence Module (bottom).

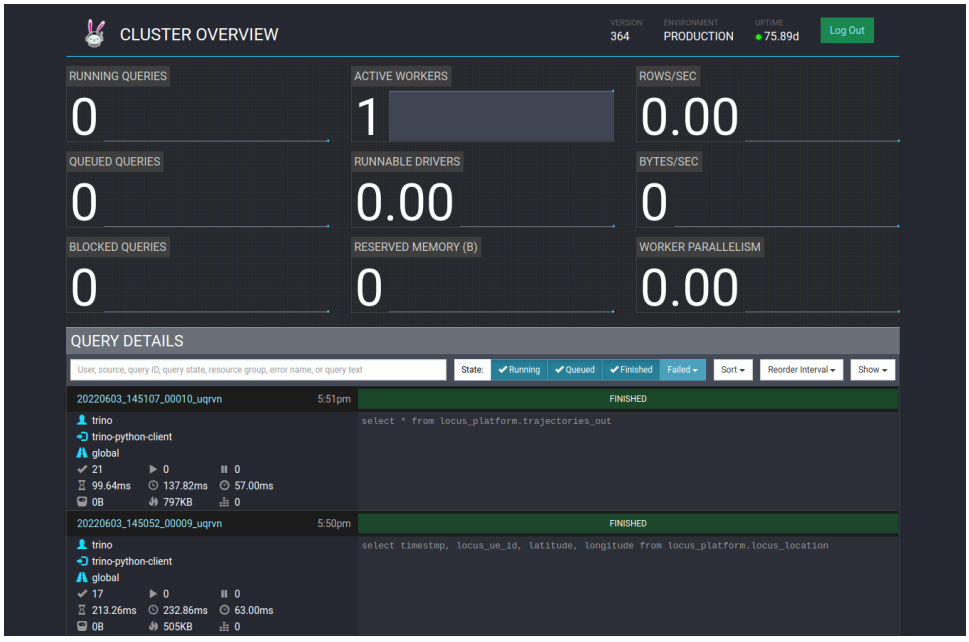


Figure 36: Trino Cluster Overview and list of SQL queries.

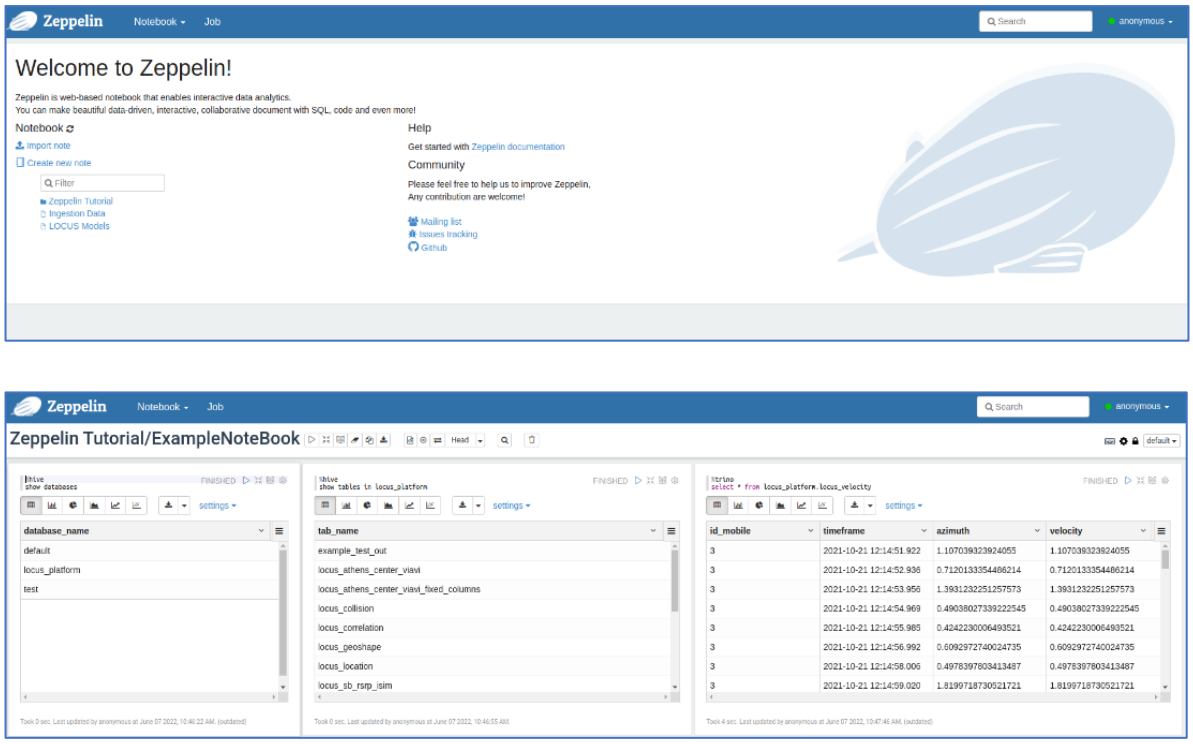


Figure 37: Apache Zeppelin welcome page (top) and example notebook (bottom).

3.2.2 Dataset description and schema

In terms of datasets, for the purposes of the PoCs, there are two types already to be found in the module:

- one related to a simulated scenario (IncelliSim dataset), and
- one based on the real measurement scenario (coming from the UMA testbed).

Both datasets follow similar structure, containing network measurements, mobile locations (in the form of latitude-longitude coordinates) and network measurements per location (e.g. RSSI, RSRP, RSRQ).

These datasets are then exploited by the various LEN, Smart Network Management (SNM) and NSE functions. Indicatively, with respect to the LOCUS PoC#3 which currently exploits these datasets, the associated measurements are fed to a LEN function representing the fingerprinting model developed in WP3 to train and produce the first destination table with the predicted locations (*locus_location*). This is then used as input for the other service models developed in the context of WP4 and 5 for POI/Path detection and trajectory prediction. The latter produce the intermediate and destination tables *locus_geoshape* (containing extracted POIs/Paths), *locus_velocity* (containing velocity measurements of mobiles) and *locus_trajectories* (containing predicted trajectory per mobile). These will be also used as input for the final service models, the correlation model which will produce *locus_correlation*, with the related geoshapes of the previous steps, and the collision model, with the possible

collision between predicted trajectories and other shapes or trajectories (*locus_collision*). The abovementioned tables adhere to the data schema analysis described in Deliverable 2.5 [2]. The description of these selected tables and their columns are presented in Table 2 to Table 8 below.

Table 2: Source table - UMA testbed data, indicatively for 2 sites.

Column name	Data type
position	string
numberofsites	int
site0_rssi	double
site0_rsrq	double
site0_pci	int
site0_rsrp	double
site0_registered	double
site0_dbm	int
site0_levelsignalstrength	int
site1_rssi	double
site1_rsrq	double
site1_pci	int
site1_rsrp	double
site1_registered	double
site1_dbm	int
site1_levelsignalstrength	int
x	double
y	double

Table 3: Intermediate table – Velocity details ("locus_velocity").

Column name	Data type
id_mobile	int
timeframe	string
azimuth	double
velocity	double

Table 4: Destination table – Positioning results ("locus_location").

Column name	Data type
simulation_id	string
timestmp	double
locus_ue_id	string
latitude	double
longitude	double
relative_x	double
relative_y	double
relative_shape	string
source	string
accuracy_class	string

Table 5: Destination table – Area geometry details ("locus_geoshape").

Column name	Data type
shape_id	string
geometry	string
timeframe	string
parent_shape_id	string
type	string
source	string

Table 6: Destination table – UE-shape correlation ("locus_correlation").

Column name	Data type
timeframe	string
id_mobile	int
shape_id	string

Table 7: Destination table – Trajectories' details ("locus_trajectories").

Column name	Data type
trajectory_id	int
user_equipment	int
start_timestamp	string
end_timestamp	string
geometry	string

Table 8: Destination table – Collision details ("locus_collision").

Column name	Data type
collision_start	string
collision_end	string
colliding_ue_1	int
colliding_ue_2	int
colliding_trajectory_1	int
colliding_trajectory_2	int
geometry	string

4 LOCUS Virtualized Analytics Services prototypes

This section describes a set of development updates related to the LOCUS virtualized analytics services implementation, as a follow-up of the design and preliminary prototypes of the correspondent WP5 use cases (NSE-UCs). While this section does not cover the whole set of NSE-UCs, it focuses on those analytics services that required an update in terms of software development since D5.3 [1] and D5.2 [4].

Among the others, the virtualized analytics service prototypes described in this section (even if not integrated with the LOCUS platform yet – and therefore not yet fully compliant with the solution presented in section 2) are candidates for integration in the final LOCUS PoCs. This will require specific adaptations to integrate them with the LOCUS localization analytics as a service prototype described in this document; such integration will be continued in the last phase of the project and will be described in D6.3 [5].

4.1 Individual mobility pattern detection and prediction

In D5.1 [3], a generic ML pipeline was defined for analytic services. Our individual mobility pattern detection and prediction service includes a set of deep learning models trained on a number of different datasets. For complete list of datasets used to train models, see deliverable D5.2 [4]. The objective of orchestrating a ML pipeline is to expose each trained model as VNFs. For individual mobility and prediction service, Figure 38, graphically describes the ML pipeline for individual trajectory prediction.

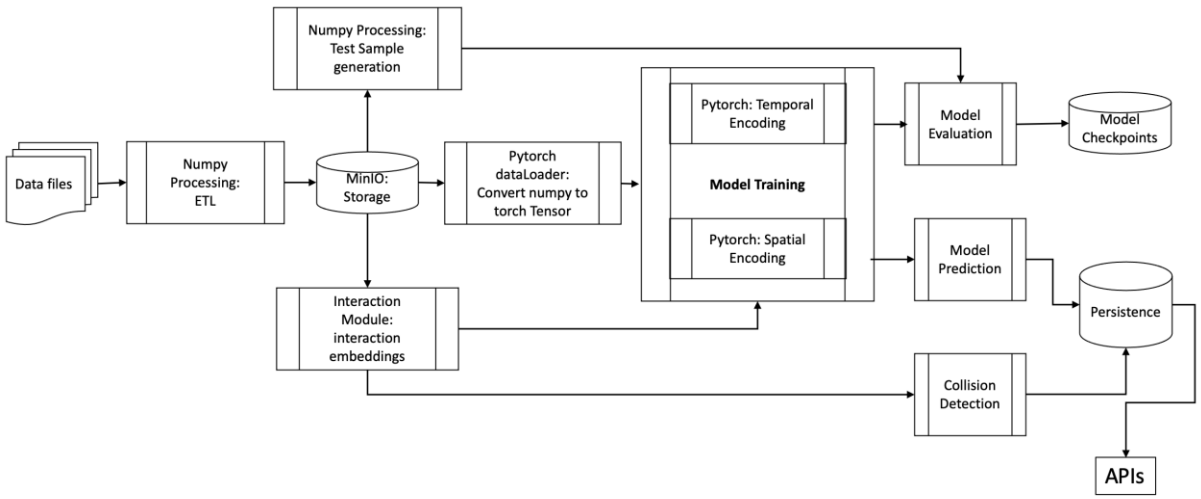


Figure 38: Trajectory prediction virtualized pipeline

The pipeline starts off with reading data files containing pedestrian trajectories. It is important to note that models work on a dataset that contains the series of coordinates of pedestrians

in a 2D plane. The first action in the ML pipeline is to read trajectories as sequences and that is where “Numpy processing” is utilized as per generic ML pipeline component. Then specific to this pipeline are two processes:

1. Split the trajectories into train and test samples.
2. Extract interactions between trajectories to model the behaviours of pedestrians

The data files are read and trajectory sequences are generated using *numpy* [38]. The observed trajectories are stored in MinIO bucket [21] for later usage and reduce the cost of executing same operation. The trajectory’s building operation has *traj_seq_len* as input parameter. This operation of reading data, constructing trajectories, and storing them is managed by an Argo workflow named as *Traj_etl*. Argo workflow [39] is an open-source container native workflow engine for orchestrating parallel jobs on Kubernetes. It enables the creation, management, and monitoring of data processing or ML models training tasks using DAGs.

The DAG definition in Argo is as simple as defining a sequence of tasks that are to be executed as shown in Figure 39.

```
14        dag:
15          tasks:
16            - name: read
17              template: read-template
18            - name: construct
19              arguments:
20                artifacts:
21                  [
22                    {
23                      name: read-result,
24                      from: "{{tasks.read.outputs.artifacts.read-result}}",
25                    },
26                  ]
27              template: transform-template
28              dependencies: [read]
29            - name: load
30              arguments:
31                artifacts:
32                  [
33                    {
34                      name: transformed-data,
35                      from: "{{tasks.construct.outputs.artifacts.transformed-data}}",
36                    },
37                  ]
38              template: load-template
39              dependencies: [construct]
```

Figure 39: Argo DAG for Trajectories Extraction-Transformation-Loading (ETL)

In Figure 39, a DAG is defined in Argo workflow, this DAG reads trajectory data from files, constructs trajectories and loads them into a MinIO bucket. The task in Argo workflow is a

described through a deployment *template*, that can of type *docker container*, *DAG*, *script*, *steps*, *resources*, or *suspend*. The templates for each task in the proposed workflow are defined as shown in Figure 40.

```
      - name: read-template
        container:
          image: locus/read
          command: [python, read.py]
        outputs:
          artifacts:
            - name: read-result
              path: /traj_data_np.pickle
      - name: construct-template
        inputs:
          artifacts:
            - name: read-result
              path: /traj_data_np.pickle
        container:
          image: locus/construct
          command: [python, construct.py]
        outputs:
          artifacts:
            - name: transformed-data
              path: /transformed_data.pickle
      - name: load-template
        inputs:
          artifacts:
            - name: transformed-data
              path: /transformed_data.pickle
        container:
          image: locus/load
          command: [python, load.py]
```

Figure 40: Argo Task templates

Numpy objects are serialised into a MinIO bucket as shown in Figure 41.

```
minioClient = Minio(endpoint = endpoint, access_key = minio_access_key, secret_key = minio_secret_key)
minioClient.fput_object(
    bucket_name='locus_data',
    object_name='traj_seq.pickle',
    data = io.BytesIO(pickle.dump(traj_seq)),
    file_path='locus_data/traj_seq.pickle'
    )
```

Figure 41: Definition of numpy objects serialization

Once the pipeline is defined in Argo Workflow template, it composes containers as nodes in a DAG which can be fully automated and executed upon the trigger of an event. For model serving, Seldon core pipeline follows the deployment of ML pipeline as discussed in D5.3 [1], and models are served via Seldon [20]. Argo workflows can be defined programmatically using a Python library Hera-workflows [40]. In the Figure 42 , we show the same Trajectories Extraction-Transformation-Loading (ETL) workflow in Python. The task definition in python and computation graph is parallelized.

```
ws = WorkflowTemplateService(host="https://localhost:2746", verify_ssl=False, token=TOKEN)
w = WorkflowTemplate("ETL", ws, namespace="argo")

extract_task = Task("read", extract, image="locus/read:v2",
                    output_artifacts = [OutputArtifact(name="RawData", path="/data/raw_data.csv")])

transform_task = Task("transform", generate_expectations, image="locus/transform:v1",
                    input_artifacts = [InputArtifact(name="RawData", path="/data/trajectories.csv", from_task="read", artifact_name="Trajs")])

store_task = Task("store-train-data", store_train_data, image="locus/transform:v1",
                    input_artifacts = [InputArtifact(name="RawData", path="/data/trajectories.csv", from_task="read", artifact_name="Trajs")])

extract_task >> transform_task
extract_task >> store_task
transform_task >> store_task

w.add_tasks(extract_task, transform_task, store_task)
w.create(namespace="argo")
```

Parallel tasks of storing raw as well as transformed data

Figure 42: Defining Argo tasks programmatically

4.1.1 Event based Pipeline Trigger

The deployed pipeline so far requires a manual trigger. One simple method to automate the pipeline is to run it as Cron job if ETL is required in a scenario where new data is arriving periodically and we are sure that new data will be available before running the ETL pipeline.

To setup Argo Workflow as Cron job the command in Figure 43 is used.

```
faisal@mlops1:~$ vi argoworkflow.yaml
faisal@mlops1:~$ argo -n argo cron create argoworkflow.yaml
Name:                          etl-cron-wf-wlwct
Namespace:                     argo
Created:                       Wed Jul 06 10:02:55 -0700 (now)
Schedule:                      0 1 * * *
Suspended:                     false
StartingDeadlineSeconds:       0
ConcurrencyPolicy:             Allow
NextScheduledTime:             Wed Jul 06 18:00:00 -0700 (7 hours from now) (assumes workflow-controller is in UTC)
faisal@mlops1:~$ kubectl -n argo port-forward deployment/argo-server 2746:2746
Forwarding from 127.0.0.1:2746 -> 2746
Handling connection for 2746
Handling connection for 2746
Handling connection for 2746
```

Figure 43: Argo workflow as Cron job

However, certain scenarios in individual mobility use-cases require data processing and model training as a response to a specific event. For example, trajectory prediction service may require that ETL is executed whenever there is new type of trajectories in the data source. In this scenario, a webhook event source can be defined in Argo template and ETL workflow is submitted anytime the webhook receives a post request. The main components of an event-driven pipeline in Argo are event source, sensor, and trigger. An event source defines the configuration required to consume events from external sources such as pub/sub, webhooks etc. Figure 44 shows the template for defining an event source. It only allows POST HTTP requests and a unique port is defined for each webserver.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: EventSource
metadata:
  name: webhook
spec:
  service:
    ports:
      - port: 12000
        targetPort: 12000
  webhook:
    # event-source can run multiple HTTP servers. Simply define a unique port to start a new HTTP server
    retrain:
      # port to run HTTP server on
      port: "12000"
      # endpoint to listen to
      endpoint: /retrain
      # HTTP request method to allow. In this case, only POST requests are accepted
      method: POST
```

Figure 44: Event Source definition in Argo

With event source defined, Argo Sensor [41] template is used to define inputs and triggers. It will listen to events on eventbus and act as event manager to resolve and execute triggers.

In Figure 45, a given configuration shows the triggers and submits ETL pipeline on an event of POST request hitting "/retrain" entrypoint of the trajectory model training pod.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Sensor
metadata:
  name: webhook
spec:
  template:
    serviceAccountName: argo
  dependencies:
    - name: train-hook
      eventSourceName: webhook
      eventName: retrain
  triggers:
    - template:
        name: webhook-workflow-trigger
        k8s:
          operation: create
          source:
            resource:
              apiVersion: argoproj.io/v1alpha1
              kind: Workflow
              metadata:
                generateName: webhook-
                namespace: argo
              spec:
                entrypoint: retrain
                templates:
                  - name: retrain
                    steps:
                      - - name: template-run
                          templateRef:
                            name: locus-etl
                            template: locus-etl
```

Figure 45: Event Sensor Argo template

With events, our complete trajectory forecasting pipeline in Argo is shown in Figure 45.

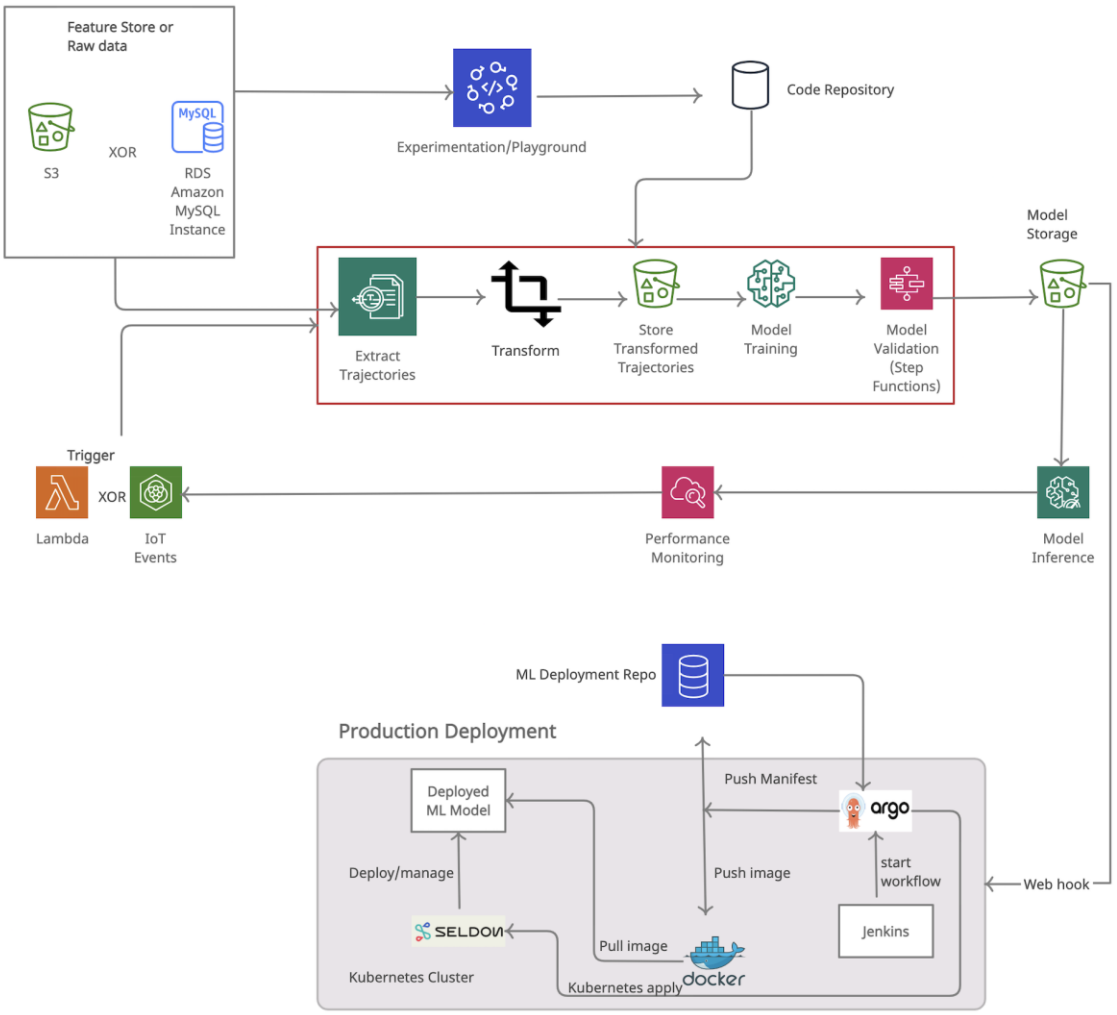


Figure 46: A complete model training and deployment framework for trajectory prediction service

The top part of is ETL, model training part for experimentation and validation of trajectory prediction models. The bottom part shows production-level setup with Argo as tasks orchestrator, Jenkins as CI/CD, and Seldon for model deployment. The persistent module is based on MinIO to store raw data as well as transformed data in separate buckets. The trained ML models that become production-ready are stored in in MinIO based model storage. The events-based trigger can be any IoT sensor or process with Argo definitions.

4.2 Transportation optimization based on the identification of traffic profiles

Figure 47 graphically describes the Virtualized ML Pipeline for the UC on Transportation optimization based on the identification of traffic profiles, based on the ML work done in WP5 and related to the LOCUS PoC#3.

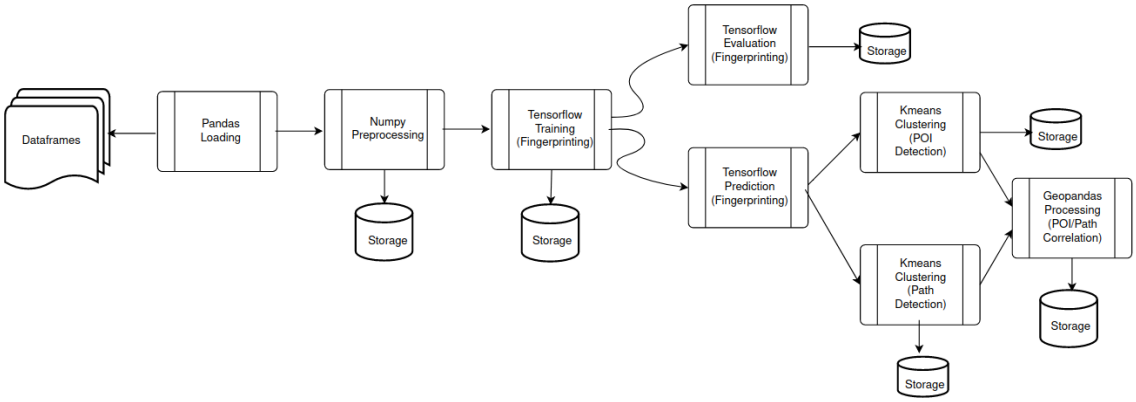


Figure 47: ML Pipeline for the UC on Transportation optimization based on the identification of traffic profile

More specifically, the pipeline presented ingests dataframes stored in table format in the LOCUS Persistence module. The next two boxes represent the prerequisite steps of loading and pre-processing the data before feeding them to the first service model, namely the training part of "Fingeprinting" (LEN function). The flow is then split in two directions. The first one assures that the Fingeprinting model has been successfully trained and can provide accurate location predictions (model evaluation process). In the second one, the Fingeprinting Prediction Service will be called to provide new positioning estimations. These will be consumed by two separate but related services, one that will detect commons POIs (POI Detection) and one that will detect common Paths (Path Detection). Those objects (Paths/POIs) will be saved as geometric objects (polygons for POIs and curves for Paths). The final service model will correlate those extracted sub-regions in real-time or near real-time, by finding geometric intersections between the different objects.

Every Service Model has a separate communication channel with the LOCUS Persistence Schema. In that way, it easy to return to "previous" versions of the service and re-start the procedure at any time from any pipeline node.

Lastly, the geo-correlation service can be used to provide spatial analytics and expand the map of the area with network or other data available.

4.3 Crowd mobility

Figure 48 illustrates the logical flow (pipeline) of the data flow for the crowd mobility analytics, in particular for the Group-In system, which realizes people group inference from wireless signals. Initially, the data that are collected from sensors (e.g., Raspberry PI with Bluetooth receiver antenna) and placed in a NoSQL key-value database by the mobile devices (i.e., Bluetooth low energy beacons). The objects are then pre-processed in Python using the numpy library [38]. The pre-processing includes functions for min-max normalization, data

sampling, and fingerprinting. After the pre-processing, the training phase considers a small portion of the dataset for finding the best parameters. The selected parameters are stored in the storage. Both training and inference goes through the illustrated pipeline, expect the latest long-term clustering, which is applied only in the inference phase (not during the training period), as this component relies on the accurate predictions.

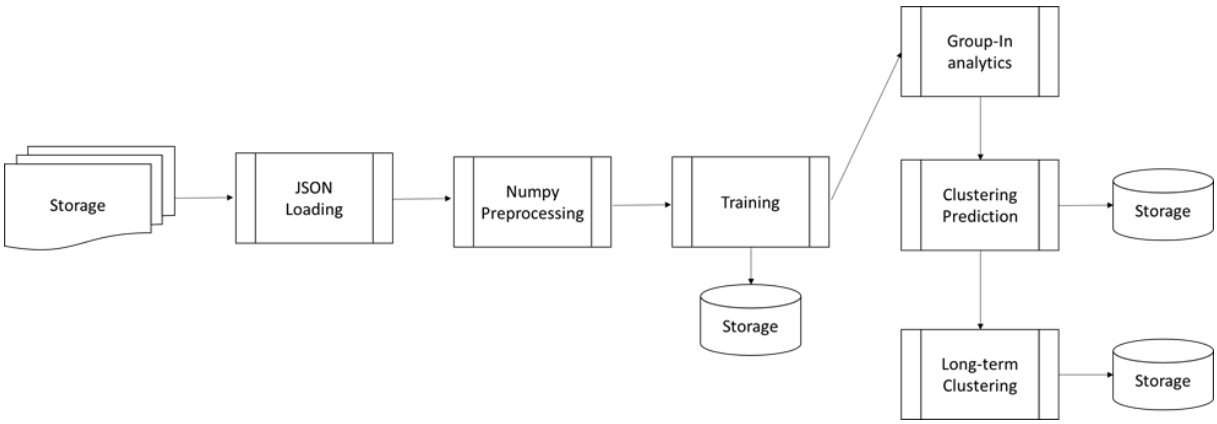


Figure 48: Crowd mobility - Logical flow of the functions in the Group-In analytics for people group inference

The trained parameters define the clustering parameters for the DenGraph [53], Highly Connected Subgraphs (HCS) [54], and MaxClique [55] clustering algorithms. These algorithms leverage graph model that are created during the pre-processing phase. Group-In analytics uses wireless fingerprints that are generated in the pre-processing phase for 1) analysing the fingerprints (e.g., data aggregation) and 2) creating a centralized or decentralized graph model (upon developer choice). After these initial analytics, the graph model is passed to the clustering prediction where the above listed clustering algorithms are applied. The prediction results are saved in the persistent storage. Lastly, through the application of the inference (predictions) through a longer time period, the results are statistically analysed in the long-term clustering. The results of the long-term clustering are saved in the persistent storage.

The functionalities listed in the pipeline are developed in Python and leverage Python libraries such as numpy [38], scikit-learn [42], network [43], Flask [8], CouchDB [44]. The Group-In analytics implement the algorithms described in [45] in the Python language.

4.4 Context-Aware flow monitoring and crowd mobility patterns

In D5.2 [4] the Kubeflow ML pipeline for the Open Street Map Context-Aware NSE-UC1 flow monitoring service and crowd mobility patterns functionality (Figure 49) has been described resulting in an improvement of the trained models.

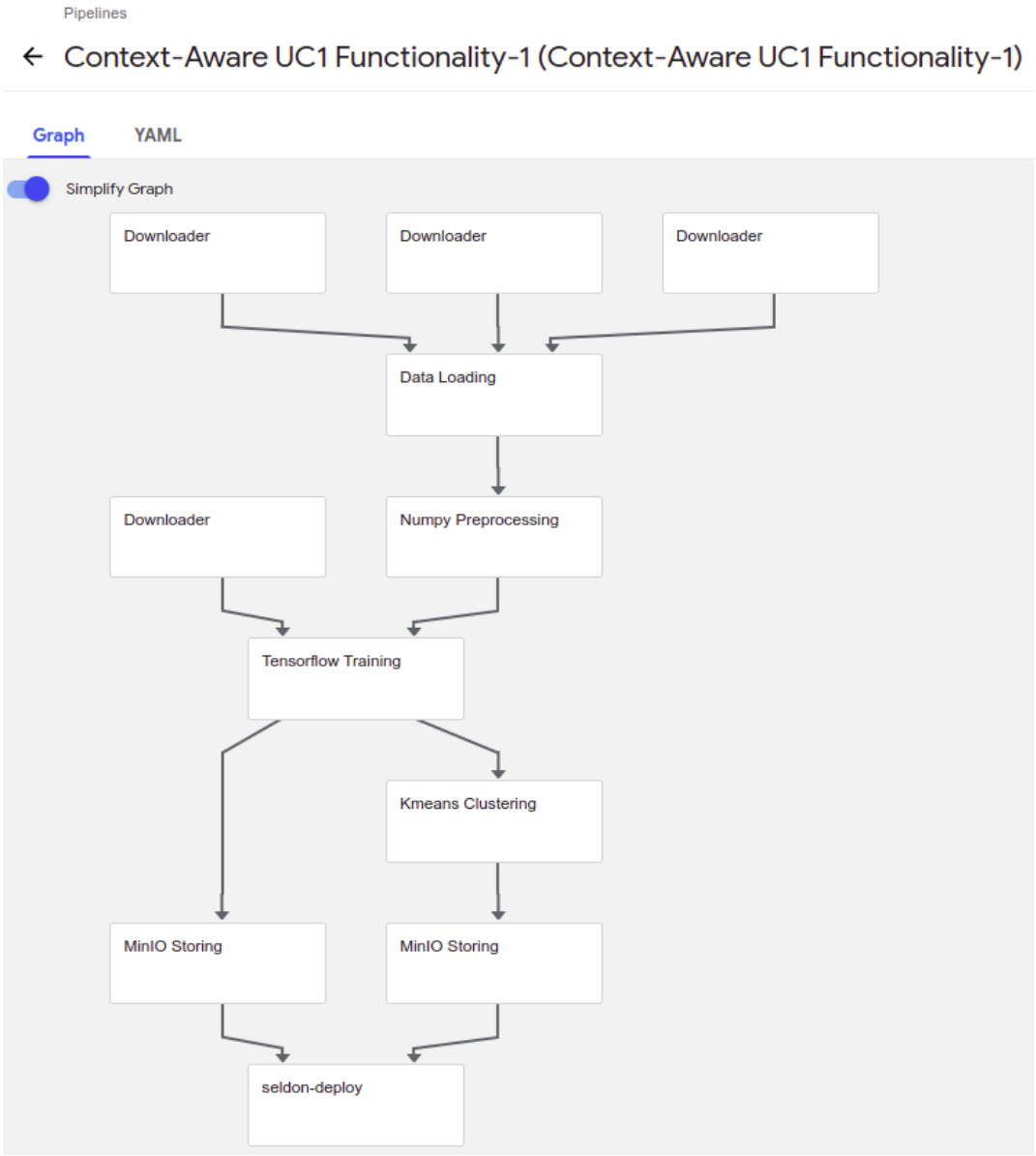


Figure 49: Context-Aware virtualized machine learning pipeline

Once the above-mentioned pipeline has been on-boarded on Kubeflow it can be executed from the Kubeflow UI. Figure 50 shows the execution of the context-aware ML pipeline with all its steps successfully deployed and executed. The first step of the pipeline, Models Keeper, replaced the three-downloading step of the previous version of the Machine Learning pipeline; this step is simply a container that includes all the datasets used in the pipeline to train the Encoding (Tensorflow Training step) and Clustering (K-Means Clustering step) models in order to avoid downloading the datasets in every new execution of the pipeline but only pull the Models Keeper container once.

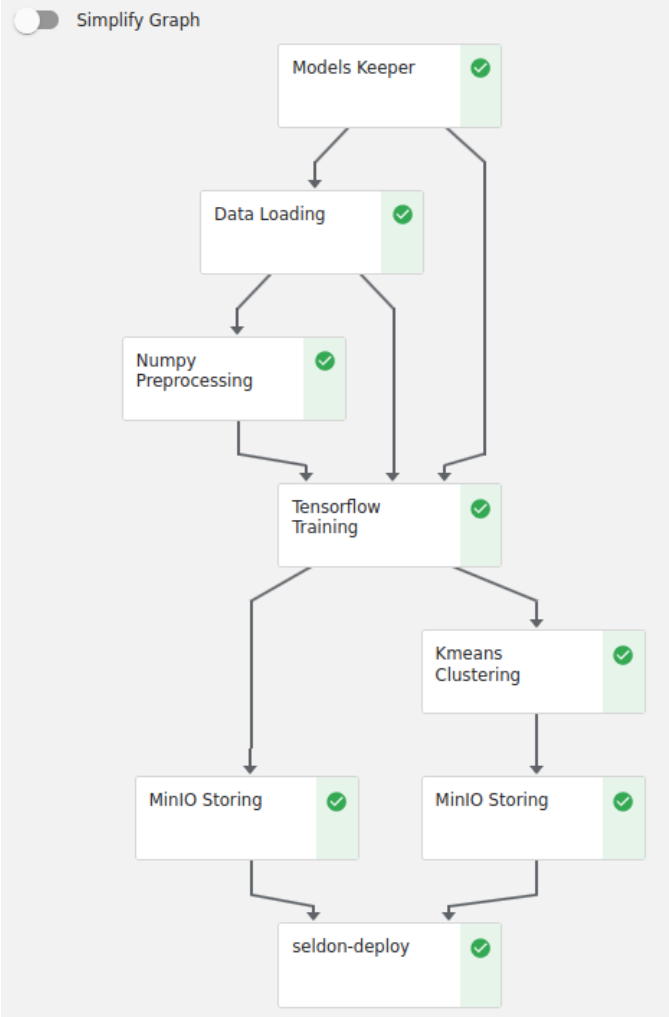


Figure 50: Execution of Context-Aware Machine Learning pipeline

Once the last step of the pipeline is terminated and the models-graph deployed, the REST predict endpoint exposed by the Seldon deployment can be used to make predictions on new data triggering the predict functions of the Encoder and Clustering models as already demonstrated in D5.3 [1] and D5.2 [4]. Figure 51 shows the result of a prediction request.


```
5829438,0.46819672],[0.345243394,0.142136589,-0.344735861,-0.127340958,-0.73982203,0.486229658,0.16273497,0.528077781],[0.445395827,0.267573059,-0.609062731,-0.46
853736,-0.156809673,0.424129218,0.208528563,-0.0051840283],[0.490661561,0.599818408,-0.807533,-0.838319361,0.868506312,0.209795952,0.229144722,-0.47390151],[0.591
397643,0.486318022,-0.776834607,-0.752057076,0.538626552,0.384484559,0.310607076,-0.198757619],[0.584975421,0.439159,-0.767213345,-0.730630755,0.44532305,0.388851
643,0.312346399,-0.106764704],[0.432795763,0.217583194,-0.74906528,-0.587588131,0.266335338,0.186078414,0.12635532,0.0349898934],[0.591397643,0.486318022,-0.77683
4607,-0.752057076,0.538626552,0.384484559,0.310607076,-0.198757619],[0.647930145,0.317022622,-0.807849169,-0.665944815,-0.238656551,0.640977502,0.287625313,0.2125
68939],[0.345243394,0.142136589,-0.344735861,-0.127340958,-0.73982203,0.486229658,0.16273497,0.528077781],[0.345243394,0.142136589,-0.344735861,-0.127340958,-0.73
982203,0.486229658,0.16273497,0.528077781],[0.380877256,0.144690394,-0.474078476,-0.293615162,-0.510032415,0.40428,0.161411613,0.340217054],[0.591397643,0.4863180
22,-0.776834607,-0.752057076,0.538626552,0.384484559,0.310607076,-0.198757619],[0.636355162,0.414188653,-0.788706422,-0.733905375,0.227401555,0.494003713,0.292509
526,-0.0679006279],[0.636355162,0.414188653,-0.788706422,-0.733905375,0.227401555,0.494003713,0.292509526,-0.0679006279],[0.636355162,0.414188653,-0.788706422,-0.
733905375,0.227401555,0.494003713,0.292509526,-0.0679006279],[0.358452529,0.137853414,-0.394486904,-0.189030766,-0.688927412,0.480508864,0.15829438,0.46819672],[0
.617341161,0.504563,-0.778098345,-0.770271063,0.467311531,0.436166286,0.322587311,-0.207694754],[0.591397643,0.486318022,-0.776834607,-0.752057076,0.538626552,0.3
84484559,0.310607076,-0.198757619],[0.345243394,0.142136589,-0.344735861,-0.127340958,-0.73982203,0.486229658,0.16273497,0.528077781],[0.334435105,0.130266964,-0.
335578144,-0.115670696,-0.730015278,0.448708743,0.153665051,0.549961567],[0.170919612,-0.0874728858,-0.28245759,-0.0359037779,-0.68210274,-0.00246358849,-0.090914
1824,0.638554335],[0.499832213,0.183108598,-0.659458101,-0.484241664,-0.404599488,0.530150414,0.171226665,0.191185877],[0.624560714,0.51642096,-0.782713175,-0.776
125789,0.490041852,0.440352976,0.327450395,-0.219437212],[0.591397643,0.486318022,-0.776834607,-0.752057076,0.538626552,0.384484559,0.310607076,-0.198757619]]},"m
eta":{"requestPath":{"encoder":"seldonio/tfserving-proxy:1.6.0"}}}
```


Figure 51: Context-Aware Machine Learning Pipeline Prediction request result

4.5 Other implementations

Beyond the NSE-UCs, ML pipelines have been developed and used in LOCUS for other types of services and functions as well, such as those for the localization enablers and the smart network management functionalities.

The following section describes how a virtualized ML pipeline has been developed for the Soft-Information (SI) -based localization for 5G networks.

4.5.1 SI-based Localization Service

SI-based localization for 5G networks has been developed in the context of WP3 and relies on ML models, referred to as generative models, to provide improved localization accuracy compared to classical localization algorithms (see research results reported in the deliverables D3.1 [46], D3.2 [48], D3.3 [47], D3.4 [49], and D3.7 [50]). In D6.2 [51], the implementation of SI-based localization as a network service in the LOCUS platform is presented. As reported in D6.2, the generative models used by SI-based approach are trained offline and embedded directly in the Docker image composing the network service deployed on the LOCUS platform. This allows for fast evaluation of the generative models and reduced latency in the estimation of the position. However, this approach is not suited for production environments, where frequent retraining of the generative models is required. In WP5 deliverables D5.1 [3], D5.2 [4], and D5.3 [1], virtualization of a ML pipeline related to NSE UC1 flow monitoring service and crowd mobility patterns functionality was presented. Leveraging the same Open-Source frameworks (i.e., Kubeflow [19] and Seldon [20]), virtualization of the ML pipeline and generative model serving is currently being tested locally and full integration with the LOCUS platform is expected to be performed in the context of WP6. Currently, Kubeflow is used to orchestrate the execution of the SI ML pipeline composed as in the following.

1. Data download: download of the training data from an external storage, implemented as a reusable component;
2. Pre-processing: scaling and normalization of the downloaded training data, implemented as a reusable component;
3. Generative model training: training of the generative model using Scikit-learn Python library and storage of the model in the LOCUS persistence module, implemented as a reusable component; and
4. Generative model serving: serving of the generative model leveraging the Kubeflow SDK and using a reusable model server via Seldon core Python wrapper fetching from the LOCUS persistence module.

Once the ML pipeline is executed, the SI-based localization service can interrogate the reusable server via a HTTP POST request each time a new set of measurements is available.

Figure 52 shows how the SI-based localization service, Kubeflow pipeline, and Seldon server interact among each other and with relevant LOCUS platform components. In particular, SI-based localization service reads the measurements published on the message bus exchange *Clear_Data*. Then, the service interrogates the reusable server, which evaluates the generative model based on the measurement values. Lastly, the response of the server is used to estimate the position of the device that originated the new measurements and publish it on the *PositionAndNetworkInfo* exchange.

When data fusion is considered, SI-based localization requires a different generative model for each measurement type used for providing the localization estimates [52]. In this context, the use of Kubeflow allows to reuse the same ML pipeline for each type of measurement considered, enabling efficient training and serving of multiple models using the same components.

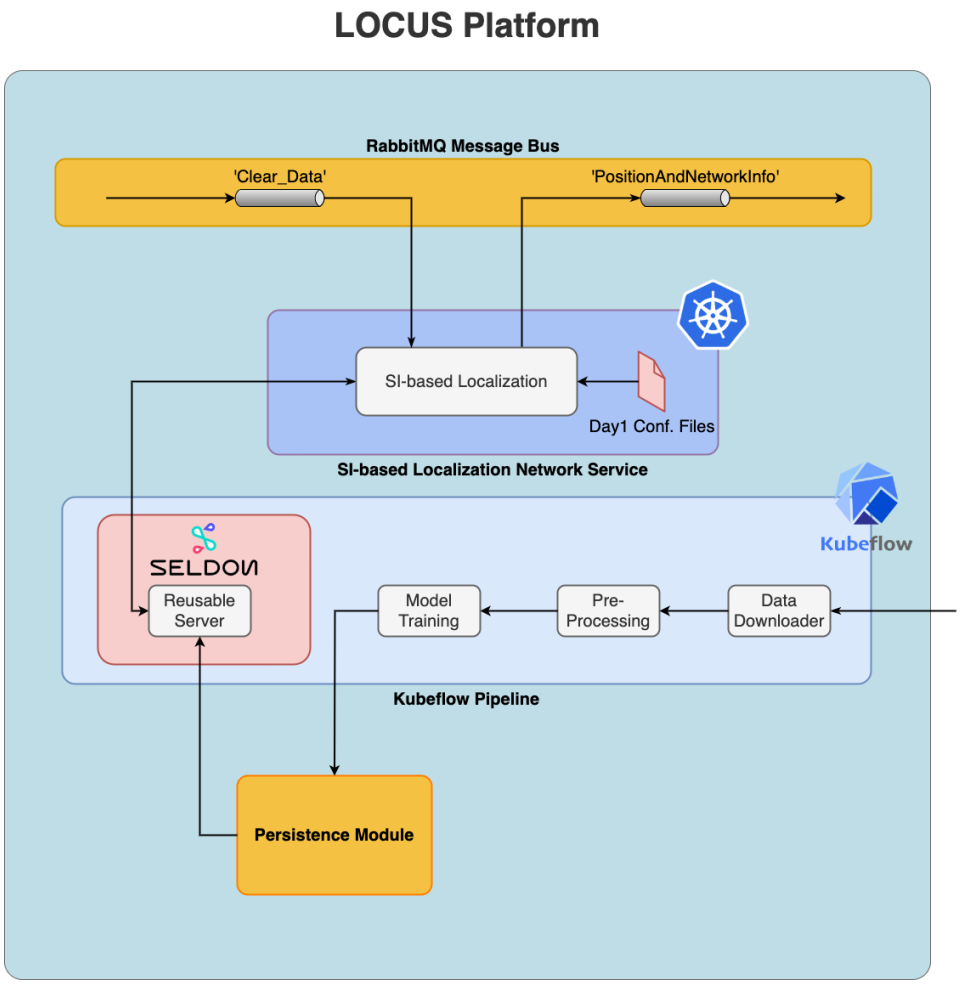


Figure 52: Overview of the SI-based localization service with the virtualized ML pipeline for generative model training and serving.

5 Conclusions

This deliverable presented the software prototype of the LOCUS localization analytics as a service solution, together with its deployment in the project testbed deployed at the OTE premises. In particular, the implemented prototype is composed of two main complementary components: the LOCUS API layer and the LOCUS platform control.

The LOCUS API layer represents the northbound interface of the LOCUS platform. It is implemented leveraging on existing opensource tools for analytics and data API consumption through an advanced API gateway and service discovery features (i.e., Consul, Swagger, Zuul). The API gateway and service discovery features are integrated with a custom API catalogue and a service subscription module that allows external applications and users to discover the available analytics services and activate them on-demand. On the other hand, the LOCUS platform control allows to decouple the API layer functionalities and the analytics services exposed towards external entities from the complexity of internal analytics functions management and execution, in terms of deployment as NFV virtualized functions, data operations and constraints. This has been implemented as a combination of software tools, which integrates custom applications for analytics service coordination, and relies on existing opensource tools for analytics service and Machine Learning pipeline management and virtualization (such as Apache AirFlow and Kubeflow).

Moreover, the deliverable also reported on the implementation of the LOCUS data platform prototype, which combines a solution based on RabbitMQ for real-time data streams exchange with a data persistence module based on Hadoop, Hive and Trino. These two solutions enable the analytics services and functions to communicate and exchange data following different paradigms, while supporting real-time and batch processing.

As part of the next steps, the activities related to the LOCUS localization analytics as a service solution are expected to be followed-up in the last phase of the project, in terms of support to the final LOCUS PoCs developments, integration and demonstration in WP6. This is planned for both the API layer and platform control components, with the aim of evaluating their performances and integrating the final PoC functionalities in the LOCUS platform deployed at the OTE premises.

References

[1] H2020 LOCUS Deliverable D5.3 “Design of the localization & analytics as a service solution”

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

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

[4] H2020 LOCUS Deliverable D5.2 “Design and implementation of virtualization technologies and pattern recognition mechanisms for physical analytics”- final version”

[5] H2020 LOCUS Deliverable D6.3 “Assessment of applications integrated with geolocation mechanisms”

[6] Keycloak, Open Source Identity and Access Management, https://www.keycloak.org/

[7] Java Spring Framework, https://spring.io/

[8] Python Flask, https://flask.palletsprojects.com/en/2.1.x/

[9] Java Spring Security, https://spring.io/projects/spring-security

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

[11] Python FastAPI, https://fastapi.tiangolo.com/

[12] Swagger: API Documentation & Design Tools for Teams, https://swagger.io/

[13] Zuul, https://github.com/Netflix/zuul

[14] H2020 LOCUS Deliverable D4.4 “Implementation of the Virtualization platform for network control and management: final version”

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

[16] EuCNC & 6G Summit conference, https://www.eucnc.eu/

[17] Consul: Identity based networking, https://www.consul.io/

[18] Apache Airflow, https://airflow.apache.org/

[19] Kubeflow, https://www.kubeflow.org/

[20] Seldon, https://www.seldon.io/

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

[22] Minio, https://min.io/

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

[24] Charmed Kubeflow guide, https://charmed-kubeflow.io/docs/quickstart

[25] TensorFlow, https://www.tensorflow.org/

[26] PyTorch, https://pytorch.org/

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

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

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

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

[31] Apache Kafka, https://kafka.apache.org/

[32] Openstack, https://openstack.org/

[33] Apache Hadoop, https://hadoop.apache.org/

[34] Apache Hadoop YARN, https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

[35] Apache Hive, https://hive.apache.org/

[36] Trino, https://trino.io/

[37] Apache Zeppelin, https://zeppelin.apache.org/

[38] Numpy library, https://numpy.org/

[39] Argo Workflow, https://argoproj.github.io/argo-workflows/

[40] Hera workflows, https://github.com/argoproj-labs/hera-workflows

[41] Argo sensor, https://argoproj.github.io/argo-events/concepts/sensor/

[42] scikit-learn library, https://scikit-learn.org/

[43] networkx library, https://networkx.org/

[44] CouchDB library, https://couchdb-python.readthedocs.io/

[45] Solmaz, Gürkan, et al. "Group-in: Group inference from wireless traces of mobile devices." 2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). IEEE, 2020.

[46] H2020 LOCUS Deliverable D3.1, "5G Based Preliminary Localization Solutions"

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

[48] H2020 LOCUS Deliverable D3.2, “5G Based Advanced Localization Solutions” D3.2, 2021.

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

[50] H2020 LOCUS Deliverable D3.7, “5G Based Localization Solutions, Final Version”

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

[52] A. Conti, S. Mazuelas, S. Bartoletti, W. C. Lindsey, and M. Z. Win, “Soft information for localization-of-things,” Proc. IEEE, vol. 107, no. 11, pp. 2240–2264, Nov. 2019

[53] Falkowski, Tanja, Anja Barth, and Myra Spiliopoulou. "Dengraph: A density-based community detection algorithm." IEEE/WIC/ACM International Conference on Web Intelligence (WI'07). IEEE, 2007.

[54] Erez Hartuv and Ron Shamir. 2000. A Clustering Algorithm Based on Graph Connectivity. Inform. Process. Lett. 76, 4-6 (2000), 175–181.

[55] Kazuhisa Makino and Takeaki Uno. 2004. New Algorithms for Enumerating All Maximal Cliques. In Scandinavian Workshop on Algorithm Theory. 260–272.