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 D5.3

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

Deliverable Type:	R
Dissemination Level:	Public
Contractual Date of Delivery to the EU:	30/04/2021
Actual Date of Delivery to the EU:	11/05/2021
WP contributing to the Deliverable:	WP5 – Localization & Analytics for New Services
Editor(s):	**VIAVI**, Takai Eddine Kennouche
Author(s):	**VIAVI**, Takai Eddine Kennouche
	Nextworks, Giacomo Bernini, Michael De Angelis
	IBM, Joseph Antony
	NEC, Gurkan Solmaz, Giuseppe Siracusano
	Incelligent, Aristotelis Margaris, Yannis Filippas
	SAMSUNG, Tomasz Mach
	CNIT, Andrea Conti, Domenico Garlisi

	TEI, Stefano Stracca, Marzio Puleri
Internal Reviewer(s):	**CNIT**, Stefania Bartoletti, Flavio Morselli, Nicola Blefari Melazzi
Short Abstract:	This report describes a microservices-based solution for Location & Analytics as a Service. It describes a catalogue services provided by LOCUS devised to enable versatile analytics pipelines in support of verticals, with examples from use cases studied in WP5. The report describes the data movement/persistence approach as well, and the set of REST/Intent based APIs to enable interaction with the LOCUS Solution.
Keyword List:	Analytics, Microservices, Virtualization, intent-based, Localization, NFV, Kubernetes

Executive Summary

One of the goals of the LOCUS project is to provide support for location-based analytics and vertical applications. To achieve this, LOCUS aims at exposing a set of functionalities and services on top of a flexible and scalable architecture. It relies on industry-proven API, virtualization and data management technologies, and integration with state-of-the-art 3GPP and non-3GPP localization technologies, and ML-based techniques.

This deliverable describes three key aspects of the proposed Location & Analytics as a Service Solution. First, it provides an analysis of the use cases studied in WP5 in terms of functionalities, which can be implemented as a set of pipelines of decoupled microservices. This decomposition allows the definition of a catalogue of services that LOCUS can provide to support a variety of applications and target a wide range of scenarios. In addition to that, details on service orchestration, virtualization and ML optimization are provided.

Second, this deliverable describes the data movement and persistence necessary for efficient and scalable data management service interaction with data, in addition to integration with external data sources, based on Kafka and modern database technologies.

Finally, this deliverable provides details on the API gateway that exposes system capabilities to external applications. It also includes a discussion of an Intent-Based API approach to expose LOCUS capabilities in a more natural-language-oriented interface, which facilitates interaction with system capabilities and abstracts system complexity from external users.

VERSION CONTROL TABLE			
VERSION N.	**PURPOSE/CHANGES**	**AUTHOR (S)**	**DATE**
0.1	Initial ToC	Takai Eddine Kennouche	29/10/2020
0.2	Updated ToC	Takai Eddine Kennouche	08/02/2021
0.3	Introduction	Takai Eddine Kennouche	06/04/2021
0.4	Virtualization of services/pipelines	Giacomo Bernini	06/04/2021
0.5	Data Virtualization	Takai Eddine Kennouche	12/04/2021
0.6	ML Optimization	Giuseppe Siracusano	15/05/2021

0.7	Intent Based API REST API	Takai Eddine Kennouche Giacomo Bernini	
0.8	details on Pipeline LCM & Orchestration per UC pipelines	Aristotelis Margaris Joseph Antony	22/04/2021
0.9	Revised APIs section. Edits/Corrections and references.	Giacomo Bernini Takai Eddine Kennouche	28/04/2021
1.0	Corrections and Integration of reviewers' suggestions.	Takai Eddine Kennouche.	10/05/2021
1.1	Final revision	Nicola Blefari Melazzi	11/05/2021

INDEX

EXECUTIVE SUMMARY **3**

LIST OF ABBREVIATIONS **7**

TABLE INDEX **9**

FIGURE INDEX **10**

1 INTRODUCTION **12**

1.1 DOCUMENT STRUCTURE 13

2 LOCUS ANALYTICS SERVICES **14**

2.1 SCENARIOS FOR LOCATION BASED ANALYTICS 14

2.2 ANALYTICS SERVICES SPECIFICATIONS 14

2.2.1 List of Analytics Services 14

2.2.2 LOCUS Analytic Services Template 15

2.2.3 Crowd Flow Monitoring using Trajectory forecasting and clustering 16

2.2.4 Group-In: to learn group mobility Characteristics 18

2.2.5 Vulnerable road user 19

2.2.6 Time to Collision as a service in V2X 21

2.2.7 Logistics in a seaport terminal using AGVs 23

2.2.8 Transportation optimization based on identification of traffic profiles 24

2.2.9 Positioning and Flow Monitoring for Controlling COVID-19 27

2.3 MICROSERVICES AND FUNCTIONS CATALOGUE 28

2.4 LOCUS PRIVACY SERVICES 29

2.5 VIRTUALIZATION OF ANALYTICS SERVICES AND PIPELINES 32

2.5.1 Seldon for model and graph serving 33

2.5.2 Kubeflow Pipelines 42

2.5.3 Next steps 49

2.6 OPTIMIZATION OF ML MODELS FOR VIRTUALIZATION 50

2.6.1 Background on Neural Network Processing 51

2.6.2 SOL AI acceleration middleware 52

2.6.3 SOL Operations & Optimizations 53

3 LOCUS DATA PLATFORM TECHNOLOGIES & VIRTUALIZATION **58**

3.1 SCOPE FOR LOCUS DATA PLATFORM TECHNOLOGIES & VIRTUALIZATION 58

3.2 LOCUS PERSISTENCE MODULE – DATA LAKE 58

3.3 LOCUS STREAMING MODULE - DATA MOVEMENT 62

3.3.1 Pipelines and Data Movement 62
3.3.2 The Message Broker 64
3.3.3 Data Collection 67
3.3.4 Multi-tenant deployment 68
4 LOCALIZATION-BASED ANALYTICS AS A SERVICE API 70
4.1 CONCEPT AND ARCHITECTURE 71
4.1.1 API Gateway components 73
4.1.2 Pipeline Orchestrator components 77
4.2 NORTHBOUND APIS OVERVIEW 83
4.3 INTENT-BASED API 84
4.3.1 Intent as a concept 84
4.3.2 Intent driven interactions with LOCUS Analytics Functions 86
5 CONCLUSIONS 93
6 BIBLIOGRAPHY 94

List of Abbreviations

ABBREVIATION	FULL NAME
5G	Fifth generation technology standard for cellular networks
AGV	Automated Guided Vehicle
AI	Artificial Intelligence
API	Application Programming Interface
CNF	Containerized Network Function
COVID-19	Coronavirus disease 2019
CRUD	Create, Read, Update, Delete
ETSI	European Telecommunications Standards Institute
FR	Functional Requirement
GRU	Gated Recurrent Unit
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IBA	Intent-Based API
JSON	JavaScript Object Notation
KFN	Kubernetes Network Function
KPI	Key Performance Indicator
LSTM	Long Short-Term Memory
ML	Machine Learning
NFV	Network Function Virtualization
NSD	Network Service Descriptor
NSE	New Services
MANO	Management and Orchestration
PCA	Principal Component Analysis
POI	Point of Interest
RAT	Radio Access Technology
REST	REpresentational State Transfer
SBA	Service Based Architecture
SBI	Service Based Interface
SDK	Software Development Kit

UC	Use Case
VDU	Virtual Deployment Unit
VNF	Virtual Network Function
VM	Virtual Machine
VRU	Vulnerable Road User
WiFi	Wireless Fidelity
WP	Work Package
WP5	Work Package 5

Table 1: Abbreviation List

Table Index

Table 1: Abbreviation List 8
Table 2: List of Analytic Services from WP5 15
Table 3: A general template to collect details about the various LOCUS Analytic Services. 15
Table 4: Crowd Flow Monitoring as a Service 17
Table 5: Group-In Service 19
Table 6: Status of NSE-UC3: VRU Clustering 20
Table 7: Time to Collision as a service in V2X analytics. 22
Table 8: Logistics in a Seaport terminal using AGVs 23
Table 9 - transportation optimization using traffic profiles. 25
Table 10: Flow Monitoring for Controlling COVID-19 as a Service 27
Table 11: Functions for privacy service present int the LOCUS catalogue 31
Table 12: API Catalogue Analytics Service information model 73
Table 13: Internal Analytics Function information model 80
Table 14: Internal Analytics Service information model 80
Table 15: IBA template for use cases 86
Table 16: NSE-UC1-Func1 Requirements for IBA 87
Table 17: NSE-UC2-Func1 Requirements for IBA 88
Table 18: NSE-UC3 Requirements for IBA 89
Table 19: NSE-UC4 Requirements for IBA 89
Table 20: NSE-UC5 Requirements for IBA 90

Figure Index

Figure 1: A generic ML-based pipeline for the analytic services 15
Figure 2: Pipeline for trajectory forecasting 17
Figure 3: Pipeline for trajectory clustering 17
Figure 4: Pipeline for Group-In Service 18
Figure 5: Pipeline for Time to Collision as a service in V2X 22
Figure 6: Pipeline for Logistics in a Seaport terminal using AGV 23
Figure 7: Pipeline for Traffic Profile Monitor Service 25
Figure 8: Flow monitoring for Controlling COVID-19 – Pipeline 27
Figure 9: Microservices and Functions Catalogue for Crowd Flow Monitoring Service (Section 0) 29
Figure 10: Flow diagram of the selected algorithm 32
Figure 11: Reference Machine Learning Pipeline 32
Figure 12: Reference Machine Learning Pipeline - Seldon Deployments 33
Figure 13: Minikube - Seldon environment 34
Figure 14: Single Model deployment logic 34
Figure 15: Reusable Model Servers (source: Seldon) 35
Figure 16: Non-reusable Model Servers (source: Seldon) 35
Figure 17: Non-reusable Model Server for Clustering Model 36
Figure 18: Requirements file for non-reusable clustering model server 36
Figure 19: Dockerfile for non-reusable clustering model server 37
Figure 20: Clustering model Kubernetes deployment file 38
Figure 21: POD with running containers for clustering model serving 38
Figure 22: Initial part of the output of the prediction request sent to the clustering model 39
Figure 23: Final part of the output of the prediction request sent to the clustering model 39
Figure 24: Graph deployment logic 40
Figure 25: Graph Kubernetes deployment file 40
Figure 26: POD with running containers for Encoder and Clustering models serving 40
Figure 27: http response of the prediction request to the graph [encoder + clustering] 41
Figure 28: Encoder and Clustering models graph logic 42
Figure 29: Reference Machine Learning Pipeline - Kubeflow Deployments 42
Figure 30: Create a reusable Kubeflow Pipeline component 43
Figure 31: MicroK8s - Kubeflow environment 43
Figure 32: Pandas loading script Dockerfile 44
Figure 33: Pandas loading Kubeflow Pipeline component definition 44
Figure 34: Kubeflow pipeline with five components loaded 45
Figure 35: input/output details of the tensorflow_training component 47
Figure 36: input/output details of the kmeans_clustering component 49
Figure 37: Deep Learning Framework common architecture 51

Figure 38: SOL AI acceleration middleware 52
Figure 39: SOL usage 54
Figure 40: SOL Operations 54
Figure 41: per-layer execution 55
Figure 42: DFP 55
Figure 43: SOL benchmark 56
Figure 44: Example of a Database-oriented flow 59
Figure 45: The LOCUS data schema as an interface point between functions 61
Figure 46: Pipelines Composition and Instantiation 63
Figure 47: Kafka powered distributed pipeline 64
Figure 48: Kafka Bus 66
Figure 49: Kafka-based service interaction 67
Figure 50: Kafka Connect 68
Figure 51: Kafka Multi-site deployment 69
Figure 52 Localization Analytics as a Service architecture 72
Figure 53: Internal Analytics Catalogue and API Catalogue onboarding approach 79
Figure 54: Intents lifecycle 86

1 Introduction

This deliverable delves into aspects of Localization & Analytics as a Service and provides a design to enable a wide range of location-aware verticals. The content of this deliverable is related to activities being conducted in Task 5.1 “Virtualization and ML Technologies for Location-based Services” and Task 5.2 “Virtualized platform for localization & analytics as a service”. The deliverable also complements ongoing WP2 activities in relation to the specification of the LOCUS architecture.

Deliverable D2.4 “System Architecture: Preliminary Version” has set the goal of relying on a Microservices architecture, to provide the flexibilities and scalability required by the use cases and scenarios targeted by the LOCUS project and described in previous deliverables (D2.1 provides a summary). This deliverable emphasizes this design choice, and with that said, it focuses on vertical enablement, and follows modern paradigms for ML/Analytics Pipelines, micro-services, APIs and data architecture, as will be specified in the following sections.

Deliverable D5.3 described a variety of UCs, which leverage AI/ML technologies for the consumption of geolocation information and the production of knowledge and insights related to mobility attributes and their interplay with the environment. This deliverable is a step further towards providing a solution that effectively enables said use cases and allows the generation of the desired value from geolocation information in all its forms and from all sources studied in the LOCUS project.

We aim at providing a modern solution for localization & analytics as a service that provides:

- Location transparency, i.e., functionalities provided by a particular service can be accessed without needing to know the actual physical location of said service. This allows for a distributed architecture where the physical resources of the platform can be efficiently utilized without disrupting operations.
- Implementation transparency, i.e., the access to a particular service is independent of its internal implementation. The service developers will have the freedom to use the best implementation tools for a particular service.
- Overall agility, with the ability to respond quickly to constant change in both business and technology. Related to the previous points, we would like to swiftly and seamlessly adopt new tools and libraries to implement services, without disrupting the choreography and operation of other system components.
- Overall scalability, all architectures can scale; the challenge is ease of scalability. Microservices-based architectures surpass monolithic and service-oriented architectures in terms of ease of scalability.

1.1 Document Structure

This deliverable contains five sections:

- Section 1 Introduces the deliverable and scope of its content.
- Section 2 presents the LOCUS Analytics Services. This section outlines the potential scenarios benefiting from location analytics and motivates the choice of pipelining and microservices-based architecture. In this section UCs will be discussed in terms of services and pipelines, in order to identify core functionalities and services to support all possible location-based vertical scenarios.
- Section 3 introduces the LOCUS Data Platform Technologies & Virtualization. This section delves into the backbone of the analytics solutions, the data architecture. It discusses a Kafka-based topology that targets a scalable edge-core deployment of a microservices architecture, with flexible and resilient decoupling of data movement, collection, persistence, and services.
- Section 4 describes the Location-based Analytics as a Service API. This section outlines the technological choices made to expose the LOCUS analytics services and pipelines to external systems. It discusses a RESTful API to provide fine-grained access to services and generated analytics. In addition, a concept for a high level and abstract Intent-Based API that aims at hiding the architecture's complexity from external systems, while allowing for a more natural and familiar interface to express the actual user intents behind the interaction with the LOCUS system.
- Section 5 provides final remarks.

2 LOCUS Analytics Services

2.1 Scenarios for Location Based Analytics

Location based analytics blends data from different sources with geo-spatial locations (geographic data) to provide context and derive valuable insights, and in turn helps at better decision making for businesses. Location analytics has been one of the fastest growing fields in recent times. Many businesses have successfully embraced location analytics to create ever more sophisticated and pragmatic scenarios. Some of the benefits of using location analytics in business scenarios include optimizing the business use cases based on location intelligence, effective customer targeting, increased marketing relevancy, long-term strategic and cost-effective planning, better marketing territory management, and even to create contingency plans to cope with the arrival of other business competitors.

LOCUS aims to offer localization, together with analytics, and their combined provision "as a service". The various functionalities developed in WP5 (detailed in the following sections) along with their localization analytics will be exposed to 3rd party verticals as analytic services. These LOCUS services can be leveraged to suit several business applications. For instance, the location intelligence derived from 'the crowd flow monitoring service' combined with real-time analytics can be leveraged in a retail application to understand visitor journeys i.e. to visualize how customers interact with a venue at a given time using real-time heat maps and dashboards, to serve customer targeted advertisements, to analyse customer loyalty and engagement to boost customer retention and conversion. If the locations are geo-fenced this service can be used to monitor the crowd movement, operations, and resources for surveillance purposes.

2.2 Analytics Services Specifications

This section elaborates on the analytics services that are developed based on the WP5 functionalities. To start with, a general ML-based pipeline and a template was defined to collect implementation details about each analytic service.

2.2.1 List of Analytics Services

The list of analytic services that are envisaged and to be provided from the WP5 are in Table 2.

Table 2: List of Analytic Services from WP5

WP5 Use Cases	Analytic Service
NSE-UC1	Crowd Flow Monitoring using Trajectory forecasting & clustering
NSE-UC2	Group-In to Learn group mobility characteristics
NSE-UC3	• VRU Clustering • Time to Collision (V2X Service)
NSE-UC4	Logistics in a seaport terminal using AGVs
NSE-UC5	Transportation optimization based on traffic profiles identification
NSE-UC6	Positioning and flow monitoring for controlling COVID-19

2.2.2 LOCUS Analytic Services Template

This section presents a general template (Table 3) for the various analytic services in LOCUS and a generic ML-based pipeline as a reference for the analytic services (Figure 1). The goal of this template and the generic pipeline is to provide guidelines for functional decomposition of the analytic services and collect details about the input data, input-output data handling, dependencies and requirements, the different analytic functions, and the various operations involved that will be useful for the virtualization and packaging of functions for each analytics service.

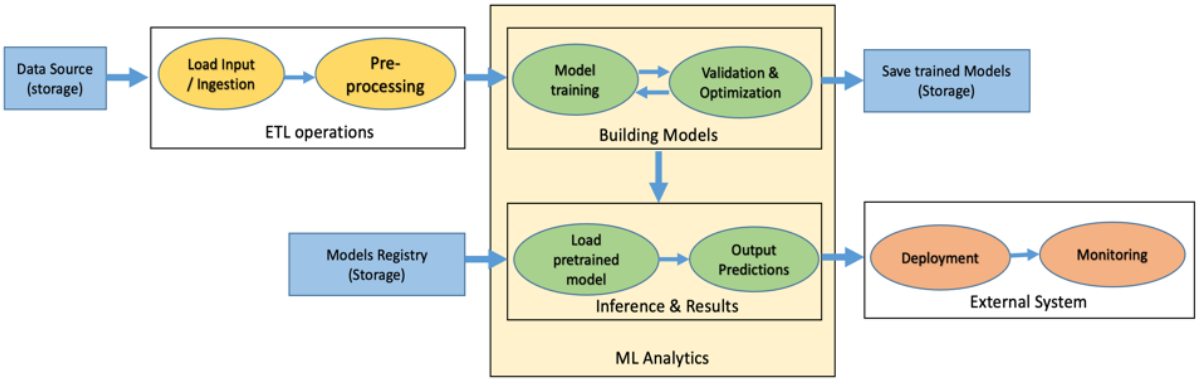


Figure 1: A generic ML-based pipeline for the analytic services

Table 3: A general template to collect details about the various LOCUS Analytic Services.

< Service Name >	
Pipeline	A figure representing the functional decomposition of the analytic services i.e., the different modules like input/output handling, data processing, analytic functions etc. in the

	implementation of the functionality. Figure 1 shows a generic ML-based pipeline as a reference for analytic services.
Input data	Data source, data handling & storage, data format, data fields, components, configuration parameters, dependencies etc.
Data pre-processing	Details about data-cleaning, data sampling, scaling & normalization & standardization, other transformation, data split into training & test, any other useful details for virtualization.
Building models/other analytics	ML algorithms used, model training details, hyperparameter optimization, performance evaluation and validation steps, any other implementation details
Inference and Results	Model selection, predictions, intended output, output data format, KPIs, any post-processing involved.
Deployment	Details about deployment, monitoring, and management.

2.2.3 Crowd Flow Monitoring using Trajectory forecasting and clustering

NSE-UC1 addresses the general question of identifying patterns of individual or collective mobility behaviour, in terms of clusters, trends, densities etc in indoor/ outdoor/ hybrid locations.

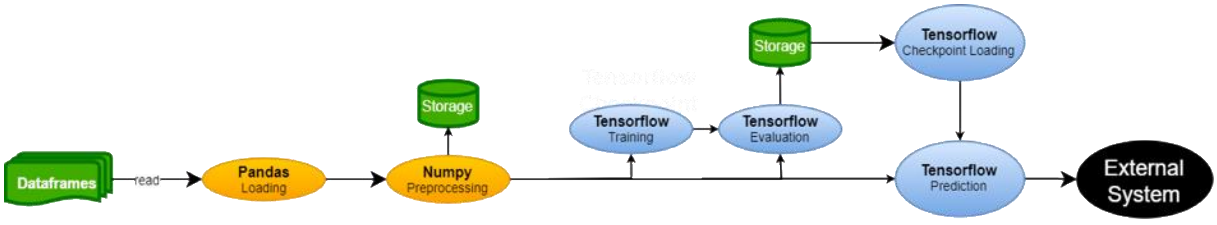


Figure 2: Pipeline for trajectory forecasting

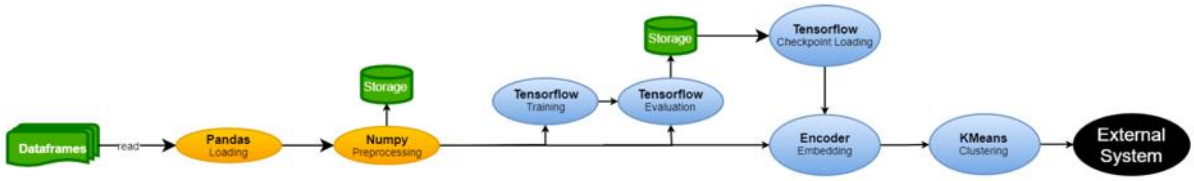


Figure 3: Pipeline for trajectory clustering

We investigated trajectory predictions and clustering using RNNs with LSTMs/GRUs, CNNs and hybrids, sequence to sequence (Seq2Seq) auto-encoders and variants for the crowd flow monitoring functionality.

Figure 2 shows the pipeline for single-step prediction using RNN with LSTM/GRU models. The goal is to predict the next point given a random anonymous trajectory with spatial and time stamp information. We tested sequence to sequence models based on LSTM/GRU for representation learning and clustering the features from autoencoder using K-means clustering, an instance is shown in Figure 3.

Table 4: Crowd Flow Monitoring as a Service

Crowd Flow Monitoring	
Pipeline	Figure 2 and Figure 3 show the pipelines for trajectory predictions and clustering used for the crowd flow monitoring service.
Input data	**Trajectory information:** trajectory/track id, time stamps, geo-spatial coordinates (latitude & longitude). **Additional information:** velocity, acceleration, user ids, mode of transportation (walking, bicycle, bus, train, etc.). **Data Sources:** Input data for offline training from open datasets like open PFLOW, input data for online training from data store through REST API/Kafka. **Data Format:** CSV, JSON, or XML
Dependencies & ML Libraries	Python, TensorFlow, Numpy, CuPY, Numba, Pandas, cuDF, Scikit-Learn, tslearn.
Data pre-processing	The pre-processing steps include trajectory interpolation, correction and smoothing, outlier detection and filtering,

	resampling based on timestamps; data transformations to calculate rate of turn, velocity, and dynamic time warping; trajectory data split into train, test, and validation.
Building models/other analytics	GRU/LSTM seq2seq encoder-decoder models for single-step and multi-step predictions, K-means clustering.
Inference and Results	• If contextual data available, then the output provides the spatio-temporal patterns indicating crowd movement such as possible visitor paths and POIs. • Interpretable visualizations from the trajectories clustering indicating crowd mobility patterns.

2.2.4 Group-In: to learn group mobility Characteristics

LOCUS will perform crowd analytics in urban areas using limited auxiliary sensors such as cameras for benchmarking and improving the accuracy of the estimation, as well as machine learning with training from the historical data. This is developed under the functionality: Group-In[1] in NSE-UC2. Group-In collects only wireless traces from the Bluetooth-enabled mobile devices for group inference. The key problem addressed in this work is to detect not only static groups but also moving groups with a multi-phased approach based on noisy wireless Received Signal Strength Indicator (RSSIs) observed by multiple wireless scanners without localization support.

Group-In provides two outcomes: 1) group detection in short time intervals such as two minutes and 2) long-term linkages such as a month.

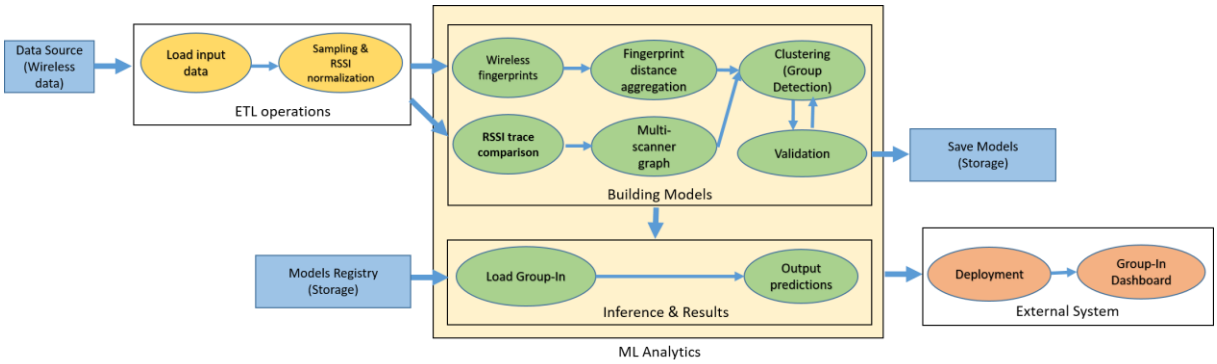


Figure 4: Pipeline for Group-In Service

Table 5: Group-In Service

Group-In	
Pipeline	Figure 4 shows the general steps involved in the Group-In service
Description	Provides analytics for monitoring crowd mobility and inferring group structures. Multi-phased analytics based on noisy wireless Received Signal Strength Indicator observed by multiple wireless scanners.
Input data	Location data from time-stamped trajectories or in other aggregated forms such as mobility flows, heatmaps, densities, etc. Collection of user location and other essential data from sensors like Bluetooth, WiFi, etc.
Data Scope	Wireless data (e.g., Bluetooth advertisement, Wi-Fi)
Data pre-processing	Sampling, RSSI normalization, trajectory identification/formation.
System Components	*Data Movement, Data Persistence, LMF, NFVO*
Building models/other analytics	Trajectories/Flows identification. Clustering algorithms: • DenGraph: Density-based scanning approach on graphs models • HCS: Clustering highly connected subgraphs • MaxClique: Clustering fully connected subgraphs
Validation Strategy	Jaccard and pairwise accuracy of output groups.
Identified Functions	ML Functions, Trajectories Identification, Flows Identification
Inference and Results	• The inference of static or mobile groups. • Estimate of group size. • Group detection in short time intervals such as 2 minutes. • Group detection in long-term linkages such as a month.

2.2.5 Vulnerable road user

Vulnerable Road User (VRU) use case represents new 5G vertical – Cooperative and Automated Mobility and it was initially described in LOCUS D2.1. It introduced challenging requirements, required by road safety based on the pre-standardization work performed in 5G Automotive Association. To that end, LOCUS developed two related functionalities in D5.1,

in alignment with ETSI Intelligent Transport Systems standard and proposed ML techniques for both functionalities:

- Vulnerable road users clustering
- Time to collision as a service in V2X

In addition, the 'Time to collision as a service in V2X' functionality can be potentially applied to NSE-UC4: Logistics in a seaport terminal using AGVs use case and relevant discussion to investigate integration opportunity has started in terms of common modelling aspects between AGVs and vehicles and a possible collaboration between partners within and after the project lifetime. Indeed, an actual implementation within the project lifetime is unrealistic, as it would require further analysis and effort.

LOCUS identified recent progress in the VRU protection standards work:

- ETSI TS 103 300-2 – defines functional architecture and requirements for VRU (published May 2020)
- ETSI TS 103 300-3 – defines VRU awareness service, message, and protocols (published Nov 2020)
- 5G Automotive Association 'Vulnerable Road User Protection' whitepaper (published Sep 2020)

As a result, in an alignment with the objective of this document to streamline UCs and functions selecting the most representative ones, WP5 reviewed and analysed both VRU functionalities taking into account their commercial status, similarities with other functionalities developed in LOCUS and their implementation potential. This analysis will be brought to the attention of the innovation and exploitation focus group within the T7.3 activity (Innovation and Exploitation). Table 6 below describes the outcome of this work with D5.3 proposal.

Table 6: Status of NSE-UC3: VRU Clustering

NSE-UC3 Functionality	Status	Proposal
Time to collision as a service in V2X	• Commercial deployment – short-term • Important enabler for new transport applications (driving insurance, traffic management, transport planning) • Potential applicability to NSE-UC4: Logistics in a seaport terminal using AGVs	Develop further in D5.3 – Analytics service specification and UC mapping to an Intent-Based-

		Analytics paradigm provided

Vulnerable road users clustering	• Commercial deployment – mid-term
	• Similar to other functionalities
	  • **NSE-UC1-Functionality-1**: Identifying crowd mobility patterns (spatio-temporal) – Location analytics such as possible visitor paths, POIs (indoor/outdoor/hybrid)
	  • **NSE-UC2-Functionality-2**: Using multi-modal data for crowd mobility – COVID-19 as a special case
	• Identified similarities with user clustering in High Precision Localization for Stadium Entry use case currently being developed in WP3/T3.2
	• Started internal discussion on potential adoption of WP3/T3.2 simulation results to VRU clustering but input not ready for D5.3 due to WP3 work taking priority
		Streamline and consider functionality 'merge' with NSE-UC-1 Functionality 1 and NSE-UC-2 Functionality 2 to avoid overlap in D5.3

2.2.6 Time to Collision as a service in V2X

By leveraging V2X technology capabilities, this analytics service could support transport vertical related businesses such as a driver/car insurance, traffic management or a transport network companies to improve their decision-making process. It could be achieved by providing a comprehensive Time to Collision parameter data analysis tool for the relevant stakeholders to support their analytics. To this end, the following representative scenarios were identified with detailed description of the service analytics provided in Table 7.

a) Analysis of a collision risk profile of a specific road/V2X user during a predefined period – for a driver/car insurance business need.
b) Analysis of a collision risk profile of a specific road network section/location during a predefined period – for traffic management or a transport network planning business need.

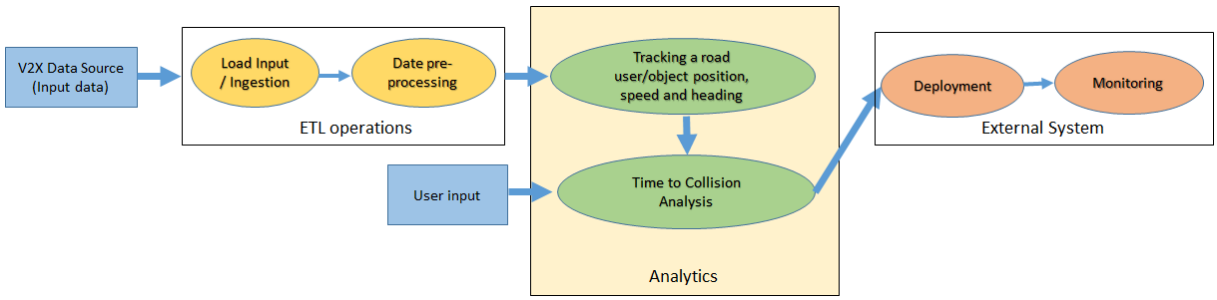


Figure 5: Pipeline for Time to Collision as a service in V2X

Table 7: Time to Collision as a service in V2X analytics.

Analytics specification for Time to Collision as a service in V2X	
Pipeline	Figure 5 shows a pipeline for Time To Collision as a service in V2X
Input data	Road user/object position, speed and heading, road user identifier, road network section/location identifier, timestamp.
Data pre-processing	Time to Collision data sampling is linked to periodicity of receiving the V2X/road safety messages which include position, speed and heading of the road user.
Building models/other analytics	Time to Collision value is calculated by tracking a road user/object position, speed, and heading.
Inference and Results	Based on a predefined time period, unique road user or a road network section/location identifier, collision risk profile characteristics for the requested user, road section/location is generated. Results are based on the statistical analysis of the Time to Collision parameter values distribution. Analysis may provide more advanced classification functions such as an evaluation if the user is in a low, medium or a high-risk collision category.
Deployment	V2X connectivity and messaging support is assumed between the road users.

2.2.7 Logistics in a seaport terminal using AGVs

This functionality is related to logistics in a seaport terminal using AGVs and addresses the verification of the performances of the 5G positioning system in an operative context. The inputs from the 5G positioning system are used by the AGV navigation system to drive it during its shuttling operations.

Analytics is made by the mission/navigation system to determine the next move the AGV has to do to accomplish its mission, based on stored map and real time location information received by the positioning system.

No ML is used in this use case, but an expert system based on CLIPS. Each rule in the expert system is associated to specific co-functions for implementing the computational parts. The set of rules is a basic set that can be extended to cover more complex operations. All rules operate in parallel exploiting the facts that are introduced or created by the rules at runtime. So, choices depend on the current system situation and policies that are introduced in the system to handle the different events.

AGV path computing uses the A*[2] algorithm, which is interfaced as a co-function with the expert system. It is called every time a new mission is assigned to an AGV.

The computed path is provided to the AGV navigation/control system that manages the movements of the AGV based on the position data and the path provided by the expert system. The Control system of the AGV is a nonlinear fuzzy controller.

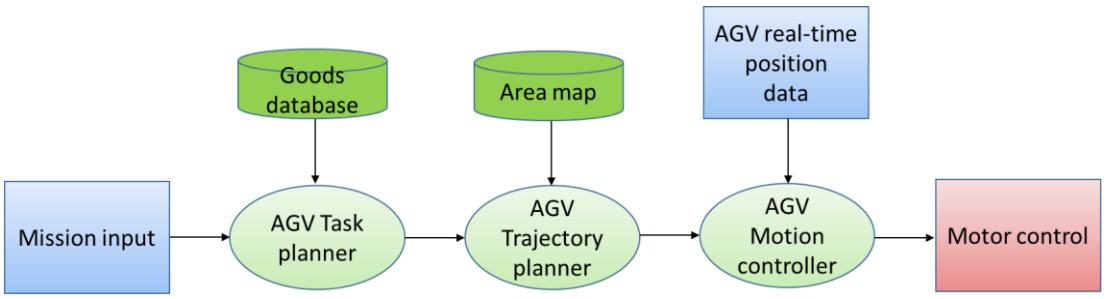


Figure 6: Pipeline for Logistics in a Seaport terminal using AGV

Table 8: Logistics in a Seaport terminal using AGVs

AGV	
Pipeline	Figure 6 shows the process pipeline for this Use Case.
Input data	AGV position data, Goods inventory database, Logistic Area map, mission order from operator
Data pre-processing	The computed path is provided to the AGV navigation/control system that manages the movements of the AGV based on the

	position data and the path provided by the expert system. The Control system of the AGV is a nonlinear fuzzy controller. It first normalized both the desired target position (provided by the path computed by the A* algorithm[2]) and the position data in the range [0,1]. Then, these data are fed to the fuzzy rules. Three sets of fuzzy rules are defined to handle long, medium, and short-range movement. In this way it is possible to optimize the vehicle, reaction optimizing the movement accuracy and the AGV responsiveness
Building models/other analytics	The relational DB including the information about the status and position of the AGV, and the freights data (type, weight, size and position) was implemented using MySQL. The database can be updated and queried from the management system. A specific set of queries were defined and the interface between the system manager and the relational database was defined and implemented.
Inference and Results	The control of AGV motors is executed based on the AGV position data and the path computed by the Trajectory Planner
Deployment	The management system is implemented using an expert system based on CLIPS that was embedded in a master program made using python. The A* algorithm for the AGV path computing exploits a specific python library and is interfaced as a co-function with the expert system. The Control system of the AGV is a nonlinear fuzzy controller. The relational database including the information about the status and position of the AGV, and the freights data (type, weight, size and position) was implemented using MySQL as DB.

2.2.8 Transportation optimization based on identification of traffic profiles

In NSE-UC5, the analytics request translates into a series of sequential processing jobs (shown in Figure 7) that will ultimately produce the ongoing traffic profile updates mechanism. These internal functional blocks are related to data input (in conjunction with the LOCUS data

persistence module), data pre-processing in the microservice container environment, invocation of the specific ML jobs for the path identification and persistence of their respective output into the LOCUS geospatial schema, aggregation of the velocities and statistical profiling of each UE device found in the underlying area. This analytics service is a stateful service that will be performed in a repetitive manner in order to update each identified path traffic profile. Updates found on the total traffic profiles of the areas can then be filtered to propagate the appropriate responses on the service's subscriber via the LOCUS API gateway.

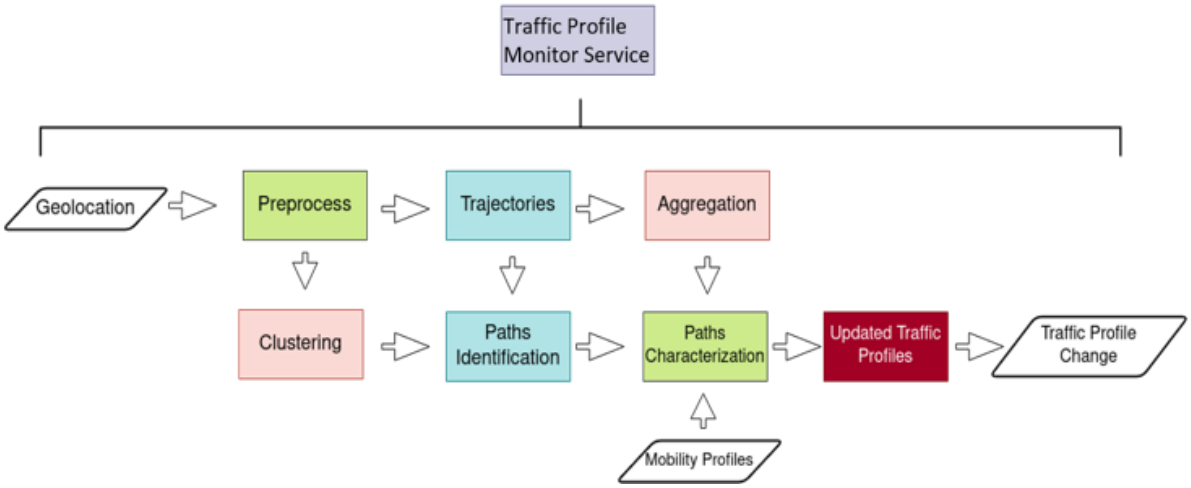


Figure 7: Pipeline for Traffic Profile Monitor Service

Table 9 - transportation optimization using traffic profiles.

Transportation optimization using traffic profiles	
Pipeline	Figure 7 shows the process pipeline for this Use Case.
Input data	Geolocation information (latitude, longitude, accuracy, user device identifier, timestamp) of several UE devices for the selected area of analysis.
Data pre-processing	The consecutive geolocation entries are required to be time-aligned and padded to match the lowest granularity of the dataset. For this purpose, stationary geolocation entries are imputed and interpolated making a more homogeneous input dataset. In addition, these entries are post-processed in sequence in order to compute a) the delta-angle of relocation b) the delta-distance of relocation that are both used as inputs for the training process
Building models/other analytics	For the purposes of the trajectory identification, a combination of clustering and geospatial post-processing is performed. The DBSCAN / OPTICS implementation is executed

	on the location dataset (as described in the previous section) in order to create several paths that synopsize (cluster) the movements of various UE devices. By utilizing the generated features (velocity, angle, location) the clustering results closely resemble the underlying street structure and are aligned with existing maps of the area
Inference and Results	The inference results are split into two categories: a) identification of the final shapes of the user general trajectories (paths) that are commonly followed and are in close correspondence with the various streets/roads of the underlying area, b) Classification of the transportation profile in order to create accurate description of the mobility context for each of the identified paths. This transportation profile is a distribution between pedestrian, vehicular, fixed-means transportation (e.g. 20% pedestrian, 30% vehicular, 50% fixed-means) based on the UE distribution
Deployment	The system is initialized with no preselected areas to monitor for their traffic profiles and no existing traffic profiles. External requests populate the execution entries of the ML pipeline with specific areas (denoted as polygons or square geo-areas) that will be used as filter for the geolocation input data. The system's user will then use a polling (or websocket) endpoint to monitor the output of the traffic profile for the selected area. In the background, the LOCUS platform will perform the ML pipeline and generate identified paths which will be stored in the persistence module and notify the application user to access it (if they wish to use them as input for visualization of a curtain dashboard or other use). Afterwards the corresponding traffic profile will also be produced and propagated to the end user by the same channel. The interval denoted from the application user will parametrize the re-evaluation of the traffic profile pipeline which will trigger traffic change events via the designated channel. These triggers will then be integrated in an external decision support system for the transportation optimization (e.g., a smart traffic light control system)

2.2.9 *Positioning and Flow Monitoring for Controlling COVID-19*

The goal of this functionality/ service is the following. Given an identified case of COVID-19 infection, it traces back the persons to have potentially been in proximity with the positive case within a certain number (to be set) of previous hours/days and to estimate risk factors and their spatiotemporal evolution using epidemiological data combined with the flows of people moving from one area to another area.

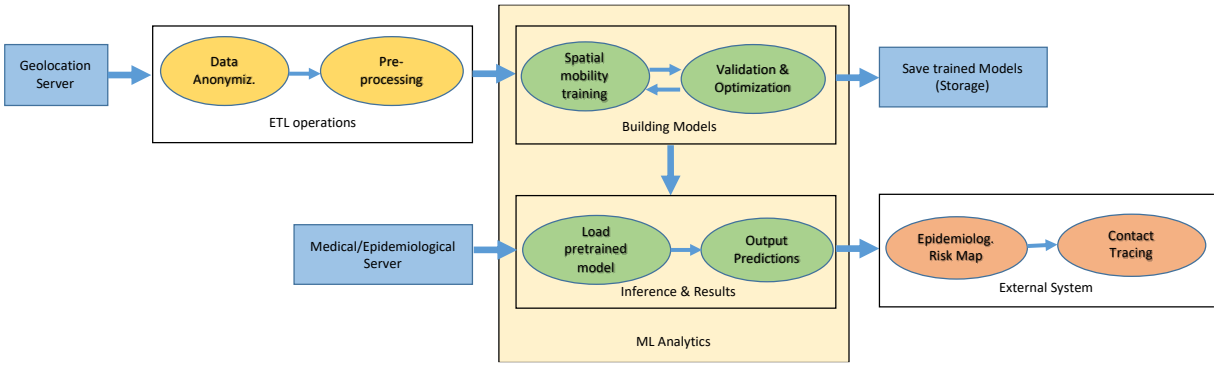


Figure 8: Flow monitoring for Controlling COVID-19 – Pipeline

Table 10: Flow Monitoring for Controlling COVID-19 as a Service

Controlling COVID-19	
Pipeline	Figure 8 shows the pipelines for contact tracing and epidemiological risk mapping for Controlling COVID-19 service.
Input data	Geolocation data: anonymized geo-spatial coordinates (latitude & longitude), timestamps. Medical/Epidemiological data: Contagiousness indicators per area, personal epidemiological data if available.
Data pre-processing	Pre-processing includes geolocation data filtering based on specific RAT-dependent/RAT-independent. technologies in 5G ecosystem.
Building models/other analytics	• Spatiotemporal map (patterns) indicating evolution of epidemiological risk factors, using mobility maps to assess possible future hotspots.
Inference and Results	• Analysis of the relations between regions to find the need for reducing mobility among specific regions. • Contact tracing related to people those who were in the same area in close time instants.

2.3 Microservices and Functions Catalogue

As already anticipated and discussed in the previous deliverables D2.4 (Deliverable D2.4 "System Architecture - preliminary version", 2020) and D5.1 (Deliverable D5.1, "Design and implementation of virtualization technologies and pattern recognition mechanisms for physical analytics, preliminary version", 2021) , LOCUS will deploy and offer its analytic services in a microservices based virtualization platform implemented either based on Kubernetes/Docker containers or traditional VM based VNFs. In practice, a microservices architecture splits an application into multiple smaller parts/ microservices that perform fine-grained functions, and each microservice will have a different logical function. Traditional applications with monolithic architectures have all the application's components and functions in a single instance, whereas microservices break apart monolithic applications into smaller parts. Some of the advantages of microservices are:

- Microservices are loosely coupled and are independently deployable, this allows more autonomy. Each microservice can be deployed independently as required, enabling faster updates and continuous improvement by supporting the CI/CD (Continuous Integration, Continuous Delivery & Deployment) process.
- Microservices are independently scalable as they run autonomously. There are container orchestration tools to scale individual microservices automatically by allocating more processing power or even spinning up new instances of microservices, as required.
- Microservices provide better fault isolation and in turn allows to build more resilient applications. When a specific microservice fails, that single microservice can be easily isolated and this can prevent cascading failures that would cause the entire service or application to crash.
- Microservices provide better data security and compliance. Microservices are predominantly run through secure APIs, which protect the data it processes by making sure it is only accessible to specifically authorized applications, users, and servers.

We defined the various LOCUS analytic services/applications and their pipelines in the previous section. The list of analytic services (see Table 2) forms the basis for a services catalogue. The next logical step is to split each service/ application into microservices and identify the different logical functions involved in it. This step will enable us to build a functions catalogue and to identify the common functions across the different pipelines instantiated for the various analytic services. Ultimately, we will have a services Catalogue and a Functions Catalogue.

In this context, the crowd flow monitoring service (defined in section 2.2.3) can be broken into several high-level services such as to identify possible visitor paths (Route), an accessible zone, closest POI; to monitor a given area, and the events occurring at a given

indoor/outdoor/hybrid environment based on the spatio-temporal crowd mobility patterns. To implement each of these services, we chain a set of logical functions at different stages from the data ingestion, data pre-processing that includes several transformations, and data learning based on the prescribed ML analytics for that service. Figure 9 shows the functions catalogue for the crowd flow monitoring service with the associated high-level services, the set of logical functions grouped under different stages of processing in the analytics pipeline.

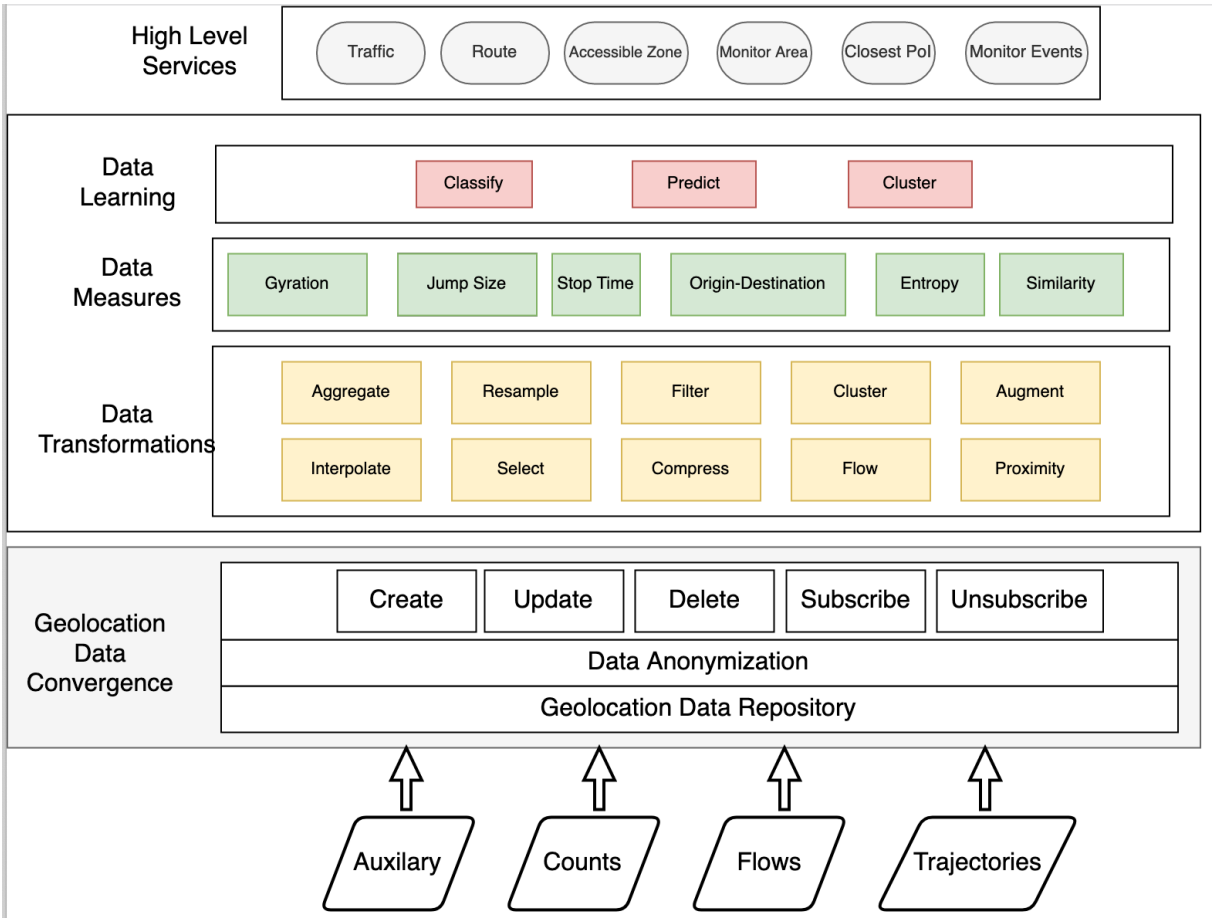


Figure 9: Microservices and Functions Catalogue for Crowd Flow Monitoring Service (Section 0)

The goal is to develop similar functions catalogue for all the other analytic services. Ultimately, we will build the LOCUS functions catalogue that contains all the set of logical functions from various LOCUS analytic services.

2.4 LOCUS privacy services

As presented in D2.4 and D5.1, the LOCUS architecture includes specific services dedicated to location security and privacy. This section presents the functions that will be available in the LOCUS catalogue and the services to reduce privacy threats. Privacy prevents controlling and influencing the information related to users. We consider the model where users execute

queries with position and content requests. Indeed, it is possible for the service provider to identify places that these users frequently visit, reveal their personal information or identify change in lifestyles, store, and provide personal data to third parties.

To prevent these types of attacks, LOCUS identifies a set of specific functions that are used to implement privacy services. In particular, the identified functions are k-anonymity and result aggregation. k-anonymity processes the data in order to make the output related to one user indistinguishable from at least other k-1 individuals. While result aggregation uses aggregate responses of statistical extracted features (counts, frequencies, etc.) to respond to 3rd party queries, here, the request is provided from the Location Based Service (LBS). To produce the anonymous counterpart, the privacy services have to incorporate algorithms that works towards two main directions:

1. removing any obvious identifiers that are part of the user request (e.g., ID, name)
2. effectively transforming the exact location of request into a spatiotemporal area (the area of anonymity). This area must include a sufficient number of nearby users so as to prevent the attacker from locating the requester. These users formulate the anonymity set of the requester.

The study and the evaluation of the algorithms to implement the two functions in LOCUS is addressed in task 2.2 and documented in D2.3. They consider k−1 different user groups (dummy locations in sparsely populated regions) to achieve k-anonymity that also consider user request types. The procedure is as follows:

1. Construct a space data structure for organizing users' locations in a *2*-dimensional space.
2. Search and select the nearest neighbour users to implement the anonymity set that minimize the probability to extract sensitive users' information.

Space data structures are a family of data structures that arrange geometric data for efficient search. For example, doing queries like "find 1000 closest users to the real user", and returning results with minimal latency even when searching in database containing very large number of records. There are two fundamental query types: nearest neighbours and range queries. They are addressed in an efficient way through spatial data structures [3].

Space data structures divide a space into smaller convex subspaces by hyper planes. The subdivisions and their resulting sub-regions are represented by tree structures. In this process, sub-regions are recursively partitioned into smaller sub-regions, and this procedure terminates when all sub-regions satisfy one or more predefined requirements. For the LOCUS functions catalogue we consider two different approaches: kd-tree and quad-tree. Table 11 presents the set of functions, and relative descriptions, that are available in the LOCUS catalogue to implement the privacy services. In addition to the two spatial tree approaches, the catalogue also includes the function to search the users in the space and to evaluate the anonymity (entropy) of the selected user group.

Table 11: Functions for privacy service present int the LOCUS catalogue

Functions	Descriptions
Quad-tree	Quad-tree approach divides a space into subspaces, which are also called quadrants [4]. It is based on a recursive regular decomposition of a space into four equal sizes. Quad-tree method starts with a big tile that covers the whole area and divides it by two horizontal and vertical lines to have four equal areas which are new tiles and then inspect each tile to see if it has more than a predefined threshold, points in it. The process continues till there would be no more tile with several points bigger than threshold.
kd-tree	Kd-tree is constructed by recursively dividing regions alone the X and Y a is, alternately [5] . The division is done at the median points along the axes. In the canonical method of kd-tree construction, points are inserted by selecting the median of the points into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. This method leads to a balanced tree, in which each leaf node is approximately at the same distance from the root.
Spatial search	In spatial data structure, search is performed starting from the top tree level and drilling down, ignoring all the boxes that don't intersect our query box. The method compares the data in each node with the one we are looking for. If the compared node doesn't match, then it proceeds to the right child or the left child, depending on the outcome of the following comparison: If the node that we are searching for is lower than the one we were comparing it with, we proceed to the left child, otherwise (if it's larger) we go to the right child. Such approach is faster than a naive loop search.
Entropy evaluation	The entropy is used to measure the uncertainty of a group of users, the bigger the value of the entropy the more uncertain the group. The function takes into account the entropy based on Euclidean distance between the users' locations and the request content[6] .

Finally, the functions are used to implement the privacy services. Figure 10 shows an example of flow diagram for the privacy service. Real user request and users state information are used as input. They feed the function to create the space data structure. After the successful construction of the tree, the service executes the spatial search to extract the subset of

neighbour users to the real users. Thus, the entropy for each subgroup is computed, to evaluate the uncertainty of the user groups. Finally, the user group with the largest entropy is filtered and provided as output.

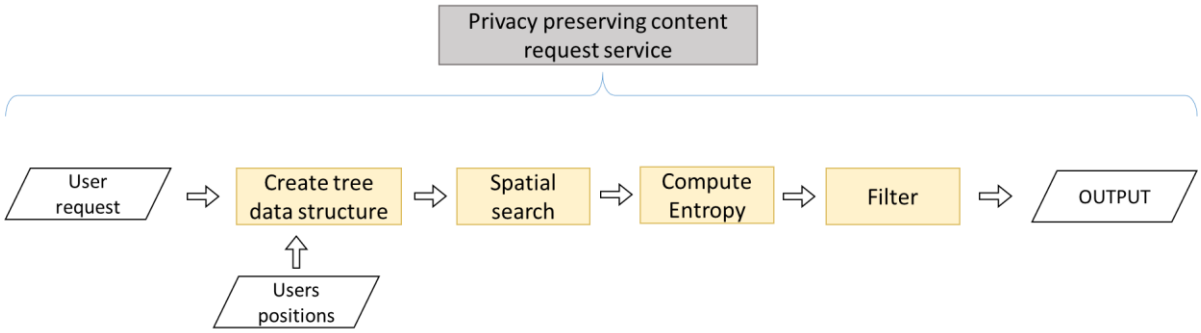


Figure 10: Flow diagram of the selected algorithm

2.5 Virtualization of Analytics Services and Pipelines

This section provides a description of the experimental work carried out for the virtualization of the NSE UCs ML pipelines. In particular, this work is the continuation of what was reported as initial work in deliverable D5.1, progressing with the validation in virtualized environments leveraging opensource frameworks like Seldon[7] and Kubeflow[8] for delivering and executing pipelines on-demand.

As described in D5.1, the code that implements a pipeline for UC1 Functionality-1 for identifying crowd mobility patterns, has been used to carry out all the experiments done for the virtualization of the machine learning pipelines. Figure 11: Reference ML Pipeline shows the referred machine learning pipeline.

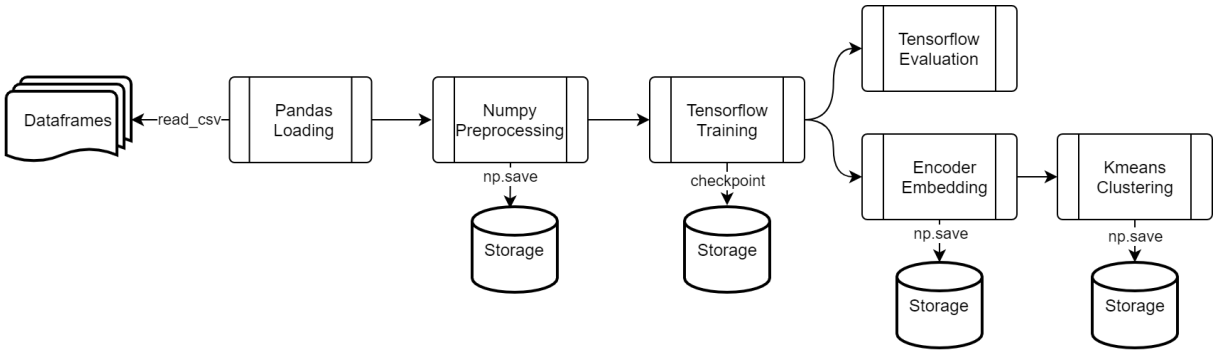


Figure 11: Reference Machine Learning Pipeline

The work has been performed following two parallel streams:

- Single model and graph model serving is performed using Seldon with already trained and saved models from the reference machine learning pipeline software code.

- Kubeflow Pipeline is used for the realization of a complete machine learning pipeline able to perform each step of the reference machine learning pipeline in a virtualized way.

2.5.1 Seldon for model and graph serving

This section outlines the runtime environment and the experiments carried out for the serving of a single machine learning model and the serving of a graph of models.

First of all, by serving we mean the provision of a trained machine learning model (or a machine learning models graph) so that it can be queried (e.g., via REST interfaces) and therefore predictions on new data can be made. That said, Seldon could therefore be a tool to manage the last two steps of the reference NSE UC-1 pipeline shown in Figure 11 (Encoder Embeddings and Kmeans Clustering) since the main scope of Seldon Serving is the automated deployment of machine learning models on Kubernetes as Pods [9]. To implement and validate this first approach for model serving, the encoder and clustering machine learning models have been generated from the reference pipeline code steps taking care of data loading, pre-processing and training, using the framework Google Colaboratory (Colab)[10]. As an alternative to Google Colab for generating and saving the encoder and clustering models, a local deployment of the reference machine learning pipeline software code using python3 and all the required dependency (installed via pip3) could also be used.

Figure 12 shows how the reference machine learning pipeline software code steps are divided in this specific environment.

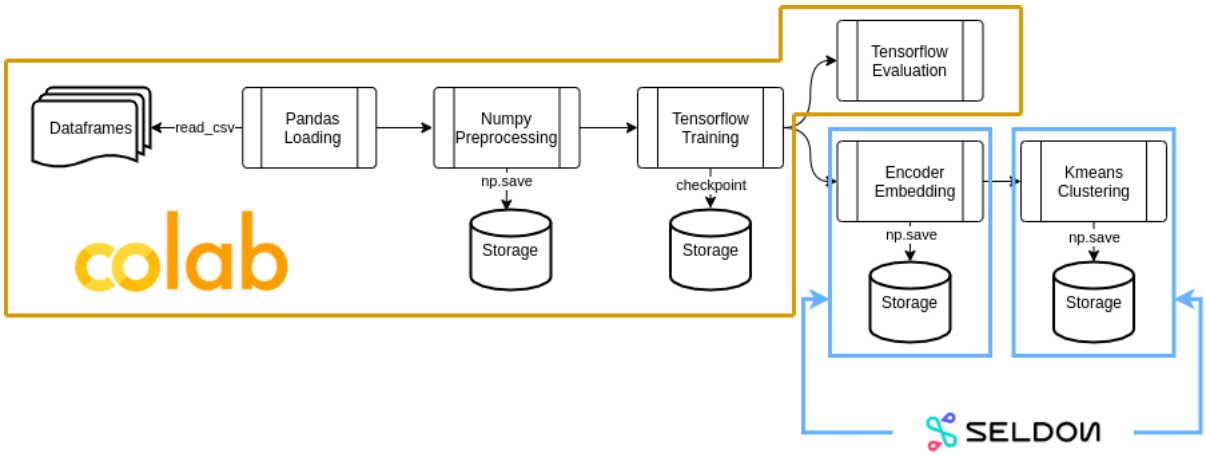


Figure 12: Reference Machine Learning Pipeline - Seldon Deployments

As displayed in Figure 13, this experimental scenario is deployed in an NFV-oriented environment with Minikube [11] as the provider of a single-node Kubernetes cluster in order to use the latter as a container orchestrator (and be used by Seldon for deploying models as PODs composed by a set of containers). The Minikube instance runs in an OpenStack (i.e., IaaS Infrastructure Manager) virtual machine with 4 vCPUs, 8GB RAM and 40GB HDD. Once the

architecture has been initialized, Seldon is installed, and a “models” namespace is created in which Seldon itself will create the PODs to serve the models.

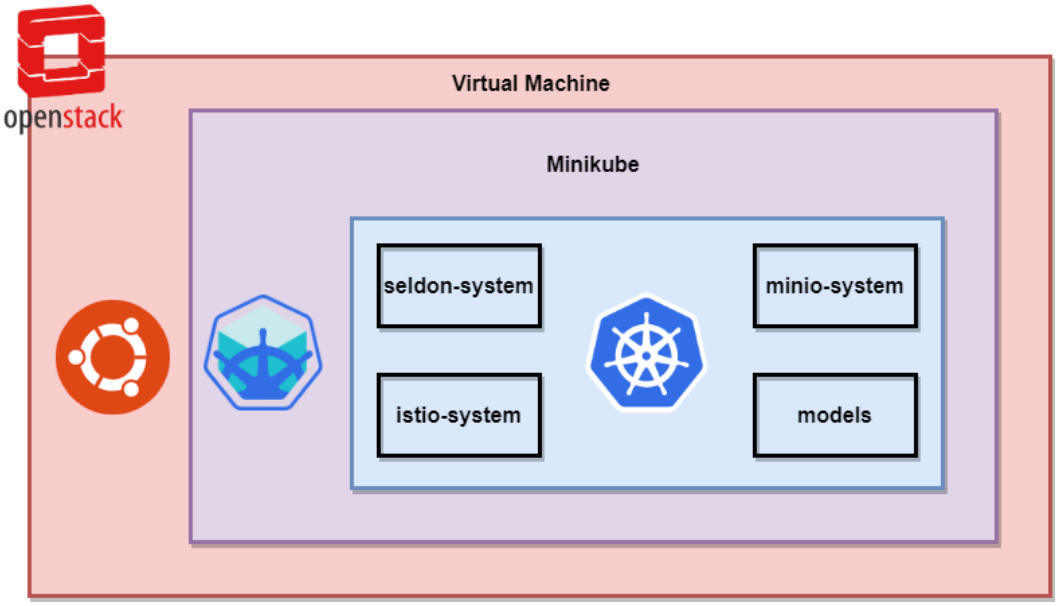


Figure 13: Minikube - Seldon environment

Before serving a model with Seldon, the trained model must be saved to a storage system accessible by Seldon, MinIO [12] in this case, or embedded in the docker image that will be managed by Seldon.

In the preliminary work reported in D5.1, the serving of a single model has been shown via the creation of a Kubernetes custom resource definition called Seldon deployment file (JSON or YAML format) that allows the Seldon user to define a model as an inference graph of Kubernetes PODs. As shown in Figure 14, once a Seldon deployment file that allows Seldon to serve a model (e.g., the encoder model or the clustering model from the reference machine learning pipeline software code) is written, a POD will be created containing all the containers required to serve the model and make the latter accessible for predict request via REST interfaces using Istio [13] as an ingress Kubernetes gateway.

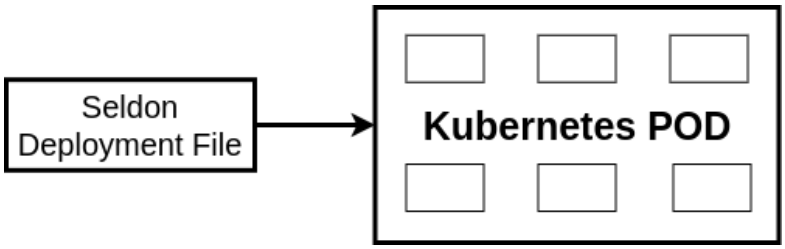


Figure 14: Single Model deployment logic

The deployment of independent encoder and clustering models leads to the creation, by Seldon, of two distinct PODs that perform the individual serving of the two models (and therefore they need to be queried separately for prediction on new data).

As part of the initial work described in D5.1, the Kubernetes/Seldon deployment files used were leveraging on built-in Seldon Model Servers, that took care of wrapping the model specified (also in the deployment file). This way, Seldon-Core was able to convert the machine learning models into production ready REST/gRPC microservices. However, this approach is not always applicable, and indeed it was not very well suited for LOCUS. In fact, very often the models expect to receive data in a certain format and Seldon's standard servers are not sufficient to cover all possible cases. Hence, the creation of ad hoc servers is necessary, and it can be done by leveraging on the *seldon-core-microservice* python wrapper. In particular, Seldon Core allows to build two type of servers, reusable and non-reusable ones:

- *Reusable Model Servers*, often referred to as prepackaged model servers, allow to deploy a family of similar models without the need to build a new server each time, for example, fetching models from a central repository (see Figure 15).

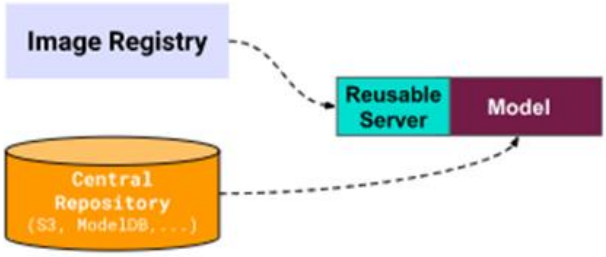


Figure 15: Reusable Model Servers (source: Seldon)

- *Non-reusable Model Servers* are specialized servers meant to serve a single model. Non-reusable Model Server do not require the central repository but require a build of a new image for every model (see Figure 16).

Figure 16: Non-reusable Model Servers (source: Seldon)

For the specific case of the NSE UC1 Functionality-1 pipeline, a non-reusable model server built specifically to manage the serving of the clustering models has been developed.

After training and saving the clustering model using Google Colab or a local deployment of the reference machine learning pipeline software code it is necessary to write such custom model server. Figure 17 shows the python code that wraps the clustering model using two particular functions: *__init__* and *predict*.

```python
from joblib import load
import numpy as np

class ClusteringModel:

        def __init__(self):
                self._model = load('model.joblib')

        def predict(self, X, feature_names = None, meta = None):
                print(self._model.cluster_centers_.dtype)
                to_predict = np.array([np.array(xi).astype(np.float32) for xi in X])
                print(type(to_predict[0][0]))

                return self._model.predict(to_predict)
```

Figure 17: Non-reusable Model Server for Clustering Model

The *__init__* function is used to load the model that must be served while the *predict* function is called whenever the model served will be queried (through an HTTP REST operation offered and proxied by Istio) in order to compute a prediction using the data specified in the request body of the HTTP request. It basically casts the input data in a format compliant to the model requirements and then calls the predict function of the model itself.

Once the model server is ready, it is required to specify in a requirement file the libraries needed by the model in order to work correctly. Figure 18 shows the requirements.txt used for the clustering model serving.

```
kneed == 0.7.0
scikit-learn == 0.23.2
numpy >= 1.8.2
joblib == 0.16.0
seldon-core
```

Figure 18: Requirements file for non-reusable clustering model server

The next step is the preparation of a Dockerfile, that is required to package the non-reusable server as a Docker image. Such software image is then specified in the Kubernetes deployment file in order to allow Seldon to instantiate the POD containing all the containers that will cooperate to serve the clustering model (see Figure 19).

```
FROM python:3.7-slim

COPY requirements.txt /clustering/requirements.txt
COPY ClusteringModel.py /clustering/ClusteringModel.py
COPY model.joblib /clustering/model.joblib

WORKDIR /clustering

RUN pip install -r requirements.txt

EXPOSE 5000

CMD exec seldon-core-microservice ClusteringModel --service-type MODEL
```

Figure 19: Dockerfile for non-reusable clustering model server

Since the reference environment for this experimental work is a single node Kubernetes cluster running in Minikube, it is required to build the Dockerfile (in order to generate the docker image) inside the Minikube environment. This can be done by executing the command:

"eval $(minikube docker-env)"

Followed by the traditional Docker build command to create the image.

Finally, the Kubernetes/Seldon deployment file has to be prepared and executed in order to serve the clustering model.

```yaml
apiVersion: machinelearning.seldon.io/v1alpha2
kind: SeldonDeployment
metadata:
  name: locus-clustering
  namespace: models
spec:
  name: locus-clustering
  predictors:
  - componentSpecs:
    - spec:
        containers:
        - image: clustering:latest
          name: clustering
          imagePullPolicy: IfNotPresent
    graph:
      name: clustering
      endpoint:
        type: REST
      type: MODEL
      children: []
    name: default
    replicas: 1
```

Figure 20: Clustering model Kubernetes deployment file

Using the *kubectl* tool, the deployment file can be then "applied" (i.e. executed through Kubernetes/Seldon with the *kubectl apply* command) in order to let Seldon create the POD and the necessary container within it. Figure 21 shows the POD with the running containers.

```
ubuntu@test-minikube-seldon:~/aiml-test$ kubectl get pod -n models
NAME                                                  READY   STATUS    RESTARTS   AGE
locus-clustering-default-0-clustering-cc4888494-9fhf5   3/3     Running   0          136m
```

Figure 21: POD with running containers for clustering model serving

As depicted in Figure 21, inside the POD, three containers are running:

- *clustering*: is the container generated from the image built in order to create the model server, so, it runs the python scripts that defines the init (to load the clustering model) and predict function (to call the predict function of the model in order to predict on new input data);
- *seldon-container-engine*: built-in seldon core container that handle the seldon-core serving logic. It reads the inference graph and orchestrates the HTTP calls to follow the sequence of the graph components;
- *istio-proxy*: side-car container, act as a proxy for external services in order to make the exposed model accessible for predictions.

At this stage, an HTTP POST request can be issued in order to trigger a prediction of the clustering model. For this purpose, a specific python script has been written which will send a

HTTP POST request to the exposed model containing the encoder output generated during the execution of the machine learning pipeline software code in the first step of this experiment. The http client will send in the request body of the prediction request the embeddings generated by the encoder model that was previously executed locally. Figure 22 and Figure 23 show the response given by the clustering model exposed by Seldon; for sake of readability the middle part of the output has been omitted.

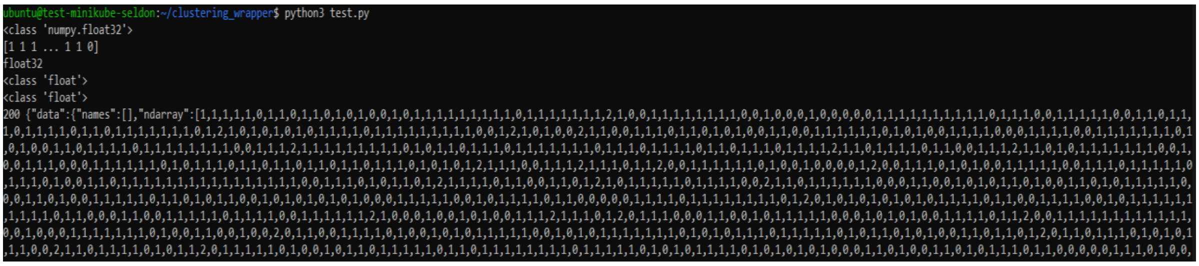


Figure 22: Initial part of the output of the prediction request sent to the clustering model

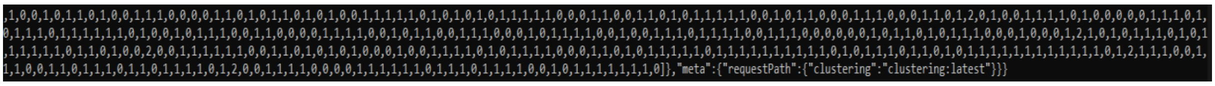


Figure 23: Final part of the output of the prediction request sent to the clustering model

Starting then from the work done for the serving of single models using Seldon, the next step in the machine learning pipeline virtualization process is the use of the Seldon graph capabilities. In particular the Seldon graph allows the encoder and clustering models to be served by Seldon together and therefore connected to each other so the output of the encoder model (generated from a predict request sent to the REST interface of the deployed Seldon graph) will be the input of the clustering model. Therefore rather than deploying independently the two models for serving them manually, Seldon graph enables the interconnection of the two models' instances and realize and actual pipeline where the serving is done as a single operation.

In practice, two different models are being served in a chain, each of which depends on the results of the previous model. Therefore, Seldon manages the job and treats all the models' servers together as one single entity. The creation of a Kubernetes deployment file for the description of a Seldon graph will result in the creation of a single POD containing all the containers needed to serve all the models defined in the deployment, together with the Istio container that will handle the incoming predict request (via the exposed HTTP REST interface).

Figure 24 shows the explained scenario while Figure 25 shows the graph deployment file that has been used to deploy the graph composed by the encoder and clustering models. As an example, in the deployment of the graph, the encoder model has been deployed using one of the serving servers made available by Seldon (a TensorFlow Server in this instance) while the

clustering model has been deployed as described above, i.e. by creating a non-reusable Model Server).

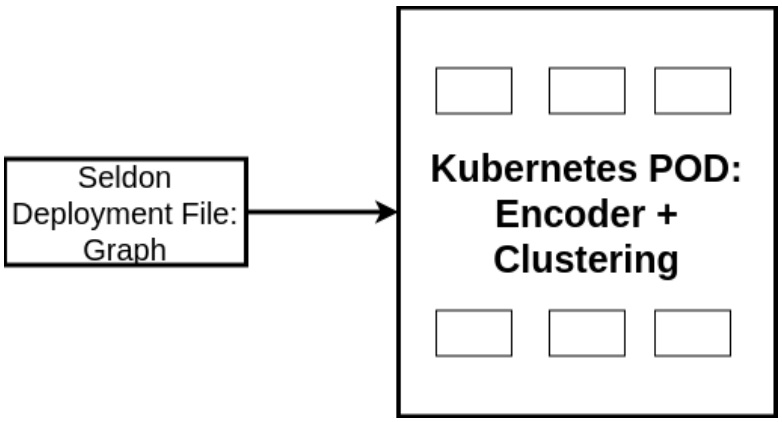


Figure 24: Graph deployment logic

```
apiVersion: machinelearning.seldon.io/v1
kind: SeldonDeployment
metadata:
  name: locus-graph
  namespace: models
spec:
  name: locus-graph
  predictors:
  - componentSpecs:
    - spec:
        containers:
        - image: clustering
          name: clustering
          imagePullPolicy: IfNotPresent
    graph:
      name: encoder
      implementation: TENSORFLOW_SERVER
      envSecretRefName: seldon-init-container-secret-minio
      modelUri: s3://locus-bucket/locusencoder
      children:
        - name: clustering
          type: MODEL
    name: default
    replicas: 1
```

Figure 25: Graph Kubernetes deployment file

After the deployment of the Seldon graph, Figure 26 shows the POD with the running containers for the serving of the encoder and clustering model together.

```
ubuntu@test-minikube-seldon:~/graph_wrapper$ kubectl get pod -n models
NAME                                                  READY   STATUS    RESTARTS   AGE
locus-graph-default-0-clustering-encoder-5dd9bbdf75-5snxr   5/5     Running   0          4h18m
```

Figure 26: POD with running containers for Encoder and Clustering models serving

As depicted in Figure 26, inside the POD, five containers are running:

- *clustering*: is the exact same container that runs in the deployment of the single clustering model as described previously;
- *encoder* and *tfserving*: are the two containers that manage the serving of the encoder model using the built-in Seldon Tensorflow Server. The encoder model is leaded from the MinIO local storage and served via the Seldon server;
- *seldon-container-engine*: is the exact same container as described in the clustering serving POD;
- *istio-proxy:* side-car container which performs the same functions described in the clustering serving.

Once all the containers in the POD are running, an HTTP POST request can be issued to trigger a prediction. This request will start the execution of the predict function of the encoder model generating an output that will be automatically forwarded as input to the clustering model. At this point, the execution of the predict function of the clustering model generates the output that will be received as response body of the initial HTTP POST request.

The predict request has been performed using a python script that pull the data, that will be used as request body, from a tfrecord file. As said above, those data will be the input consumed by the encoder model to execute it predict function.

Figure 27 shows the response of the HTTP POST request.

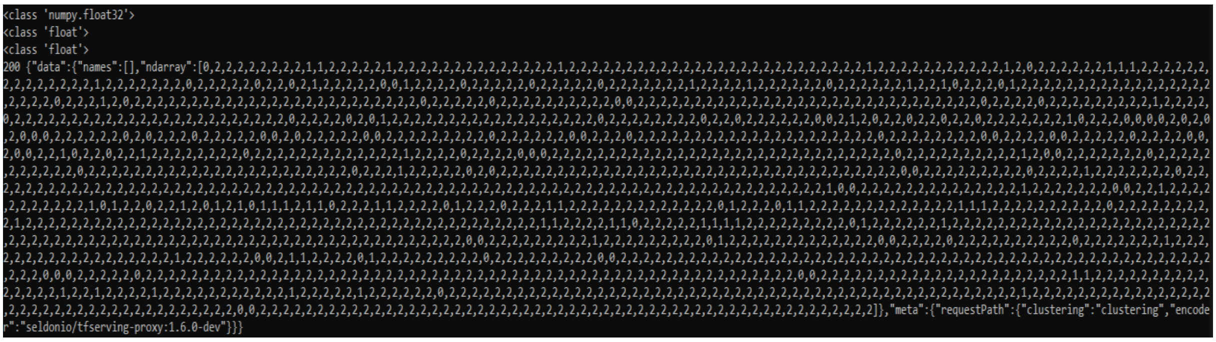


Figure 27: http response of the prediction request to the graph [encoder + clustering]

Running the test script will return the forecast calculated by the graph composed by the encoder and clustering models, as shown in Figure 28.

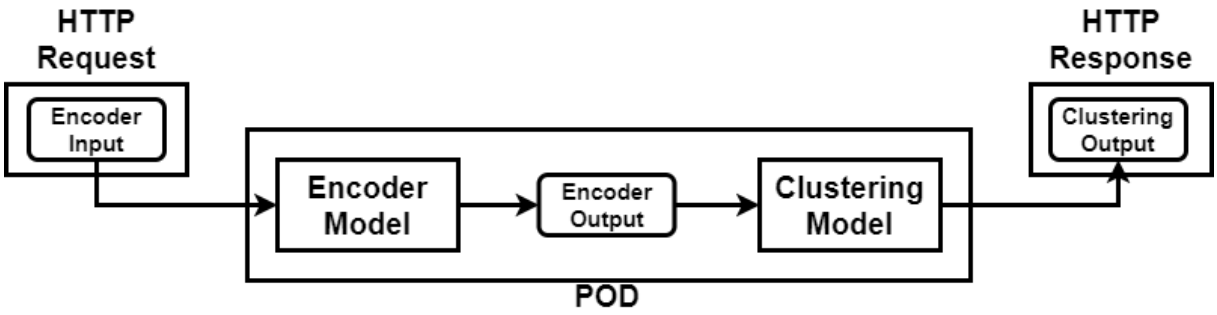


Figure 28: Encoder and Clustering models graph logic

2.5.2 Kubeflow Pipelines

Since Google Colab or a local deployment of machine learning pipeline software code are not solutions suitable for the LOCUS scope, as they do not allow to flexible pre-process data and train the various models, Kubeflow Pipelines have been considered. Kubeflow is a project dedicated to making deployments of machine learning workflows on Kubernetes simple, portable and scalable. A pipeline, in Kubeflow, is a description of a machine learning workflow, including all of the components in the workflow and how they combine in the form of a graph. A pipeline includes the definition of the inputs required to run the pipeline and the inputs and outputs of each component.

In D5.1 a brief overview of Kubeflow and the tools that are made available once installed has been provided. Among these we can also find Seldon. The idea is therefore to use Kubeflow Pipeline for all the steps that precede those of serving the models for which, instead, Seldon can be used. Figure 29 shows how the reference machine learning pipeline software code steps are covered by Kubeflow in this specific validation environment.

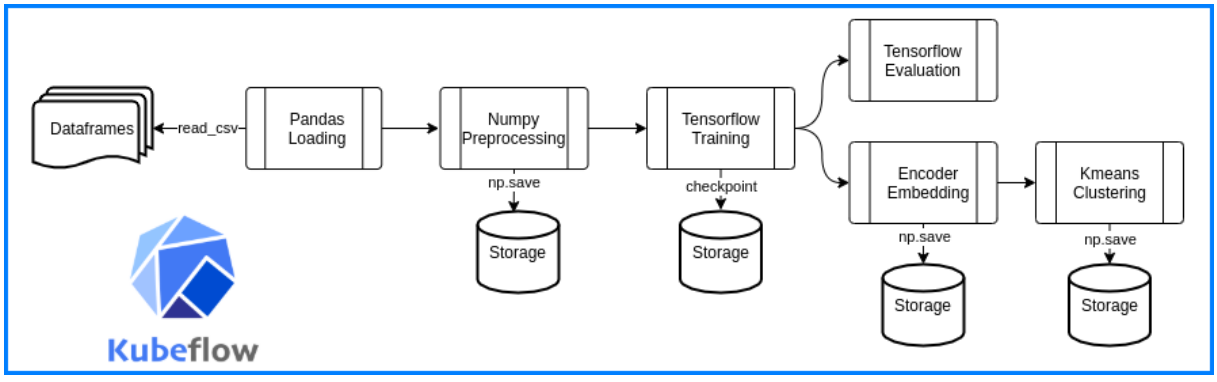


Figure 29: Reference Machine Learning Pipeline - Kubeflow Deployments

A pipeline component in Kubeflow is an implementation of a pipeline task, a self-contained set of user code, representing a step in the workflow. Each component takes one or more inputs and may produce one or more outputs. Then, a pipeline component is packaged as a Docker image, that perform a single step in the pipeline (e.g., a component can be responsible

for data pre-processing, data transformation, model training and so on). So, from each step of the reference machine learning pipeline, a Kubeflow Pipeline reusable component can be created (i.e. that can be easily reused in other pipelines). Figure 30 shows the steps required to implement such approach.

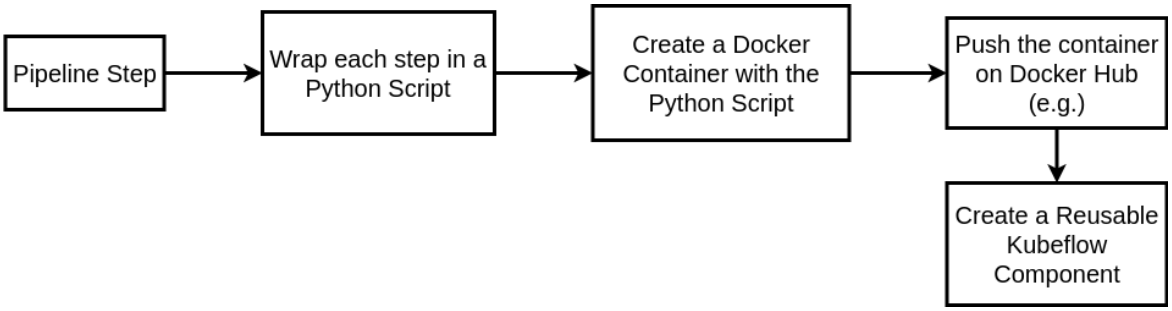


Figure 30: Create a reusable Kubeflow Pipeline component

As displayed in Figure 31, also in this case the validation of the Kubeflow approach is performed in an NFV-oriented environment with MicroK8s [14] as the provider of a single-node Kubernetes cluster in order to use the latter as a container orchestrator (and be used by Kubeflow Pipeline to run the steps of the pipelines within containers). Kubeflow is pre-installed in MicroK8s and must be only enabled via the enable command provided by MicroK8s itself.

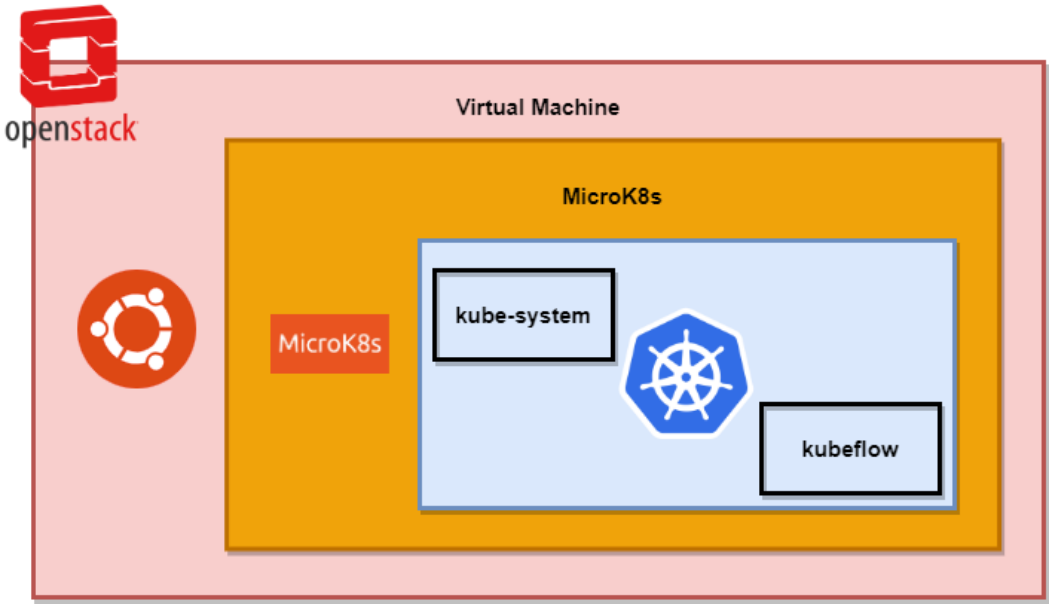


Figure 31: MicroK8s - Kubeflow environment

The NSE UC1 Functionality-1 machine learning pipeline software code has been split in different python scripts in order to individually implement the steps shown Figure 29 and apply the procedure shown in Figure 30. Each program contains the logic of a specific step of the pipeline and make use of files and command-line arguments to pass data to and from the Kubeflow component of the machine learning pipeline. Then, each program has been

containerized using specific Dockerfile. Figure 32 shows the Dockerfile of the program that wrap the pandas loading step.

```
FROM ubuntu:latest

RUN apt-get update -y && apt-get install -y python3-pip python-dev

COPY ./requirements.txt /requirements.txt
COPY ./pandas_loading.py /pandas_loading.py

RUN pip3 install -r /requirements.txt
```

Figure 32: Pandas loading script Dockerfile

Using the Dockerfile, a Docker image is built and then uploaded Docker Hub, as image repository compatible with Kubeflow. At this point a reusable Kubeflow Pipeline component specification in YAML format can be written describing the component itself for the Kubeflow Pipelines system. The created YAML file can be loaded in different pipeline definitions using the Kubeflow Pipeline SDK. Figure 33 shows the YAML definition of the pandas loading reusable Kubeflow Pipeline component.

```
name: Pandas Loading
description: Load data from .csv and process it
inputs:
  - { name: input_path }
outputs:
  - { name: output_path }
implementation:
  container:
    image: docker.io/mikeno/pandas_loading:latest
    command: [
      python3, /pandas_loading.py,
      --Input, { inputPath: input_path },
      --Output, { outputPath: output_path }
    ]
```

Figure 33: Pandas loading Kubeflow Pipeline component definition

As mentioned before, once the definition of the pipeline component has been written, the Kubeflow Pipeline component can be loaded from the YAML file and used in a Kubeflow Pipeline as shown in Figure 34.

```python
import os
import kfp

# Components
downloader_op = kfp.components.load_component_from_file(os.path.join('./pandas_loading/downloader/', 'component.yaml'))
pandas_loading_op = kfp.components.load_component_from_file(os.path.join('./pandas_loading/', 'component.yaml'))
numpy_preprocessing_op = kfp.components.load_component_from_file(os.path.join('./numpy_preprocessing/', 'component.yaml'))
tensorflow_training_op = kfp.components.load_component_from_file(os.path.join('./tensorflow_training/', 'component.yaml'))
kmeans_clustering_op = kfp.components.load_component_from_file(os.path.join('./kmeans_clustering/', 'component.yaml'))

@kfp.dsl.pipeline(
    name = 'Pandas Pipeline',
    description = 'Locus Pipeline (Pandas Loading)'
)
def pandas_pipeline(data_url):
    downloader = downloader_op(url = data_url)
    pandas_loading = pandas_loading_op(input_path = downloader.output)
    numpy_preprocessing = numpy_preprocessing_op(input_path = pandas_loading.output)
    tensorflow_training = tensorflow_training_op(input_path = numpy_preprocessing.output)
    kmeans_clustering = kmeans_clustering_op(input_path = tensorflow_training.outputs['output_path2'])

if __name__ == '__main__':
    import kfp.compiler as compiler
    compiler.Compiler().compile(pandas_pipeline, __file__ + '.tar.gz')
```

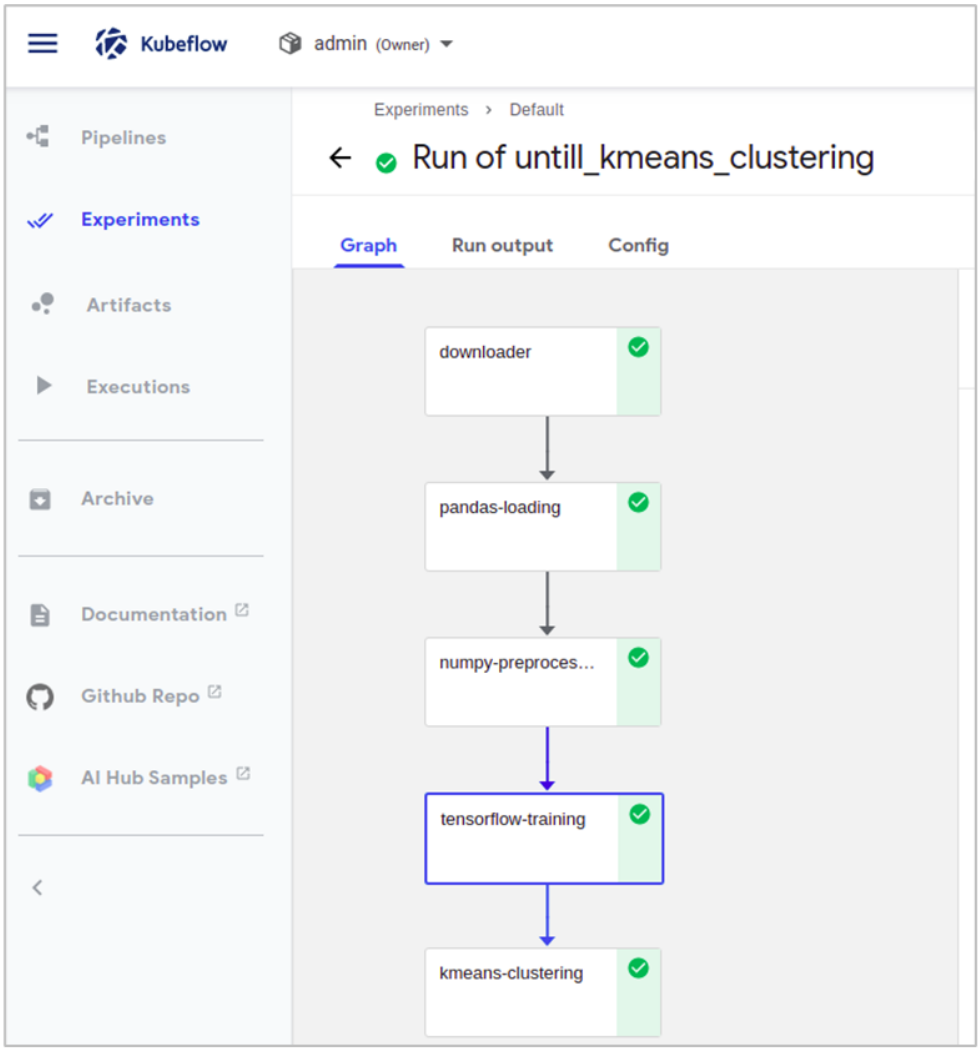
Figure 34: Kubeflow pipeline with five components loaded

Running the above code, describing a Kubeflow pipeline using the Kubeflow Pipeline SDK, will generate a tar.gz file that can be uploaded on Kubeflow Pipeline through its graphical user interface, in order to run the defined pipeline. Running this pipeline will start five containers, one for each component loaded. Each container will execute the python code that has been built inside of it representing one specific step of the pipeline. During the execution of a Pipeline, Kubeflow Pipeline will take care of passing the data between the various components as defined in the python script that describes the pipeline.

Once an imported pipeline run has been generated in Kubeflow pipeline, as each component finishes its execution, we can observe its logs and input and output data. The component *tensorflow_training* takes care of the training of the encoder model as well as the evaluation of the encoder model itself and the generation of the embeddings used for the training of the clustering model, the latter managed by the *kmeans_clustering* component of the pipeline. Figure 35 shows the details about the input/output of the tensorflow_training component after the end of its execution; we can notice that the component has only one input: the output generated from the previous component of the pipeline, the *numpy_pre-processing* component. The outputs of the tensorflow_training component, instead, are two; the first is the encoder model while the second are the generated embeddings (result of the call of the predict function over a subset of the input data). The output of the tensorflow_training component, like the outputs of the other components of the pipeline, are saved as artifact in the MinIO storage managed by Kubeflow.

As mentioned above, the kmeans_clustering component, using as input the embeddings generated by tensorflow_training, takes care of the training of the clustering model, then returns the latter as output (saved as an artifact in MinIO).

Figure 35 shows the input/output details of the component after the end of its execution.



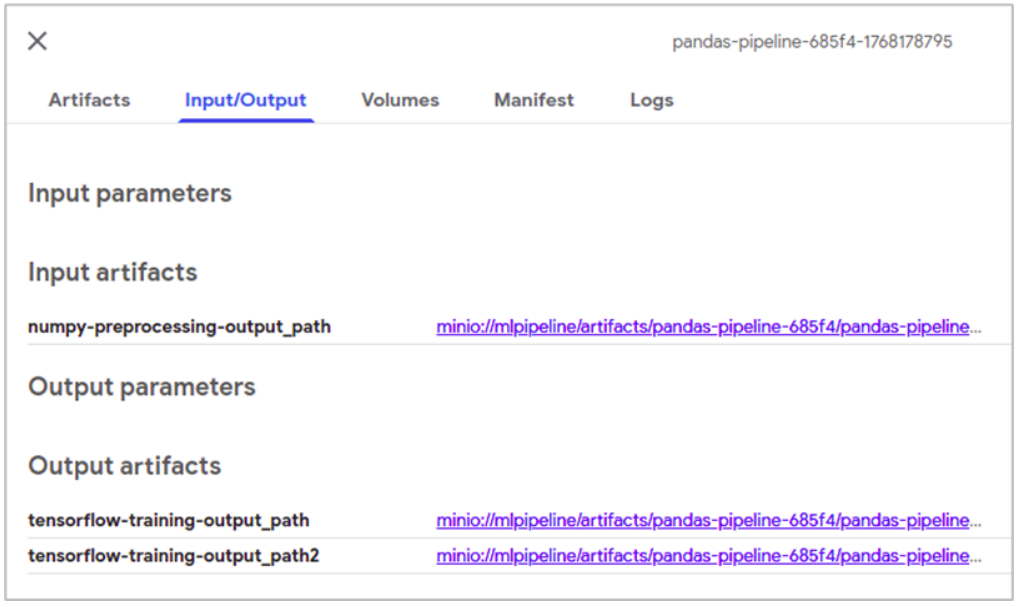


Figure 35: input/output details of the tensorflow_training component

As mentioned above, the kmeans_clustering component, using as input the embeddings generated by tensorflow_training, takes care of the training of the clustering model, then returns the latter as output (saved as an artifact in MinIO).

Figure 36 shows the input/output details of the component after the end of its execution.

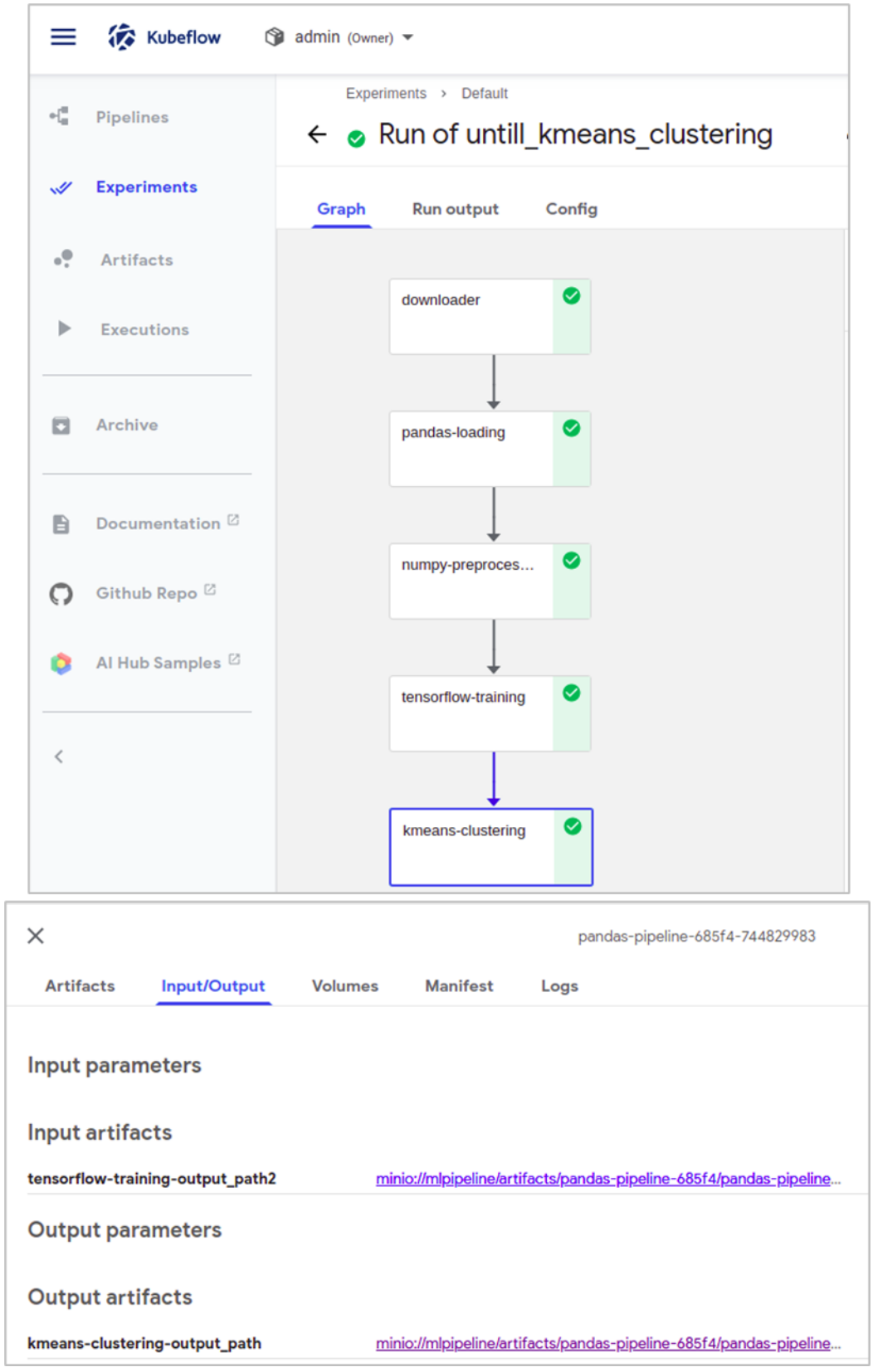


Figure 36: input/output details of the kmeans_clustering component

2.5.3 Next steps

Starting from these results, the next steps will involve the integration of the serving of Machine Learning models, made available by Seldon as the last step/component of the Kubeflow machine learning pipeline built up so far; this will rely on the Seldon instance made available by Kubeflow. It is also possible to guess, from a first analysis, that it will be necessary to convert the clustering serving server from a non-reusable server into a reusable-server in order to use the clustering model generated from the Kmeans_clustering step of the pipeline fetching the latter from the MinIO storage or from other kind of data storage and persistency available in the LOCUS platform. Applicability of this proposed approach to other NSE UCs will be also investigated. Then, as second evolution step, the virtualized NSE UCs machine learning pipelines will be further integrated with the LOCUS MANO developed in WP4 for their use in the LOCUS platform.

2.6 Optimization of ML models for Virtualization

In this section we will describe the techniques developed to optimize ML models' execution and training. The localization algorithms developed in the LOCUS platform make an extensive use of Neural Networks and Machine Learning models in general. These algorithms are expected to run at edge of the network. When running virtualized functions in such environment it is of the uttermost importance to consider the underline hardware as a precious resource, and therefore provide the best software optimization possible. In fact, there are several reasons that substantiate this.

First, LOCUS localization functions will have to share hardware resources in the 5G vRAN with several other functions such as: Digital Signal processing, network traffic classification/analysis (often ML based), other ML based services (e.g.: video analytics), etc. Second, in cloud datacenters new hardware is constantly deployed (usually the HW life cycle is one year), while this is not possible in edge datacenters due to the number of deployment sites and space limitations. With ML models continuously evolving and growing in size, it is paramount to optimize their execution platform in order to keep the pace with their evolution.

In addition to what said above, another crucial point for 5G localization functions is to respect latency constraints: this is the case of in-line functions and latency sensitive functions (e.g.: vulnerable road users), where the correct behaviour of the system directly depends on the execution time.

Usual development and execution of Neural Networks/ML models is demanded to Deep Learning frameworks, like TensorFlow and PyTorch. These frameworks execute their neural networks on a layer-by-layer basis, which can result in an inefficient use of the hardware executor's memory hierarchy and thereby in long execution times.

In the LOCUS project we addressed this problem using an AI acceleration middleware (SOL) that, sitting in between the Deep Learning Framework and the underling hardware, analyses

and then optimizes the computation graph of the Neural Network, taking into account the architecture of the target hardware.

2.6.1 Background on Neural Network Processing

The landscape of Deep Learning framework is very vast e.g.: TensorFlow, PyTorch, MxNet, CNTK, Chainer, etc.

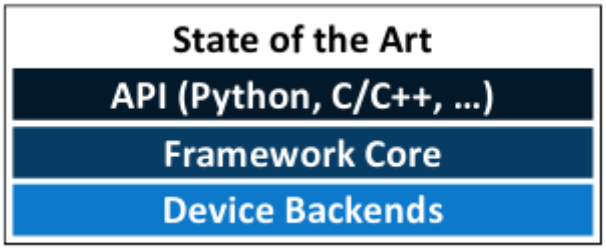


Figure 37: Deep Learning Framework common architecture

All these frameworks share the design of their internal architecture (Figure 37).

The common architecture is composed by:

- a Frontend API, usually written in, e.g., Python, C/C++or Java, that maps onto a C-based core.
- A Framework Core, that handles all of the framework's functionality. It manages the data on the devices and processes the computation graphs of the neural networks by issuing function calls to device specific backends.
- Device Backends, that either rely on hand optimized compute kernels, written specifically for the framework, or on vendor specific libraries.

When it comes to Device Backends, the landscape it is even more vast; since NNs are very compute intensive, hardware vendors provide hand optimized libraries to accelerate such workloads on their hardware: Intel provides DNNL for their CPUs and GPUs, NVIDIA provides CUDNN, AMD provides ROCm MIOpen, and ARM provides ARMCL. These libraries all come with similar functionalities, although ARMCL only supports functions for inference. Aside from this, NNPACK provides functions for X86 and ARM64 CPUs, although its performance is no longer competitive with respect to DNNL or ARMCL. All frameworks usually rely on these libraries in order to leverage their superior compute performance.

Although the hyper parameters of NN layers are well defined, the APIs of Device Backends are not, making it necessary to write separate code for each of these libraries.

2.6.2 SOL AI acceleration middleware

SOL (Figure 38) is an optimizing middleware for inference and training that transparently integrates into existing AI frameworks (e.g., PyTorch or TensorFlow).

It is designed to have a very small programming footprint towards users: a data scientist only needs to add a few lines of code to enable SOL. Beyond transparency, SOL is explicitly designed to support multiple AI frameworks (frontends) and hardware devices (backends), including X86 and ARM64 CPUs, NVIDIA GPUs and the NEC SX-Aurora vector processor. The core of SOL targets SIMD architectures in order to leverage common features across all of these hardware architectures within the same code base, and it provides a large set of optimization patterns that can modify the network's structure, improve the reuse of data buffers, optimize the use of the device's memory hierarchy and computation units.

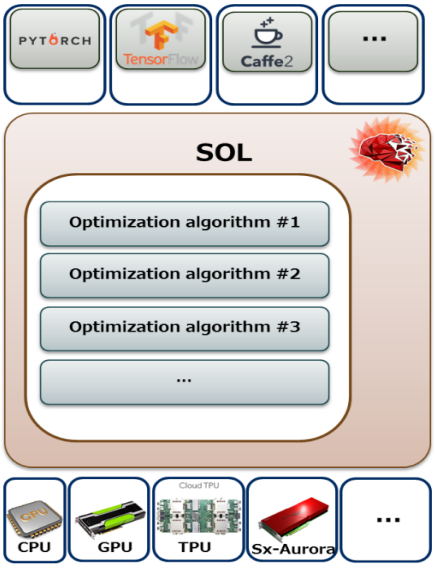


Figure 38: SOL AI acceleration middleware

The SOL AI acceleration middleware essentially consist of three main components: Compiler, Runtime and Deployment.

The SOL Compiler, the key component of the SOL middleware, at a higher level performs the following operations:

- analyses the Neural Network structure to extract its computation graph;
- performs transformation on this graph to generate more efficient computations taking into account the structure of the graph itself;
- finally, it generates computation libraries that are specific to the newly designed computation graph and to the target hardware.

The SOL Runtime oversees the transparent integration with the Deep Learning Frameworks:

- Loads the optimized kernel functions generated by the Compiler
- Connects the kernels with the framework's memory allocation system, allowing direct read/write access to the framework's tensors
- Maintains all communications between SOL, the framework and the target devices' APIs.

The SOL Deployment is a special mode of the SOL Compiler, that extracts the neural network from AI frameworks to deploy it into a library that can be integrated into a user application. The generated libraries only contain the neural network execution functions, parameters and a minimal set of helper functions, which significantly reduces the size of these libraries and simplifies their integration in the applications that need them.

The SOL Deployment is a special mode of the SOL Compiler, that extracts the neural network from AI frameworks to deploy it into a library that can be integrated into a user application. The generated libraries only contain the neural network execution functions, parameters and a minimal set of helper functions, which significantly reduces the size of these libraries and simplifies their integration in the applications that need them.

The SOL AI acceleration middleware can be easily integrated into the LOCUS Machine Learning pipeline. The training and inference components of the ML pipeline are the target components for the SOL optimization. SOL plus the overlaying Deep Learning framework can be provided in the form of Virtual Machine or Docker Container for both development and serving.

2.6.3 SOL Operations & Optimizations

From a user perspective SOL is completely transparent. In fact, the users have to simply include the SOL middleware (as in Figure 39) as python library and call the optimize function to optimize the ML model and perform its training or inference using the chosen Deep Learning framework and Hardware.

```
import torch
from torch.autograd import Variable
from torchvision import models
from sol.pytorch import optimize

model  = models.__dict__["..."]()
input  = torch.rand(32, 32, 224, 224)
model  = optimize(model, input.size())
output = model(input)
```

Figure 39: SOL usage

Figure 40 shows instead the internal operation performed by the acceleration middleware.

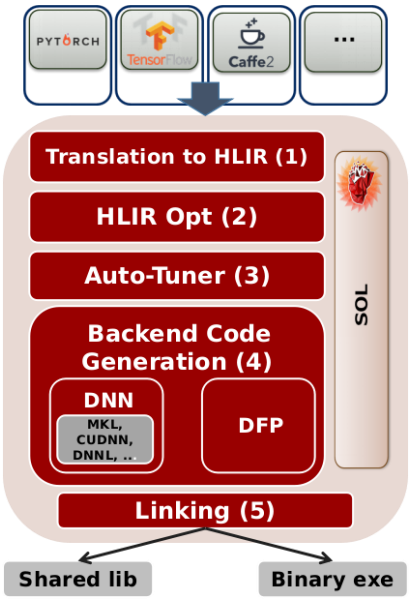


Figure 40: SOL Operations

The call to the SOL *optimize* function triggers the SOL Compilers.

The first step (Figure 40) is to extract the computation graph from the framework and translate it into SOL's own graph High Level Intermediate Representation (HLIR). The HLIR used by SOL identifies tensors by the purpose (None, Channel, Pixel) and dimension, while common IR for ML identify tensors using only the dimension

This bounds optimization to only a specific memory layout. Moreover, it enables SOL to make it easy to implement layers independently of the used memory layouts. Furthermore, the SOL HLIR can easily evaluate if using the same layout in forward and backward pass (i.e., inference and training) is faster than using separate layouts.

In the second step (Figure 40) the compiler works directly on the HLIR of the computation graph and applies general optimization e.g.:

- mathematical optimization, such as removing ReLU layers (i.e. max(x, 0)) that are followed or preceded by MaxPooling layers when the minimum value of the MaxPooling its set to 0.

- Compute graph optimization, such as layer reordering when the final mathematical results is not changed but the new order results in better data reuse.

In the third step (Figure 40) the computation graph HLIR is replicated for each available Device Backend, to apply device-specific optimization. For each HLIR computation graph the auto tuning phase is performed. For each layer type the best optimization algorithm and optimization module is selected and applied. The main optimization algorithm implemented in SOL is Depth First Parallelism (DFP). The DPF optimization relays on the observation that Deep Layer frameworks use a breath first parallelism, where the NN computation graph is executed in per layer fashion where each NN neuron in the same layer is executed in parallel (Figure 41). With this approach the computation of each layer generates large temporary data that cannot fit in GPU's/CPU's small caches.

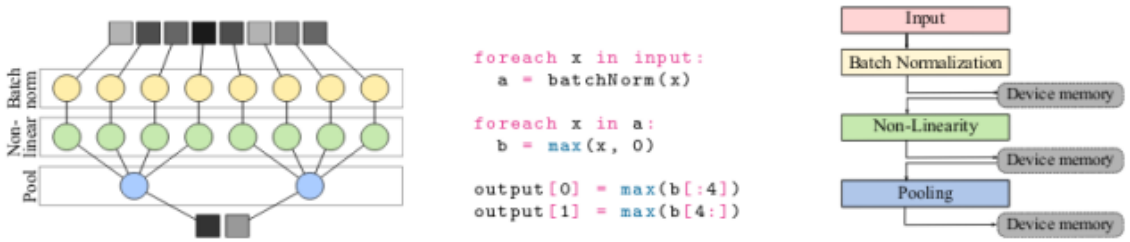


Figure 41: per-layer execution

The DFP approach (Figure 42) instead detects independent paths in the computation graphs and aggregates these paths into independent processing blocks that generate intermediate data that can fit into the hardware cache.

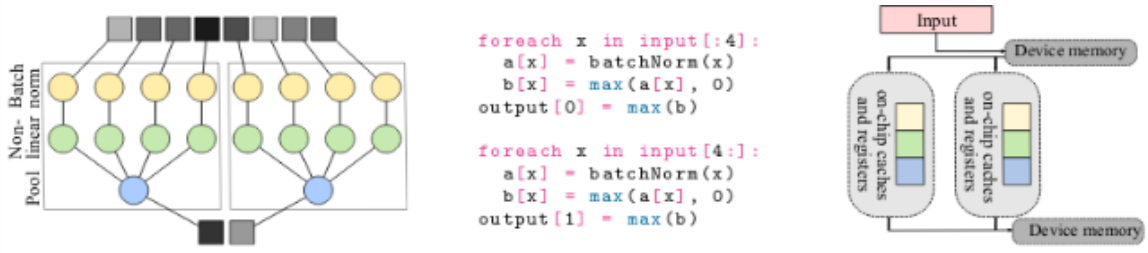


Figure 42: DFP

Moreover, DFP generates code that minimizes the number of nested loops while efficiently mapping these onto the SIMD architecture of the hardware. The DFP module can handle arbitrary SIMD architectures from very short AVX instructions to very long vector processor instructions, and it leverages features such as Shared Memory and SIMD-groups (warps). For layers where the DFP optimization cannot be used, the SOL compiler can select a different optimization algorithm. For instance, depending on the Device Backends, SOL can determine

the optimal memory layouts for the given data (e.g., DNNL prefers blocked memory layouts) and takes care that data are always given in the optimal layout to the layers, while trying to minimize the number of reorder operations. When it is possible, SOL also merges layers. In case we have multiple optimization algorithms, SOL either uses heuristics or runs a very short auto-tuning workload to determine the best combination given the layers and their hyperparameters. Note that the selected Optimization algorithms can be different for the training and inference phase.

After all layers have been assigned to an optimizing module, in the fourth step (Figure 40) SOL translates the HLIR into device backend specific code and compiles it for the target devices. This entire optimization procedure requires usually less than 1 min (including the auto-tuning) and only needs to be repeated if the input size of the network or its structure change. After compilation (step 5 in Figure 40), SOL Compiler can generate either a shared library or a binary executable, depending on the requested configuration. When the shared library is generated the SOL runtime is used to inject the custom model generated into the AI framework. The user will keep working using the Deep Learning framework in the same way he would use it with a native model, with the difference that the framework will internally call the SOL Runtime and execute the optimized implementation. The binary executable instead is generated in the case the model has to be directly executed or included in a final deployment application.

Figure 43: SOL benchmark

Figure 43 shows some initial micro-benchmarks of the SOL AI acceleration middleware. In this case we used some well-known NN architectures to measure the reduction of inference execution time provided by SOL acceleration. The benchmarks are executed using PyTorch as Deep Learning Framework and a x86 CPU (i7 9600K) and an NVIDIA GPU (RTX 2080). CPU

Inference time is reduced by a factor of 2.33 to 7.51, while GPU Inference time is reduced by a factor of 1.09 to 4.41.

3 LOCUS Data Platform Technologies & Virtualization

3.1 Scope for LOCUS Data Platform Technologies & Virtualization

Developing an efficient data-centric and data-intensive backbone for location-based analytics solution, is of paramount importance to fulfil the requirements stated in the introduction and the global vision of the LOCUS project.

This section does not aim at providing a detailed cookbook on how to install, configure and use libraries and software solutions. Instead, it focuses on providing a reasoned exposition of the expectations, with regards to data management, that arise from a distributed microservices-based architecture for location-based analytics, in addition to the trade-offs deemed necessary for the particular scope of LOCUS support for vertical applications.

Further details will be in the scope of upcoming D5.4 “Prototype of the localization & analytics as a service solution” and D2.5 “System Architecture, final version”, to cover implementation considerations and pertinent aspects beyond focus on verticals.

The microservices-based architectures, as described in previous sections and other deliverables, imposes challenges on a traditional database-oriented dataflow and data management, characterized by “Atomicity, Consistency, Isolation and Durability” (ACID), due to the distributed nature of service interactions with data and the high number of services relying on efficient and intensive data flows. A typical LOCUS pipeline (Figure 44) would require provisioning of data of a variety of types (Geolocation, Mapping, Social, RAN, ML Models, etc) and attempting to choreograph the operation of distributed services composing this pipeline in an ACID paradigm is just not possible within the stated requirements of flexibility, scalability, efficiency, etc.

Hence the reliance on a non-transactional view of data interactions, characterized by “Basic Availability, Soft-state and Eventual consistency” (BASE).

3.2 LOCUS Persistence Module – Data Lake

The LOCUS data platform operates in both streaming and batch persistence mode as it is the paradigm for various big data platforms and data lake implementations. The storage layer requires to be a high performance, distributed, big data storage infrastructure to support:

- Persistence of static meta-data from connected 3GPP network NME.
- Persistence of historical metrics/KPIs/measurements from a connected 3GPP network NME.
- Persistence of geolocation /localization information as it derives from the WP3 LOCUS analytics function activities.

- Persistence of intermediate analytics results from between LOCUS functions for inter-function data communication.
- Persistence of high-performance analytics schemas for direct queries to serve the end user and API consumers.
- Distributed file system support to save unstructured data of various forms that will act as external state for the LOCUS function (if applicable)

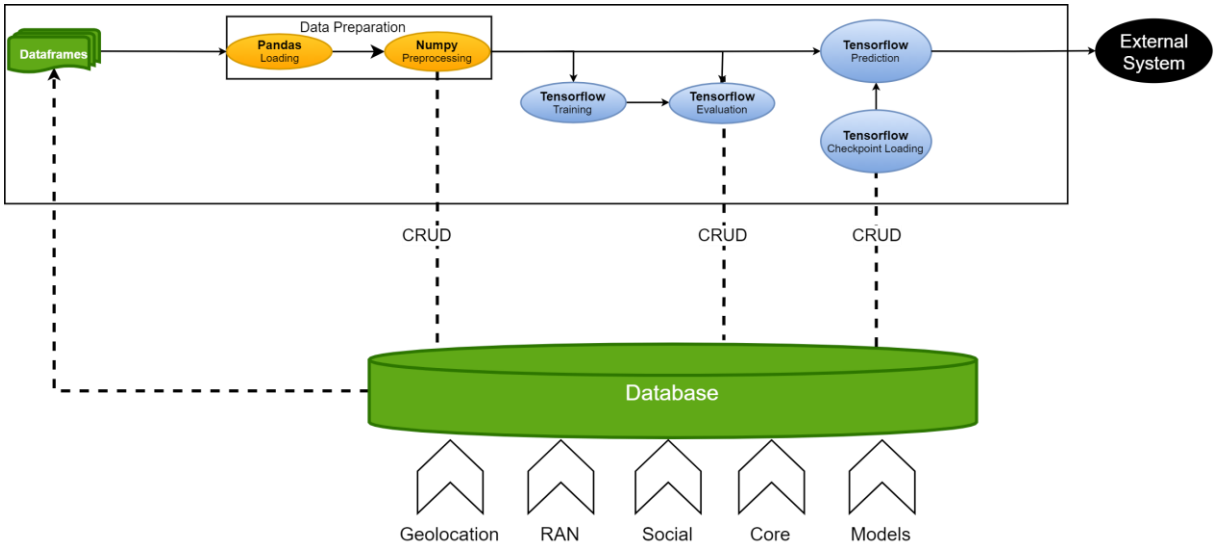


Figure 44: Example of a Database-oriented flow

A variety of open-source technologies exist in the big data ecosystem, however based on the number of references on publicly available systems, we can see that there is a convergence on a specific set of technologies:

- Apache Hadoop Distributed File System (HDFS)[15]: used as a distributed file system for structured and unstructured data.
- Apache Hive: used as the data storage layer for the execution of Create/Update/Insert operations and meta-data storage.
- Apache Spark: an alternative execution engine for data manipulation and transformation that complements Hive.
- TrinoIO: A distributed query engine on top of multiple data sources that will speed up the data access on the LOCUS data module.

These technologies are suitable for virtualization on various aspects mainly because they are designed to work as a cluster. This is true for Spark (using spark slave nodes), Apache HDFS (data storage nodes) and trinoDB (using query engine nodes). The deployment of these nodes can be performed on top of OSM on customized images or in some cases they can be deployed on demand by their dockerized versions (particularly for the spark nodes).

The LOCUS data platform's schema is a very important aspect that need to be considered because it will act as one of the interfacing points between the various LOCUS functions

(Figure 45). Functions that will communicate via schema (Hive tables) will converge into a set of input and output tabular structures that will allow for great interoperability and code reuse. Such examples of convergence are:

- LOCUS WP3 Localization enabler/geolocation schema can be a unified schema that external tools (e.g., Viavi) or internal LOCUS WP3 localization functions can deposit their location estimations. In this sense, all the higher layer functions (e.g., WP4, WP5) will be able to interface with all the localization implementations due to the unified schema.
- LOCUS Platform 3GPP Network Entity Schema: Meta-data (static) of 3GPP network entities (cells, sites, probes) including their fixed location and coverage geo-json can also be a LOCUS data platform convergence point.
- LOCUS Platform geo-json persistence schema is a repository of all the geospatial shapes imported or generated in the LOCUS environment. They can be used as inputs or outputs for the various functions. The LOCUS analytics functions (WP4/WP5 activities like for example UC5 - Traffic Optimization) result into identified geo-shapes that exist within the geographical bounds of the 3GPP network. These shapes can then be used in conjunction with the LOCUS WP3 Localization enabler/geolocation schema to perform reverse geo-query and area correlations. These shapes can also be accessible by 3rd party applications that require them for visual (map) reporting and analytics (e.g. WP4 Network KPI geospatial distribution maps)
- LOCUS platform ML model registry schema: MLFlow and other similar software solutions provide a persistence layer for the LOCUS platform that is used for model management and model meta-data storage. Such system can operate on top of HDFS/Hive and reuse the same large-scale infrastructure. Also, the ML-related code implementation and data types will converge based on the primitives imposed by MLFLow, Sonet and the selected ML framework of the solution.
- LOCUS platform end-user analytics schema design principles: Different use cases from the WP4/5 require different data models which will feed the various end-user analytics. However, some table design principles can be applied in order to ensure that the query engine (TrinoDB or other) will be able to handle the large volume of data that is generated from the LOCUS platform. These principles in general are materialized views, denormalized, federated tables with special raw file types (e.g. RC, ORC, Parquet) which are optimized for this HDFS/Hive stack and provide very efficient queries especially when used with some sort of an application layer cache.

This Data-centric architecture inspired by the Lambda architectural principle has a very straightforward way of interfacing with its corresponding streaming context: standalone CDC (change data capture) processes on top of Spark or other software can convert the changes on these core/central tables into Kafka messages including the payload and the table schema

in various formats (JSON, Avro, XML). In this sense, all the core LOCUS platform table are directly converted into their respective Kafka topics, so the implementation of various LOCUS functions can be either based on the streaming context (Kafka consumer/producer) or on the batch context (Hive/Spark and Trino).

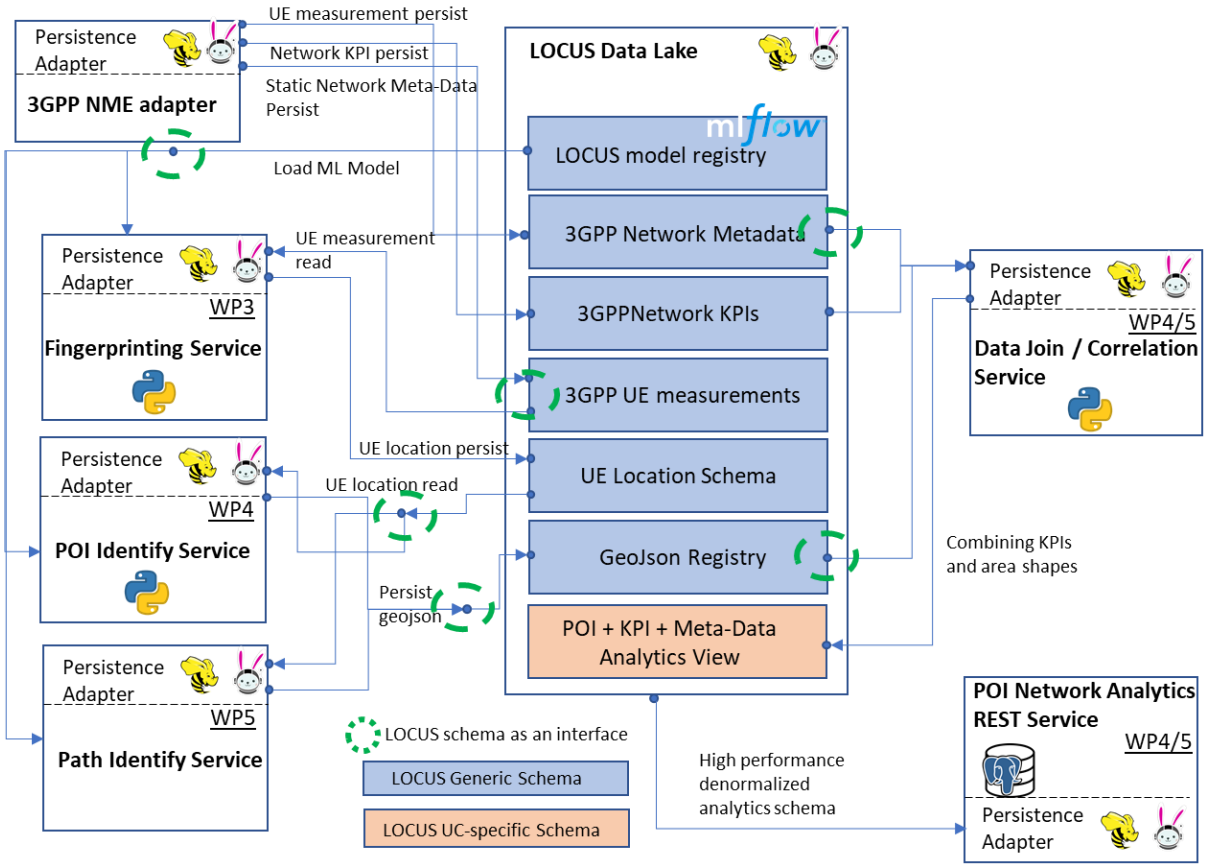


Figure 45: The LOCUS data schema as an interface point between functions

Each LOCUS Function will need to interface with the LOCUS persistence module for:

a) Definition of tables/schemas that it needs to be created for its operation.
b) Reading (based on SQL query) of data from an input source into an appropriate data structure for in memory processing (general candidates are python pandas/koalas DataFrames or Spark DataFrame)
c) Writing data from DataFrames into destination tables using append or insert.
d) Deleting data from destination tables based on SQL-based query.
e) Listing, Reading/Writing blob or unstructured data from a designated HDFS directory that is specific to this function.

For the purposes of efficient and easy implementation, these interfaces will be part of the LOCUS SDK. The best practices and implementation notes will be followed in order to ensure that the end user of the SDK will have separation of concerns between the actual analytics function they are implementing and the various platform operations. This will also assist in

the seamless deployment of the application in the LOCUS data lake by unifying the configuration (connection URL details) and other system aspects.

3.3 LOCUS Streaming Module - Data Movement

The LOCUS solution embraces Kafka all the way through, and in doing so it places broker-based messaging at the core of data movement and relies on it to enable a non-transactional paradigm best fit for the LOCUS microservices-based architecture. As will be discussed in the following subsections, Kafka provides an efficient and scalable messaging bus that allows for distributed services, real-time data transportation, fault-tolerance, and replication, on top of a versatile APIs to support a fine-grained interaction with LOCUS pipelines and services.

3.3.1 Pipelines and Data Movement

Pipelines of LOCUS Functions that implement a particular functionality, are composed at design time, using the LOCUS Functions Catalogue. These compositions are then commissioned into a services and pipelines catalogue, that is accessed at runtime, through the LOCUS external REST or Intent-Based API; as shown in Figure 46. The composition of LOCUS pipelines is an offline procedure, its specification is out of the scope of the current deliverable, but it can be envisaged either as a low-level manual intervention step, or GUI based one where a pipeline designer composes LOCUS functions together and assigns deployment configurations.

The designer is able to arbitrarily chain LOCUS Privacy, Security, Analytics, Persistence, Policy, or ML optimization functions, with corresponding function placement configurations, in order to define any number of pipelines which, when commissioned, are made available to external users for instantiation.

The LOCUS system will handle the deployment of necessary resources, such as Kafka topics, multi-site virtualized clusters if necessary, and all the necessary components for monitoring, persistence, replication, etc.

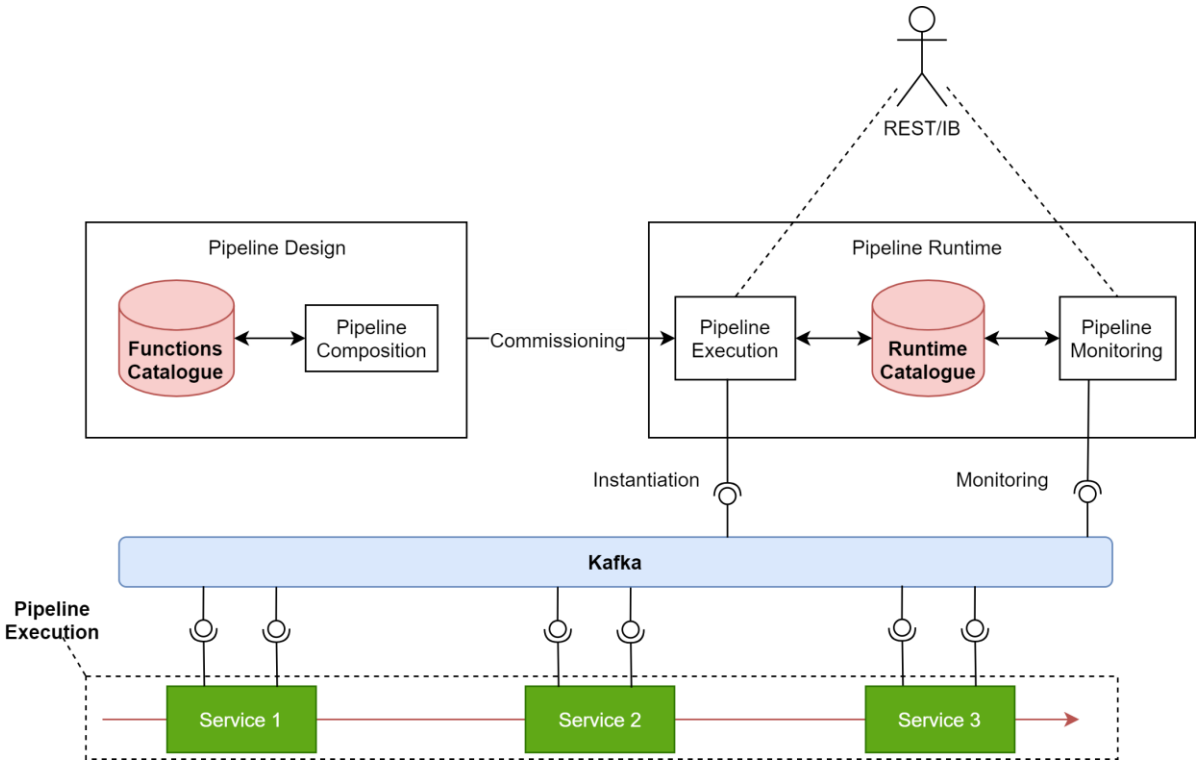


Figure 46: Pipelines Composition and Instantiation

Composing pipelines involves not only the chaining of functions to process data into a particular output, it's a step that also involves informed decisions regarding function placement, and service granularity.

Function placement means where to physically place the instantiation of a particular function throughout the network, and what set of functions can be encapsulated together as a single virtualized service. Regardless of the decisions made, there is always the need to also optimize data delivery and dataflow between services. In Figure 47 we see the same previous pipeline decomposed into a distributed set of services, deployed on physically separate parts of the network, with data interactions, such as movement and persistence, fully enabled by multi-site Kafka deployment. The distribution of the pipeline throughout multiple sites for this particular scenario is designed to:

- Optimize location information pre-processing and aggregation at the point of collection on the edge.
- Transportation of lower volume aggregated and normalized location information.
- Pre-processing of location information is done on a site with the sufficient computational resources.
- Consumption of data and execution of ML training is done on GPU enabled servers.
- Consumption of data and inference in addition to analytics and intermediary data persistence is done on a dedicated cluster.

This example configuration is only indicative of the types of considerations that could be made when composing pipelines, with the point being that the Kafka bus can adapt and enable more complex or simpler deployments.

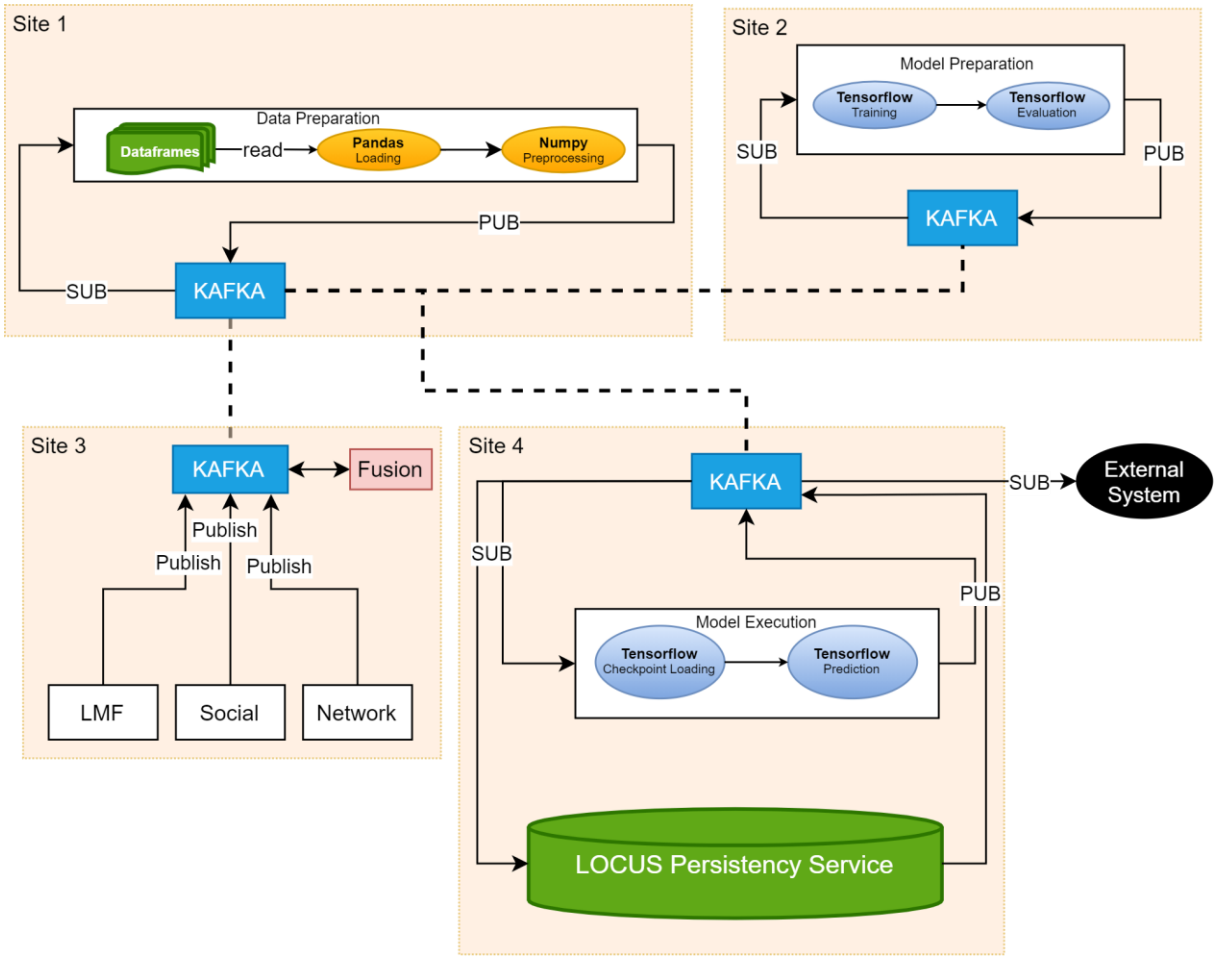


Figure 47: Kafka powered distributed pipeline

3.3.2 The Message Broker

As stated before, we have adopted Apache Kafka as the data movement backbone for the LOCUS Location Analytics as Service Solution. Kafka is an open-source message bus for stream-processing that provides data flow and message passing capabilities at high throughput and low-latency and well suited for real-time data processing.

Kafka is widely used, as it powers the Linkedin infrastructure [16], and Uber [17] uses it to process trillions of messages and daily deluges of data in a complex multi-region deployment. The Data Movement as a Platform Project (DMAAP) part of The Open Network Automation Platform (ONAP), is built on top of Kafka to enable reliable message passing and real-time data movement in support for analytics and network management. Moreover, Kafka is particularly suited for virtualized networks and cloud-native deployments and can be easily deployed on

containers and Kubernetes. In summary, Kafka is the natural choice to consider when designing a distributed real-time analytics solution as in the case of LOCUS.

A Kafka setup will consist of the configuration of a Kafka cluster with a set of brokers, and the use of a set of APIs to create producers and consumers that publish and subscribe to data on the Kafka brokers by means of Kafka topics. A Kafka broker, as shown in Figure 48, is basically a server that is the main point of contact for services interacting with Kafka. Producer and Consumer services publish and subscribe to data topics on the broker. A collection of Kafka brokers defines a Kafka cluster.

A Kafka topic represents a data feed name that is given to categorize a particular data stream. Services and Applications publish and fetch data by subscribing to data topics.

When creating topics on a broker, Kafka implements a concurrency mechanism that allows efficient scalability of consumer/producer interaction with data topics, which consists of dividing each topic into a set of partitions (see Figure 48) and allowing as many consumers as partitions to be served concurrently while scaling linearly with the consumption load.

Apache Zookeeper (see Figure 48) acts as an authoritative synchronization service and centralized store for the Kafka deployment metadata. It keeps records such as related to Broker states (whether dead or alive) topic replicas and elected leader broker, which are necessary to maintain leader-follower relationship across partitions and brokers and guarantee requests fulfilment in case of failure of partition leader. It also maintains up-to-date lists of all defined topics and their configuration, partitions, replicas, and Access Control Lists.

With that said, the partition count and replication factor (number of brokers in a cluster) are two important parameters that need to be configured for optimal data replication, fault-tolerance, scalability, and overall performance.

In Kafka, topic partitions will be replicated throughout all the brokers within a cluster. And a single broker will be elected as a leader of any given partition, so that all requests from producers and consumers will be directed to the leader partition. Write operations will then be synchronized to all replicas, and in case of a failure at a partition leader, a new in-sync replica will be elected as a new leader.

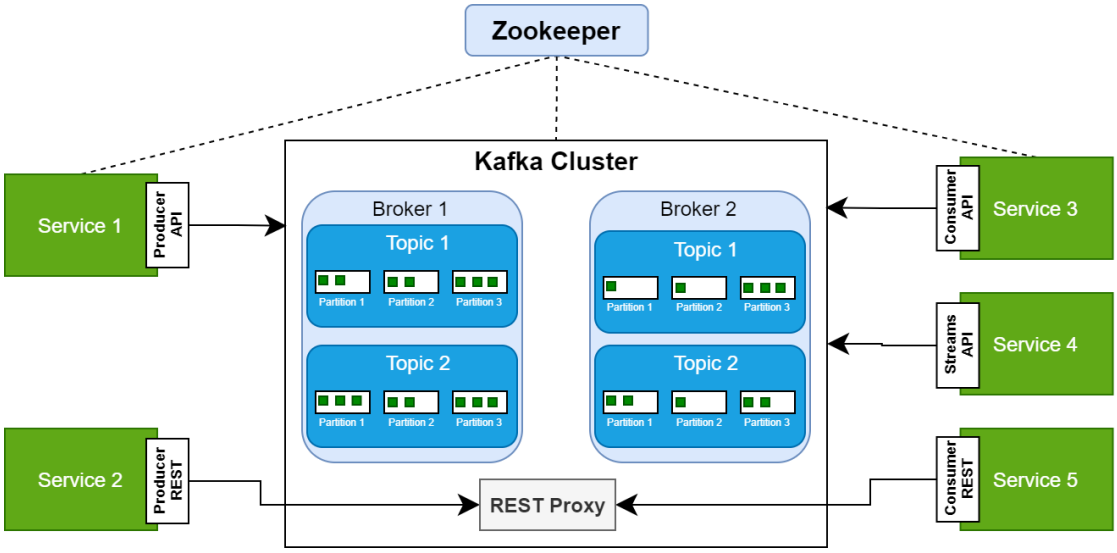


Figure 48: Kafka Bus

Producers in this ecosystem are running applications that act as sources for data streams pushed to Kafka topics, while consumers are running applications that subscribe to topics and received data streams. The Kafka project provides native Producer API[18] and a Consumer API[19], as shown in Figure 48. In many cases a single application/service needs to operate as both a producer and consumer and interact with Kafka clusters in a closed loop, for which Kafka provides a Streams API [20], that incorporate producer/consumer capabilities with advanced data processing functions (e.g. joins, aggregation, windowing and event-time processing). In addition to these native APIs, applications can also interact with Kafka clusters with a RESTful interface using a REST Proxy feature[21]. Finally, in Figure 49 an example sequence diagram of multiple services interacting with a Kafka bus is illustrated.

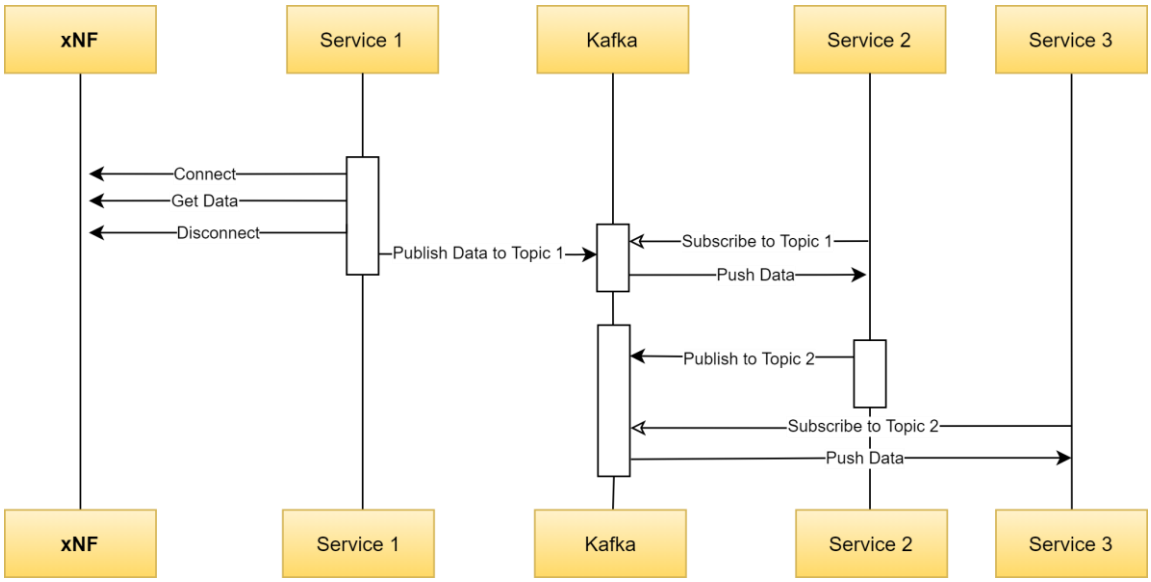


Figure 49: Kafka-based service interaction

3.3.2.1 Batch vs Stream processing

Many of the investigated LOCUS UCs would naturally rely on real-time data processing, such as UCs incorporating mobility tracking, geo-fencing, location-aware content-serving, emergency localization, etc. Hence the emphasis on a solid support for real-time streams processing provided by Kafka. With that said, many use cases still don't necessarily require real-time system operation, and a database-based batch processing can be employed to perform exploratory analysis on acquired data for instance.

Batch data is basically a time-bound view of an unbounded data-stream, so in its essence batch processing only differs from stream processing in the expectation regarding the time-window of data to be processed. Kafka provides the Streams DSL API[22] that allows interaction with data in a time-bounded fashion, in addition to the ease of connectivity to traditional database systems, which allows for full support for traditional batch-processing use cases.

3.3.3 Data Collection

Different use cases will require access to possibly more than just geolocation information, such as many forms of auxiliary data necessary to run Analytics. Such data will probably be stored in external systems using any of the available data formats and database technologies. In addition to that, UCs could also require persistence of Analytics output and intermediary processed data, for example for non-real-time Analytics, and long-term storage of results.

LOCUS services rely on the system's ability to collect and store such data whenever needed, either for real-time consumption or long-term data convergence/persistence. We envisage decoupling the problem of collecting external data from the internal logic of services and pipelines, and for that we rely on Kafka Connect[23].

Kafka Connect, as depicted in Figure 50, allows the integration of all sorts of data systems with the Kafka bus. Consequentially, Kafka consumers are able to process data collected from databases, file systems, key-value stores, etc. Kafka Connect project already provides connectors for commonly used storage technologies, whether the aim is to use them as source connectors or sink connectors. A non-exhaustive list of such collections already available is:

- Active MQ Source Connector
- Amazon S3 Sink Connector
- Confluent Replicator
- Elasticsearch Sink Connector
- FileStream Connectors (Development and Testing)
- IBM MQ Source Connector

- JDBC Connector (Source and Sink)
- JMS Source Connector

In addition to the already supported data systems, which easily allow LOCUS services to pull external data and push data to whatever storage infrastructure available, Kafka Connect also provides an API to support interfacing for other external data systems. This API is the key enabler for LOCUS to interface with proprietary systems, or to collect data from other networks, such as 5G Network elements.

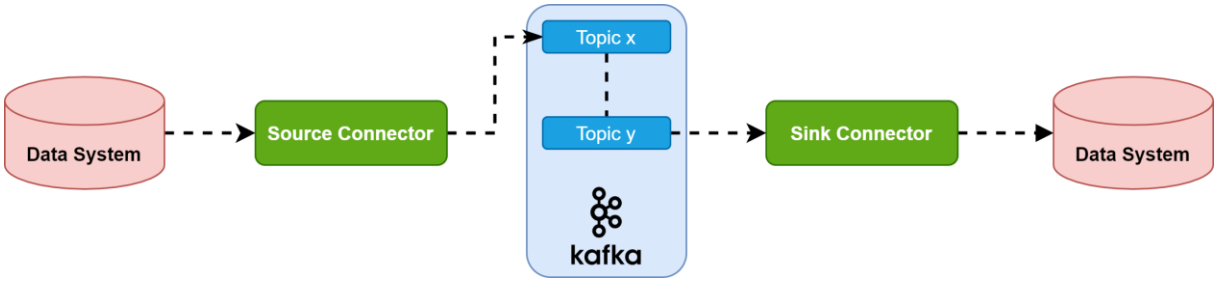


Figure 50: Kafka Connect

3.3.4 Multi-tenant deployment

As mentioned previously, in the proposed microservices-based architecture, there is a solid argument for supporting the ability for microservices to access data available within their local deployment site, in the context of multi-site deployment (such as Edge plus Core deployment), as in Figure 51. This allows to support local data pre-processing, aggregation, compression, and specific algorithms such as positioning and Time-series imputation. The capability to access data within the local deployment site is particularly important in the case of massively produced data on the edge, that could be transported more efficiently after processing.

Kafka supports multi-site deployments using either Confluent Replicator or MirrorMaker [24] tool. These tools, allow the deployment of remote Kafka clusters in addition to a centralized one. The different clusters are completely independent, although both tools handle synchronizing topics bi-directionally or uni-directionally, this way deployed services can subscribe and publish to Kafka topics regardless of service location or Kafka producer location.

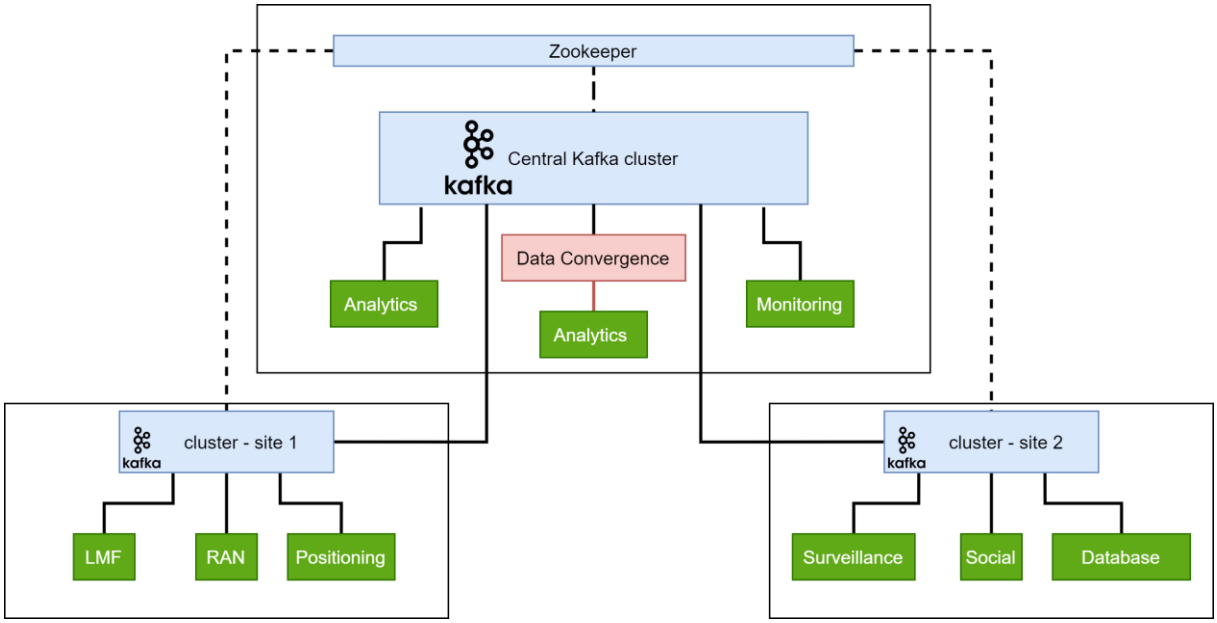


Figure 51: Kafka Multi-site deployment

4 Localization-based Analytics as a Service API

LOCUS aims at offering localization analytics as a service to the Smart Network Management and 3rd party vertical applications, and for this reason a dedicated API layer is included in the LOCUS platform to expose such services towards the application layer. This layer, implemented in the form of an API Gateway, represents the entry point of the LOCUS platform, thus implementing its northbound interface. On the one hand, the LOCUS MANO, as described in deliverable D4.3, is responsible to model the localization analytics functions and services as respectively Virtualized Network Functions (VNFs) and NFV Network Services, taking care of their automated lifecycle management (for on-demand deployment, configuration and runtime operation over the edge/core 5G virtualized infrastructure) by offering dedicated management APIs to trigger specific operations (instantiation, scaling, update, termination). On the other hand, the Localization Analytics as a Service API Gateway is responsible to provide access to the virtualized analytics functions, pipeline services and ML model predictions when they run in the LOCUS edge/core virtualized infrastructure, exposing them as services to the Smart Network Management and 3rd party vertical applications.

However, various data-intensive operations have to be considered and coordinated as well, including:

- Ingestion of Service-Based interfaces
- Pre-processing and persistence of measurements and other generated data (raw or intermediate)
- Execution of analytics functions and persistence of their results
- ML-related functions which require special lifecycle management
- Transformation on data schemas for analytics-ready query engines

In practice, it is required to consider specific data operation control functionalities to coordinate the actual execution of the analytics functions and services pipelines. This can be done through pipeline orchestration functionalities that decouple the API Gateway exposure and data consumption from the internal analytics service details and virtualization aspects (thus including its deployment and management as NFV Network Services and VNFs).

Therefore, the API Gateway, the pipeline orchestration and the LOCUS MANO have to be considered as complementary key functionalities of the LOCUS platform that together allow to implement the LOCUS localization analytics as a service model, enabling the LOCUS applications to subscribe or request for the activation of specific analytics services and ML pipelines and have transparent access to the data and predictions produced.

The following subsections first provide an overview of the Localization Analytics as a Service concept and architecture, followed by the description of the RESTful APIs and operations

exposed by the LOCUS platform, and the intent-based approach supported by the API Gateway.

4.1 Concept and architecture

As said, the Localization analytics as a Service API represents the northbound interface of the whole LOCUS platform, and it aims at exposing a flexible and open interaction with the localization, analytics and ML functions available in the LOCUS platform. From a practical perspective, it enables the external applications to consume localization analytics and ML predictions on-demand and when they require for it.

In particular, the main idea is to decouple the exposure of the analytics services from the internal analytics service data control and pipeline orchestration, as well as from the NFV Network Services and VNFs lifecycle management provided by the LOCUS MANO. This allows to hide the complexity of the internal data requirements and data exchange among the involved functions, as well as of the LOCUS MANO procedures and data models described in D4.3 (i.e. how analytics functions, service and ML pipelines are modelled as NFV Network Service Descriptors – NSDs – and VNF Descriptors – VNFDs). Therefore, the external LOCUS applications can focus on querying the available analytics services and subscribing to those that produce analytics data and ML model predictions that suites their requirements.

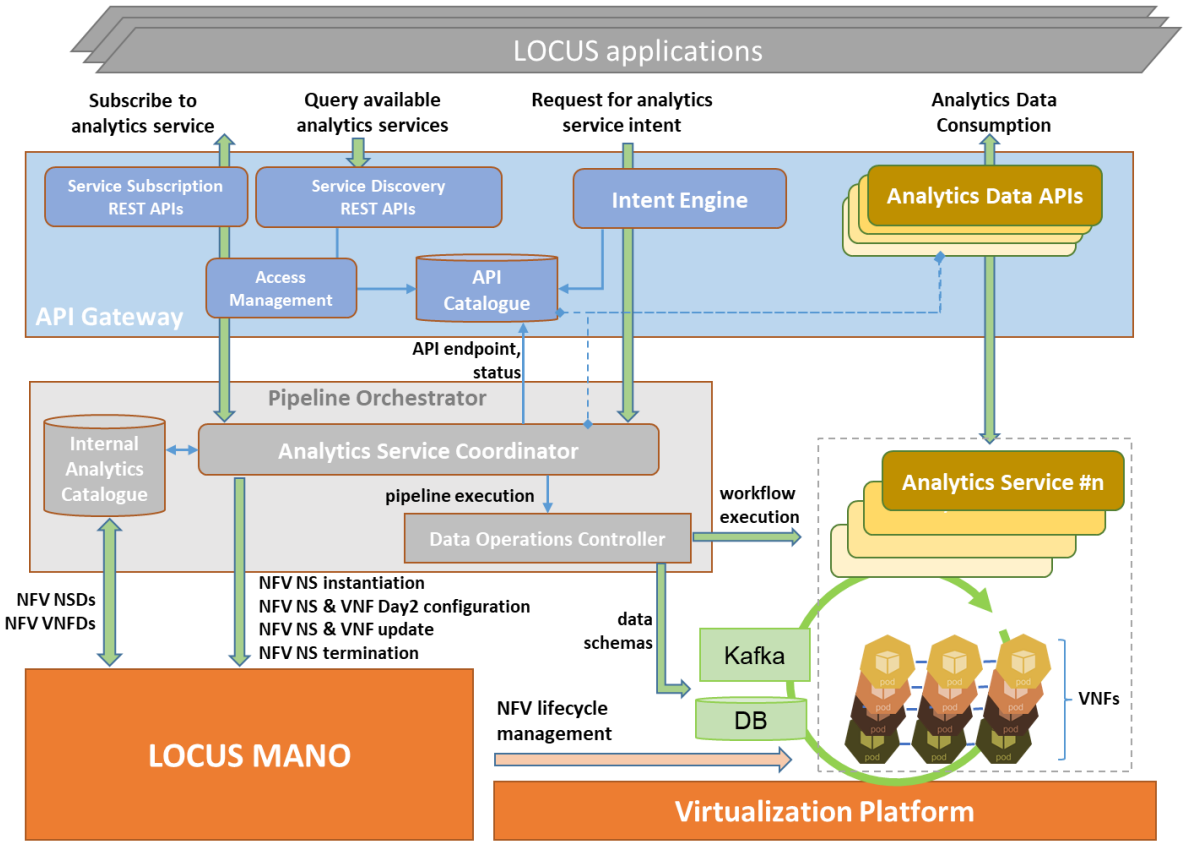


Figure 52 Localization Analytics as a Service architecture

As shown in Figure 52, the Localization Analytics as a Service approach is implemented by the integration of the API Gateway (for exposure of available services and enable their consumption) with the Pipeline Orchestrator (taking care of the internal analytics service data operations control). They sit on top of the LOCUS MANO specified in deliverable D4.3, accessing its external APIs for requesting operations over the NFV Network Services modelling the LOCUS analytics services, functions and ML pipelines for their deployment in the virtualization platform.

In particular, the API Gateway provides the following main functionalities towards the external applications:

i. discovery of available analytics services, that can be queried through dedicated RESTful APIs that provide access to the API Catalogue.

ii. subscription to catalogued analytics services, enabling the external applications to request for the activation of specific analytics data and ML model predictions.

iii. analytics service intents, allowing LOCUS applications to express their required analytics service following an intent-based approach, thus providing a simplified service goal linked to a declarative abstract policy focusing on what the application wants to achieve without detailing how (see section 4.3.1).

iv. per analytics service dedicated and isolated exposure of output data, to be consumed by the LOCUS application in the form of explicit queries or data stream, depending on the specific analytics service capability (i.e. if the analytics service/function produces its output as a continuous stream, or it has to be directly invoked or queried).

Therefore, in addition to the separation between the analytics service exposure/access, the pipeline orchestration, and the LOCUS MANO NFV procedures and data models, the API Gateway clearly separates by design the discovery and subscription to available analytics functions and services from the data consumption phase, following a full asynchronous approach. With reference to Figure 52, the functional blocks in blue of the API Gateway implement the analytics service exposure functionalities, while those in yellow/orange take care to enable per-service analytics data output consumption endpoints. In turn, the Pipeline Orchestrator takes care of hiding the internal analytics functions and services details, in terms of data operations control and deployment of the involved functions as VNFs in the LOCUS edge/core 5G virtualized platform, while gluing these with the API Gateway operations to provide seamless localization analytics as a service.

4.1.1 API Gateway components

A functional description for each of the API Gateway building blocks shown in Figure 52 is provided below.

The API Catalogue maintains the information related to the available analytics services within the LOCUS platform, and it can be either directly accessed (to consume the related output data) or activated through a subscription process by external applications.

Indeed, the API Gateway, and the Localization Analytics as a Service approach as a whole, allows both on-demand as well scheduled execution of analytics services and functions. On the one hand, having an execution schedule for the various operations that concerns the deployment and data-intensive tasks of analytics services and functions allow for performance tuning built-in the existing infrastructure from processing and memory resources to the optimized data lakes. Following this paradigm, each LOCUS analytics function pipeline can be executed according to a schedule defined offline by a LOCUS platform manager/admin as part of the design phase, taking into consideration the logical requirements of each function and therefore the resulting sequential (or parallel) operations to be executed/refreshed in a fix rate according to each UC's preferences. On the other hand, an on-demand approach enables a more flexible and open interaction with external applications, deploying and executing analytics services and functions pipelines when they are required and can be consumed.

In practice, the API Catalogue stores the available Analytics Data API endpoints (as per Figure 52) that are available to external applications to consume the analytics services and functions output data. Together with that, for each available analytics service, the API Catalogue maintains a set of metadata to describe the service itself, in terms of analytics functionalities provided, format/schema of the output data generated and how to consume it (e.g. through RESTful operations or by subscribing to a message bus for continuous data stream). To link the API Catalogue content with the Internal Analytics Catalogue one for a given available service, a unique service identifier is used. For those analytics services that are not yet deployed and executed in the LOCUS platform, the API Catalogue stores only the metadata, allowing external applications to retrieve information on what is the service and how to possibly consume it. Table 12 below provides a brief description of the information stored in the API Catalogue for analytics services exposed to external applications. In particular, the list of attributes describing and modelling a given service is reported.

Table 12: API Catalogue Analytics Service information model

Analytics Service attribute	Description
ID	Unique identifier of the service in the LOCUS platform. It references the correspondent analytics service in the

	Internal Analytics Catalogue of Figure 24. It can be used by LOCUS applications to subscribe to this analytics service
name	The name of the analytics service (e.g. the name of the related NSE UC)
type	It specifies the type of the analytics functionality provided. It is mapped with the NSE UCs described in section 2
description	Textual description of the analytics service
exposure format	It is the available and supported data exposure and consumption options for the service. This attribute is relevant when the analytics service is not yet deployed and executed in the platform. Two main options are possible: • query based, with the API Gateway exposing a dedicated REST endpoint for the LOCUS application to consume data on-demand • message bus based, with the API Gateway exposing a Kafka topic for consuming the data as a stream
output data schema	It specifies the data format produced by the analytics service as output, e.g. in the form of a JSON schema
status	Status of the analytics service, one of: • ready: if the service is ready and output data can be consumed • not-ready: if the service is not deployed and not in execution in the platform • error: the service is in error condition and cannot be consumed
API endpoint	When the analytics service is already deployed and running in the LOCUS platform, this attribute specifies how to consume the output data (e.g. if it is a pre-scheduled analytics service execution, or an on-demand service already activated by the same or other external application), i.e. the Analytics Data API (as per Figure 24) to use. It provides the following information for the data consumption endpoint: • url: the URL where the analytics service output will be available through REST GET operations, in case of query-based exposure format

	• topic: the Kakfa topic where the analytics service output will be published, in case of message bus-based exposure format • authentication: how to have authenticated access to one of the two endpoints above (e.g. details for HTTP basic or OAuth2 authentication[25])

It is worth to mention that the API Catalogue content is populated through offline onboarding procedures performed by the LOCUS platform manager/admin, as described in section 0. In addition, for services activated and executed on-demand, a runtime update of the API Catalogue content is performed by the Analytics Service Coordinator to populate the API endpoint field. Also, the Analytics Service Coordinator takes care to update the status attribute for each service (i.e. ready, not-ready, error).

The Service Discovery REST API allows the external applications to interact with the LOCUS platform. In practice through this API they can have access to the API Catalogue content to retrieve the available services that can be directly accessed, or to which they can subscribe to consume the related output. In practice, the Service Discovery REST API exposes the available analytics services through explicit queries (i.e. HTTP GET requests), so that the external LOCUS applications can retrieve both the full list of available services in the platform, as well as specific information and details on individual analytics services according to the Table 12.

The Service Subscription REST API allows the creation of dedicated per-application subscriptions to activate on-demand analytics services that are available in the API Catalogue but not yet deployed and executed (i.e. they are in “not-ready” status and thus do not have a usable API endpoint). Based on their specific localization analytics requirements, LOCUS applications can then select one of the available services and then subscribe to it. Such subscription is processed within the API Gateway and forwarded to the Analytics Service Coordinator to create (when required) a new instance of the analytics service through the interaction with the LOCUS MANO, and then automatically expose the data produced towards the LOCUS application through the dedicated Analytics Data API. In particular, as part of the subscription process, the given application has to provide at least the following information:

- The identifier of the analytics service to subscribe to,
- The type of analytics data exposure (i.e. RESTful endpoint or message bus), when multiple options are available for the given service,
- A notification endpoint to be used by the API Gateway to inform the external application when the analytics service is ready to be consumed. It is normally including two main relevant attributes: i) a callback URL to be invoked by the API Gateway to send a HTTP POST notification message (with result of the operation and details of the

Analytics Data API to use), ii) authentication credentials or information to be used by the API Gateway to access the callback URL.

- Optionally some execution/operation requirements, e.g. in terms of scheduling of service execution, real-time or specific latency requirements (that could require edge deployment), geographical constraints for service deployment, etc.

Upon successful subscription, the API Gateway provides to the LOCUS application as response an identifier of the subscription. At this point, an asynchronous analytics service activation request is forwarded to the Analytics Service Coordinator, which takes care to deploy and execute the analytics service pipeline. When ready, the external application is notified with the result of operation as described above.

The subscription identifier can be used by the LOCUS application to unsubscribe from the give analytics service, and thus trigger a termination of the exposure and consumption of the analytics service output data that is performed by the Analytics Service Coordinator.

The Access Management regulates the access of external applications to the API Gateway, and in particular to the content of the API Catalogue. First, it provides authentication mechanisms, with support of HTTP basic and token-based (i.e. OAuth2[25]) approaches. In addition, it implements identity management functionalities and provide dedicated API Catalogue views for each external application interacting with the LOCUS platform (i.e. each application can discover and subscribe to only the subset of analytics services it is allowed to access).

The Intent Engine is responsible for the implementation of the LOCUS intent-based approach within the API layer, thus enabling LOCUS applications to subscribe by expressing an intent in the form of an analytics goal. The Intent Engine is responsible to translate the intent into an analytics service available in the LOCUS platform (or even in available analytics functions to be interconnected in the form of a new service). The Intent Engine accesses the API Catalogue to map intents with platform capabilities, and if needed it interacts with the Analytics Service Coordinator within the Pipeline Orchestrator to trigger the activation of the analytics service in the virtualization platform (through the LOCUS MANO) and manage the exposure of the related output. More information about the conceptual LOCUS intent-based API approach is reported in section 4.3.

The Analytics Data APIs are dedicated data consumption endpoints that the API Gateway offers for each analytics service to let LOCUS applications consume the analytics data they need. Two alternatives are supported by the API Gateway to expose the analytics data outside the LOCUS platform: i) RESTful APIs and ii) Kafka message bus topics.

The REST APIs are managed and made available by the LOCUS platform (in particular by the Analytics Service Coordinator) at deployment and execution of the various analytics services, and the access to the query operations from the LOCUS applications are subject to

authentication, with support of HTTP basic and token-based (i.e. OAuth2[25]) approaches. The details of the endpoint to consume the analytics service output, in terms of URI and authentication related information, are available in the API Catalogue, and in case of on-demand subscription and activation it is returned by the API Gateway as part of the asynchronous notification message.

On the other hand, Kafka message bus topics can be also used to expose the analytics service output as a data stream, when it is allowed by the specific service capabilities and characteristics. The Kafka message bus is managed by the LOCUS platform and could be the same as the one used internally for the data movement as per section 0, with dedicated topics created for external application consumption of analytics data output. Also in this case, the details of this Kafka message bus (i.e. in terms of IP and secure access credentials) and related topic to use are made available in the API Catalogue as part of the API endpoint attribute of the related analytics service, and in case of on-demand subscription and activation it is returned by the API Gateway in the asynchronous notification message.

4.1.2 Pipeline Orchestrator components

As anticipated at beginning of this chapter, beyond the deployment of the analytics functions as VNFs, another important aspect to consider as part of the Localization as a Service concept is the data communication between the various functions part of the analytics service pipelines. In the LOCUS platform, on top of the virtualized resources represented by the VNFs deployed by the LOCUS MANO to host the various analytics functions, there is the need to provide a coordinated control of data operations among the functions in order to execute the analytics service pipelines according to their data constraints. These data control plane functionalities are indeed what the Pipeline Orchestrator (and in particular the Data Operations Controller of Figure 52) provides.

Data operations and communications can greatly affect opposing aspects of the LOCUS system: usability and performance. Integration via data sources (high performance big data lakes or Kafka topics) is the de-facto standard for high performance communications, assuming a large volume of data is involved (which is the case for the LOCUS localization functions and data sources considered in WP3). This way, functions are able to exchange data by reference (query or Kafka topic) on the LOCUS data persistence modules which allows for a better resource management and simpler interactions. As far as the usability of this approach is concerned, implementing new data consumption/production clients for each specific analytics function implemented in LOCUS will not be a stable and solid practice as it can produce multiple code blocks implementing the same operation that will need to be tested separately. For this purpose, a LOCUS function SDK is being designed in WP2 (and will be reported in deliverable D2.5) that will be a reusable library which will provide the best

implementations of all data manipulation tasks and will be developed in its own version-sensitive manner.

In particular, for the LOCUS analytics functions data operations (i.e. the service pipeline control plane), it is assumed that each LOCUS function includes a REST control module within its environment exposing predefined functions that acts as control points. These control points are extensible (based on the specific sub-operations that a function will consist of). However, predefined templates are planned to be included as part of the LOCUS function SDK to be able to easily integrate the control points in a remote method invocation framework that acts as data operations coordinator. The elementary control operations that a function includes are: a) health check, b) initialization, c) dispose, d) execute corresponding to internally implemented steps/paths according to each implementation. The LOCUS function SDK from WP2 also plays a key role in this functionality as it targets a reusable ready-to-use library implementation of these functionalities from LOCUS functions developers to reduce code duplication and code review overhead. In practice, the analytics function control plane is provided by the LOCUS SDK feature.

Based on these assumptions, the Localization Analytics as a Service approach requires the integration of three core elements for its implementation. The LOCUS MANO deploys NFV Network Services and VNFs that host the analytics services and functions in the 5G edge/core virtualization platform. The Data Operations Controller executes the analytics services pipelines, both assisted by the Analytics Service Coordinator to glue the API Gateway service exposure with the internal platform logics.

A functional description for each of the Pipeline Orchestrator building blocks shown in Figure 52 is provided below.

The Internal Analytics Catalogue is responsible to maintain the internal list of available analytics services and functions for their management by the Pipeline Orchestrator. It is worth to mention that this catalogue has to be considered as complementary with the LOCUS NFV MANO NSD and VND catalogues (see Figure 52) since, as described in D4.3, they already store information available for the localization analytics functions and services, but mostly related to their requirements for automated deployment, configuration and operation in the edge/core 5G virtualization platform. On the other hand, the Internal Analytics Catalogue does not require to host such information, while it rather focuses on analytics data related requirements for both services and functions. However, the Internal Analytics Catalogue needs to keep track for each analytics service and function which is the correspondent NSD and VNFD in the LOCUS NFV MANO catalogues that actually provide their implementation in the form of virtualized functions and services. For this reason, as shown in Figure 53, to onboard new analytics services and functions in the Internal Analytics Catalogue (and thus make them available and visible to the internal Analytics Service Coordination logics), the correspondent VNFDs and NSDs need to be already onboarded and thus available in the

LOCUS MANO catalogues. A reduced set of the analytics service information stored in the Internal Analytics Catalogue is also onboarded in the API Catalogue within the API Gateway to expose it towards external applications. In particular, in LOCUS, these onboarding operations are considered to be performed by the platform manager/administrator in the specific order of Figure 53, properly linking identifiers of NSDs and VNFDs in the NFV MANO catalogues with the analytics services and functions stored in the Internal Analytics Catalogue and the API Catalogue.

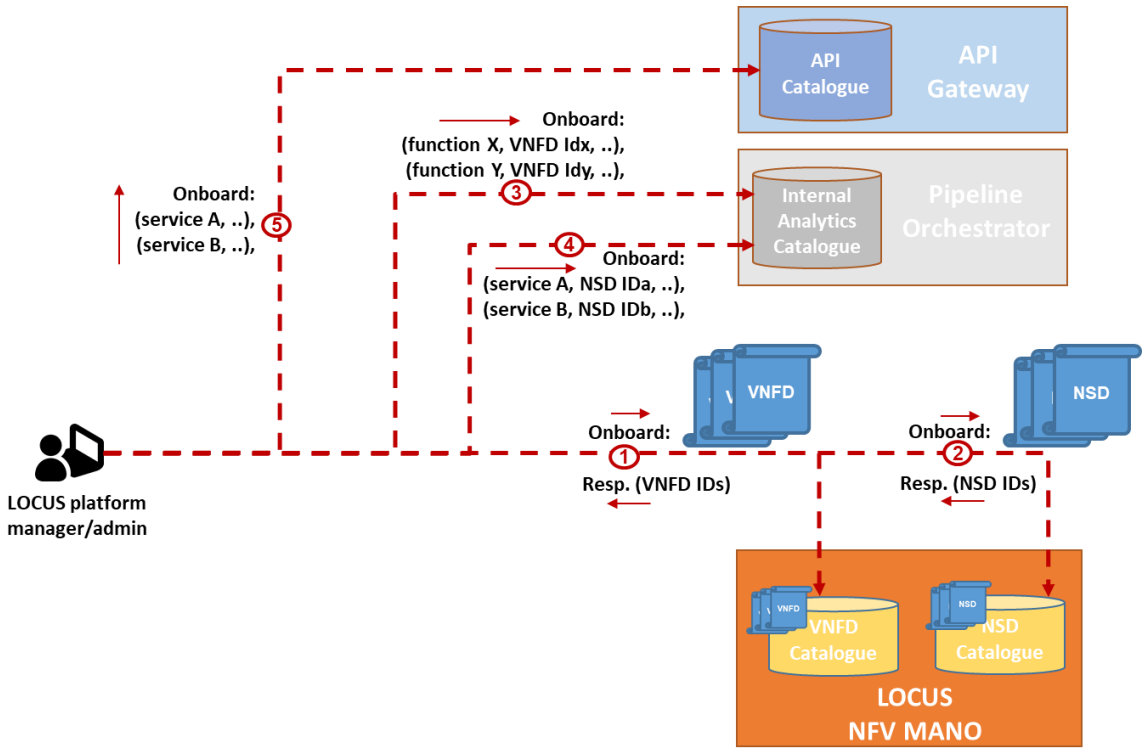


Figure 53: Internal Analytics Catalogue and API Catalogue onboarding approach

As said, the Internal Analytics Catalogue maintains for each LOCUS function and service available in the platform specific analytics data related information that is not suited for the LOCUS NFV MANO and that shall not be exposed to the API Gateway, but is required to express their internal data operations and data exchange requirements. This information is included as part of the analytics services and functions onboarding operations depicted in Figure 52 (steps 3 and 4) and is stored in the Internal Analytics Catalogue to be used and processed by the Analytics Service Coordinator and the Data Operations Controller.

Table 13 and Table 14 below provides the information stored in the Internal Analytics Catalogue for analytics functions and services. In particular, the list of attributes describing and modelling a given function and service internally for the LOCUS platform is reported.

Table 13: Internal Analytics Function information model

Analytics function attribute	Description
ID	Unique identifier of the function.
name	The name of the analytics function
type	It specifies the type of the analytics function implemented.
description	Textual description of the analytics function
input data schema	It specifies the data format expected by the analytics function at its input, e.g. in the form of a JSON schema
output data schema	It specifies the data format produced by the analytics function as output, e.g. in the form of a JSON schema
requirements	A list of analytics function requirements, that need to be fulfilled for the proper execution of the function. These may include specific input data source required (e.g. provided by another function), edge or user proximity constraints, etc.
VNFD ID	The reference VNFD in the LOCUS MANO catalogue

Table 14: Internal Analytics Service information model

Analytics service attribute	Description
ID	Unique identifier of the service. It has to be same as the one used in the API Catalogue
name	The name of the analytics service (e.g. the name of the related NSE UC). Same as in the API Catalogue
type	It specifies the type of the analytics service provided. It is mapped with the NSE UCs described in section 2. Same as in the API Catalogue
description	Textual description of the analytics service. Same as in the API Catalogue.
pipeline requirements	A list of analytics service requirements, that need to be fulfilled for the proper execution the analytics service pipeline, e.g.:

	• the execution plan in the form of a direct acyclic graph • specific input data sources required (e.g. linked to other analytics services or functions) • edge or user proximity constraints
analytics functions IDs	List of analytics functions identifier that compose the analytics service
NSD ID	The reference NSD in the LOCUS MANO catalogue, that needs to be used by the API layer to request for the instantiation of the service in the virtualization platform

The Analytics Service Coordinator is the core engine of the Pipeline Orchestrator and takes care to coordinate the various operations related to the deployment and execution of analytics service pipelines. It supports the two modes described above: the pre-scheduled deployment and execution of analytics service pipelines (thus not explicitly required by an external application), and the on-demand subscription and activation of analytics services triggered by external applications. In practice it works as a workflow engine, responsible to coordinate the deployment of the analytics functions VNFs through the LOCUS MANO, and the analytics service pipeline data control and execution plan through the Data Operations Controller.

The Analytics Service Coordinator relies on the information stored in the Analytics Catalogue to implement its coordination logic. When a new analytics service has to deployed and the related pipeline executed (either as part of an on-demand analytics service subscription, or of a pre-scheduled internal LOUCS platform deployment) the Analytics Service Coordinator process the related service information in the Internal Analytics Catalogue to first identify the correspondent NSD and thus request to the LOCUS MANO for its instantiation. Existing analytics services (i.e. already deployed and running in the 5G edge/core virtualized infrastructure) may be re-used when applicable. When the analytics service is deployed (thus the related VNFs are running in the 5G edge/core virtualized infrastructure), the Analytics Service Coordinator can leverage on the LOCUS MANO APIs for Network Service and VNF Day2 configuration to satisfy some of the analytics service operation requirements. In practice, this translates into issuing Network Service and VNF configurations through the LOCUS MANO to enable the proper data exchange among the involved VNFs (e.g. configuring Kafka endpoints and topics to use), to implement the data movement approach described in section 0. When the service is properly running in the virtualization platform, the analytics service pipeline can be executed through the Data Operations Controller.

In addition, when the analytics service is ready to be consumed by external applications, the Analytics Service Coordinator prepares the dedicated Analytics Data API (in the form of a REST endpoint of Kafka message bus topic) and updates the API Catalogue in the API Gateway with the related API endpoint information, setting the status of the external service to ready. If the deployment and execution of the analytics service was triggered by an on-demand subscription operation, the Analytics Service Coordinator also takes care to send the asynchronous notification message to the related external applications to notify on the status of the service and provide the Analytics Data API details to consume the output data.

The Data Operations Controller implements the analytics service pipeline data control plane functionalities and allows to execute each pipeline following a plan in invoking the various functions according to the data communication and data exchange requirements. Several opensource frameworks are available to implement the Data Operations Controller functionalities. LOCUS plans to leverage on Apache Airflow[26], that is a flexible solution for customized pipeline coordination and control. It is based on reusable principles such as:

- Operators, as extensible elementary operations with inputs and outputs
- Direct Acyclic Graphs (DAGs), as declarative language for operation pipelines

Apache Airflow is a tool to define pipelines, manage and schedule various executions, check their health and generally remote operate each functional block. It can be used in multiple integration modes such as: direct code execution, kube-operator (invoking the instantiation of a Kubernetes POD), and indirect invocation of a REST service. In LOCUS, the plan is to adopt the REST API integration used in conjunction with the control REST endpoints each function is expected to expose according to the LOCUS SDK. In particular, Apache Airflow Operators are a way to create data functions aiming at reusability. Creating a parametrizable operator can lead to very consistent code bases that implement the best practices in execution performance. Different instantiations and graphs (DAGs) of these operators can lead into different parts of the analytics service schedule. Ultimately all the analytics functions that the LOCUS platform provide can be executed in a continuous, rescheduled manner based on each own required time granularity.

The elementary functions that are used to compose each analytics service pipeline can vary in content and context, from data input manipulation to in memory processing and data generation. However, special treatment can be applied in the cases of ML pipeline steps. Opensource Frameworks like MLFlow[27] have seen increased adaptation in the industry to work as a "glue" framework for these operations. Various example implementations exist that show Airflow and MLFlow integrate via the remote method invocation API of MLFlow from Airflow - mixing the best from both worlds.

4.2 Northbound APIs overview

The API Gateway, as described in the previous section 4.1, exposes a set of northbound APIs which can be considered the entry point of the LOCUS platform for accessing the localization analytics as a service. The main consumers of such APIs are the LOCUS applications, i.e. the Smart Network Management and 3rd party vertical applications that are outside the LOCUS platform, but still part of the LOCUS system. In practice, the Localization Analytics as a Service APIs are mostly implemented as REST interfaces based on the HTTP protocol and JSON messages, including support of basic HTTP and OAuth2 authentication. In addition, identity management is supported over the APIs, which combined with the authentication mechanisms allows to implement isolated API contexts for each external application interacting with the LOCUS platform.

In summary, the localization analytics APIs offered can be grouped into three main categories:

- *Service Discovery REST APIs*: used to query information (as per Table 12) related to analytics services available in the LOCUS platform. These services can be already available to be directly consumed through the related Analytics Data APIs,
- *Service Subscription REST APIs*: used to subscribe to localization analytics services and trigger first their deployment and instantiation in the 5G edge/core virtualized infrastructure through the Analytics Service Coordinator and the LOCUS MANO, and after their pipeline execution through the Data Operations Controller. As a result, the information related to the Analytics Data APIs to be used to consume the analytics service output is asynchronously notified to the external application
- *Analytics Data APIs*: used to consume the output data of those analytics services deployed and in execution in the LOCUS platform. According to the specific capability and characteristic of the given analytics service, these APIs are either:
    - REST API to retrieve the analytics service output data through explicit GET requests
    - Kafka message bus topic to subscribe to for consuming the analytics service data as a data stream

The definition and specification for these LOCUS northbound APIs (including the additional identity management operations to implement isolated API contexts and views for each API consumer) will be provided with the final LOCUS architecture deliverable D2.5.

4.3 Intent-Based API

4.3.1 Intent as a concept

An intent is an expression of a specific state and/or a description of a particular output, that a user expects the system to deliver. It could also be described as “an abstract, high-level policy used to operate a network”[28]. The underlying motivation behind introducing the concept of Intent in the LOCUS Location & Analytics as a Service, is to create an abstraction from the underlying system capabilities, and to interact with the network as one logical system. The goal is to hide the inherent complexity of a system composed of a plethora of components, and processes massive amounts of data, all while privileging a user experience and interaction focused on expression of goals/Intents. The challenge is, then, to make it possible to interact with the network as a single unit, while retaining the freedom to optimize the orchestration of network capabilities (services and hardware).

The introduction of intent-based interactions with networks and systems, traces back to discussions around autonomic systems [28], network management [29] and software-defined networks [30], so as to allow networks to adapt autonomously to business goals, that are expressed without specifying detailed technical information on network operation. The Open Networking Foundation produced a “Intent NBI – Definition and Principles” document [31], that maps the concept of Intents to the Northbound Interface (NBI), and views it as a means for service consumer-producer interaction where consumer/producer implementations are decoupled, and Intent-based interactions are generalized beyond User-System interaction to include internal system closed loops.

ONAP is also interested in Intent Based Management as demonstrated by a use-case currently under development [32], with the goal of achieving an Intent Based Network that automates the identification of users’ network requirements based on natural language expressions. The concept is under study within 3GPP as well, and in particular in the technical report on “Telecommunication management; Study on scenarios for Intent driven management services for mobile networks” [20] and under technical specification in “Management and orchestration; Intent driven management services for mobile networks” [34].

An Intent-Based API for LOCUS Analytics targets real-time data pipelines, that rely on intents to identify a problem, then compose and instantiate services that process data to solve it.

In Figure 54 a typical Intent lifecycle is depicted. It involves an intent layer, an analytics layer, and a network layer:

- At the intent layer we find the core of the intent-based API, where user input is parsed and translated. Also where the analytics results are validated.

- At the analytics layer, the system creates a particular configuration to executed, and runs required data processing and analytics after execution.
- At the network layer, the system is in execution and various KPIs and telemetry data is generated.

4.3.1.1 Intent Management

This step involves parsing user input, to identify the intents within and whatever could be considered as relevant details. This could involve, for example, the reliance on Natural Language Processing (NLP) technologies to parse text-based input, or speech-to-text to process voice-based input. Understandably, this process could generate unwanted complexity if the range of possible vocabulary of input is not restricted. Basically, this step needs to be optimized to account for both the scope of system's capabilities, domain of provided services, and the nature of provided interaction (descriptive, predictive, diagnostic, etc). In addition to that the proper training of the parsing models within the same constraints.

4.3.1.2 Intent Translation

This step consists of creating a mapping between identified intents, and concrete system capabilities (resources, data, services, etc), in addition to generating a composition of these capabilities with the goal of answering the user's intent. This translates to deciding what pipeline to instantiate from predefined pipelines within the system, or if needed what functions/services to use to compose a new pipeline that would be configured and instantiated. Also, what data sources to pull and the optimizations needed for efficient data movement, and analytics execution within the identified intent requirements.

4.3.1.3 Intent Validation

At the validation step, the system is expected to perform a closed-loop optimization, that involves analysing system output and performance with regards to user's intents and constraints, then optimizing the intent translation engine accordingly. The validation strategy is important and should adapt to changing system capabilities, to guarantee continuous intent satisfaction, it always provides the means to monitor system performance in the context of user intents.

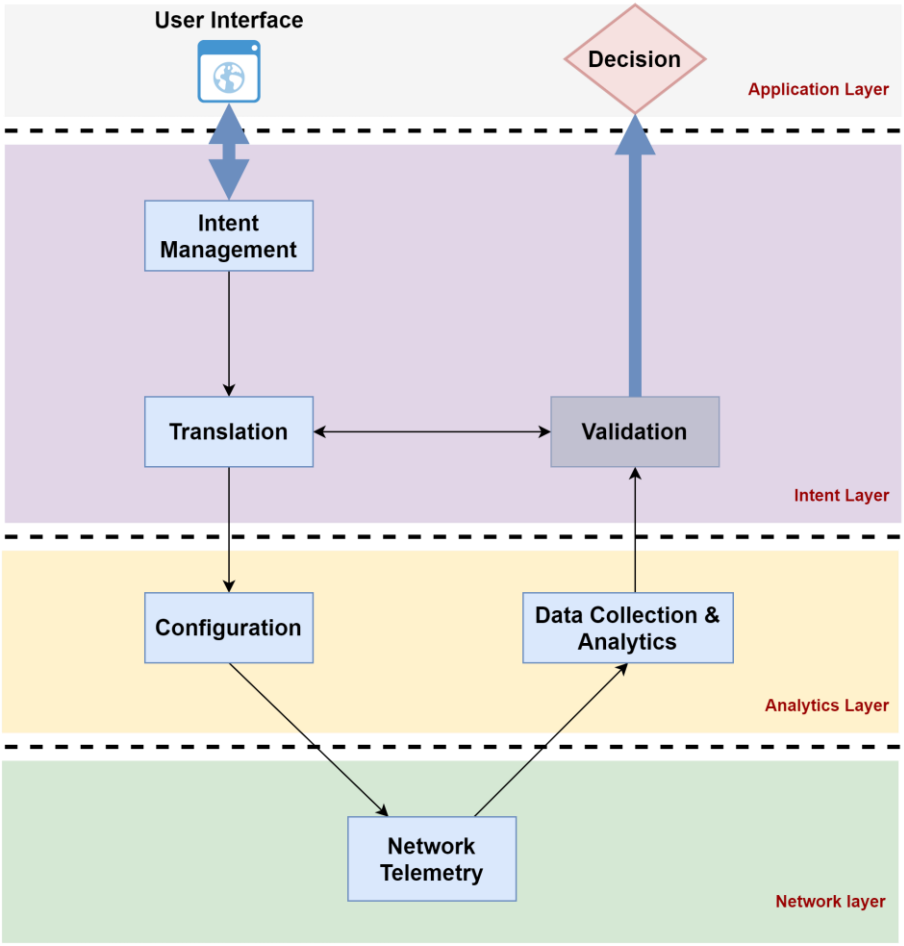


Figure 54: Intents lifecycle

4.3.2 Intent driven interactions with LOCUS Analytics Functions

In order to identify what could be a possible mapping between the use cases and functionalities studied in WP5 of LOCUS, we created the template in Table 15.

Table 15: IBA template for use cases

Use Case Title	*NSE-US**X**-Func**Y***
Description	Brief description
System Components	*LOCUS Components that are actors of this UC.*
Data Scope	Types of Data processed by this UC

~Real-Time	Expected temporal scope. Non-Real-time, Real-time, etc.
Keywords	Concepts used by the intent management to process user input.
Validation Strategy	What metrics/KPIs/Procedures will be used, if any, to validate intent translation, and generated analytics.
Identified Functions	LOCUS functions, if any, that the UC is expected to instantiate or interact with. (Ref D2.4)
Example	Optional.
Assumptions	*Describes any assumptions that are made for this use case*

Table 16: NSE-UC1-Func1 Requirements for IBA

Use Case	*NSE-UC1-Func1*
Description	Identification of patterns of individual or collective mobility behaviour, in terms of clusters, trends, densities, etc. in indoor/ outdoor/ hybrid locations.
System Components	*Data Movement, Data Persistence, Data Anonymization, LMF, NFVO*
Data Scope	Location Information, Mapping Information.
~Real-Time	No
Keywords	Points-of-interest, Routes, Flows, Density, Count, Trend, Cluster, Time.
Validation Strategy	Output confidence.
Identified Functions	POI Identification, Trajectoires Identification, Routes Identification, Flows Identification, ML Functions

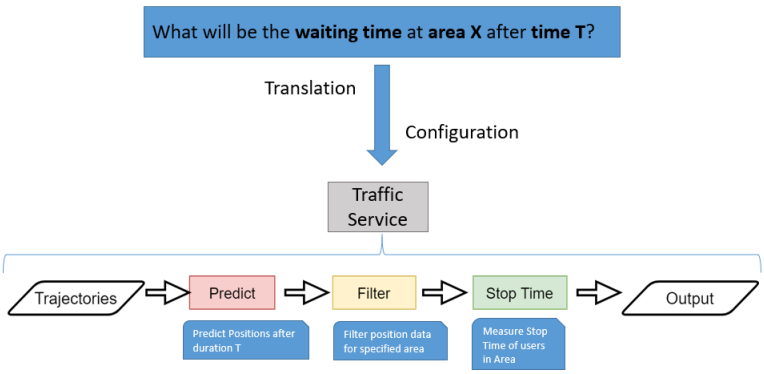
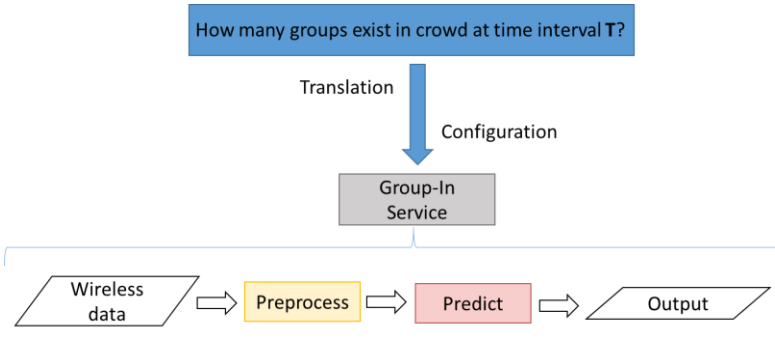
Example	
Assumptions	

Table 17: NSE-UC2-Func1 Requirements for IBA

Use Case	*NSE-UC2-Func1*
Description	Provides analytics for monitoring crowd mobility and inferring group structures. A multi-phased analytics based on noisy wireless Received Signal Strength Indicator observed by multiple wireless scanners.
System Components	*Data Movement, Data Persistence, LMF, NFVO*
Data Scope	Wireless data (e.g., Bluetooth advertisement, Wi-Fi)
~Real-Time	Non-real-time
Keywords	Group count, group size, crowd size, time interval, sampling, wireless fingerprints
Validation Strategy	Jaccard and pairwise accuracy of output groups.
Identified Functions	ML Functions, Trajectories Identification, Flows Identification
Example	

Assumptions	*Existence of wireless scanners in the environment.* *Usage of system without additional parameter configuration, for environments where ground-truth cannot be collected.*

Table 18: NSE-UC3 Requirements for IBA

Use Case Title	***NSE-UC3-Func2: Time to collision as a service in V2X***
Description	Identify collision risk profile of specific V2X user(s) or a road network section/location during a predefined time period
System Components	LMF, Localization Enablers, Localization & Analytics for New Services, Persistence Entity, LOCUS APIs
Data Scope	Location Information, Mapping Information, Time to collision
~Real-Time	No
Keywords	Trajectory, Time to collision, Routes, Time, Location, Collision likelihood
Validation Strategy	Semantic and consistency check.
Identified Functions	Trajectories Identification, Routes identification, Flows identification, Time to Collision, Profiles Identification, Categorical Aggregation
Assumptions	

Table 19: NSE-UC4 Requirements for IBA

Use Case Title	***NSE-UC4 Logistics in a seaport terminal using AGVs***
Description	Position and velocity information provided by the 5G network is used to determine the next moves of Automated Guided Vehicles (AGV) on the planned path, i.e. handles the planning of the shuttling tasks, determines the trajectory of the AGV, computes and transfers the next motor command to the AGV.
System Components	AGV motion controller, Task planner, Trajectory planner, Map Manager
Data Scope	AGV mission task (input), AGV position information (input), Area map (internal), AGV motion control (output)
~Real-Time	Realtime

Keywords	Task planning, trajectory planning, navigation, motion control, map management
Validation Strategy	The UC will be simulated in a VR environment in PoC#2. Map of the logistic area and AGV mission examples will be taken from a real sea-port environment experience
Identified Functions	• AGV motion controller function: this function controls the navigation and movements of the vehicle in the area. It receives information from the trajectory planning function and from the positioning collection function and, based on this information, it computes the next movement step • AGV task planning function: this function coordinates the activities in the seaport warehouse area and takes care of assigning the vehicles to the different missions depending on their status and their current position respect to the freight to shuttle. This function handles also the relational database containing the freights inventory and the vehicle data. It will make use of a rule-based expert system and a relational database based on mySQL. • AGV trajectory planning function: this function is used to determine the path the vehicle must follow in the test area first to reach the freight to pick and then the final destination. The algorithm will receive a map (see next function) reporting the areas where the vehicle can and can't go through. These areas include also the other freights placed in the seaport area whose position is predetermined. • Map manager: the map is built on the fly based on the information about static object in the area, and position and size of freights contained in the inventory in the relational database.
Assumptions	

Table 20: NSE-UC5 Requirements for IBA

Use Case Title	***NSE-UC5 Traffic Profile Monitoring***
Description	This use case requires location information at a large scale in an out-door area. Given this outdoor setting, where various high or low traf-fic streets, venues, and motorways as well as train routes and pedes-trians exist, a variety of different mobility profiles emerge. The pur-pose of this function is to a) analyse the localization data found in

	the locus data platform schema for the underlying area to identify consistent paths and b) analyse the mobility profile of the underlying identified paths in order to establish a baseline for the mobility pattern of the area. Then c) by monitoring changes in the instantaneous mobility profile, it can generate alerts to local authorities for transportation accidents, traffic jam events or other incidents.
System Components	• LOCUS localization enabler for an outdoor area • LOCUS Data Platform • LOCUS Analytics/ML Pipeline infrastructure • LOCUS Analytics Service Orchestrator • LOCUS Service Gateway • 3rd Party Service Receiver: This could be an App/Dashboard.
Data Scope	Geolocation Data (input), Path Data (internal), Velocity Profile (output), Velocity profile events (Output)
~Real-Time	Near - realtime
Keywords	Path identification, Mobility patterns, Pattern monitoring,
Validation Strategy	The validation of this service can be split into two phases: a) cross validation of identified user movement paths based on actual geo-json shapes provided by a 3rd party map provider (e.g. Open Street Maps) b) based on actual traffic data provided by transportation monitoring analytics (e.g. Google Maps Traffic API) we can cross validate the mobility profile change.
Identified Functions	• NSE-UC5_FR1: The system shall exploit geolocation-velocity data to identify paths in the form of geo-json polygons. • NSE-UC5_FR2: The system shall be able to correlate identified paths with the geolocation-velocity measurements. • NSE-UC5_FR3: The system shall be able to perform mobility profile extraction from aggregated measurements per identified paths. • NSE-UC5_FR4: The system must be able to maintain a state for each of the monitored paths (inside the selected area) and re-compute NSE-UC5-FR3 to monitor for changes.

	• NSE-UC5_FR5: The system shall offer interfacing capabilities for 3rd party Apps for both the identified paths, their respective velocity profiles and the change events that occur
Assumptions	• Data Privacy is ensured through anonymization and abstraction of location information. We assume that the localization accuracy is within the range of the specified accuracy requirements.

5 Conclusions

This deliverable presented a design for the Location & Analytics as a Service Solution of the LOCUS platform. The proposed design consisted of three main elements: (i) the specification of the location-based functionalities that the LOCUS systems aims at supporting, in terms of pipelines of microservices, as well as the virtualization/orchestration of said pipelines in addition to ML optimization for flexible function placement; (ii) the description of the data management foundation that enables scalable and efficient data movement and persistence, and communication between services in support for batch-based and stream-based dataflows; (iii) the specification of the API gateway that provides service discovery/subscription and analytics data consumption for external applications.

The deliverable elaborated on the use cases previously described in D5.1, providing a mapping of the proposed technical solutions relying on ML techniques, as well as a pipeline view that achieves target output by chaining a set of microservices. From such analysis, a catalogue of foundational services is studied, which rely on the development of functionalities to compose a wide range of pipelines that will be onboarded to a runtime catalogue of services, for their instantiation, orchestration, and monitoring.

Th pipelines of various functionalities are deployed in a virtualized environment, where lifecycle management and orchestration are achieved by relying on the LOCUS MANO and NFV principles, in agreement with what was reported in D4.3, for the deployment and operation of VNFs and NFV Network Services that implement WP5 use case functionalities. The LOCUS MANO takes care of function packaging in virtualization-ready images, automate edge/core deployment through MANO descriptors, and it also manages the instantiation of network resources, Kafka clusters and pipeline composition.

The deliverable included preliminary prototypes of functional decomposition for mobility clustering functionalities, virtualization, life-cycle management and serving of results. A proposal for combining ML life-cycle management frameworks inside a MANO managed infrastructure was provided. In addition, the use of an API gateway was introduced, which decouples the exposed analytics services from the internal analytics service data control and pipeline orchestration and supports RESTful interface for consuming batch-based and stream-based analytics output. On top of this, an indicative Intent-Based API layer concept was defined and mapped to some use cases investigated in WP5.

In summary, this deliverable detailed the LOCUS Architecture in its support for vertical applications, in preparation of the services specifications, data models and prototyping of the architecture that will be presented in the upcoming D2.5 and D5.4 deliverables.

6 Bibliography

[1] G. Solmaz, J. Fürst, S. Aytaç, and F.-J. Wu, "Group-In: Group Inference from Wireless Traces of Mobile Devices," presented at the 2020 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Apr. 2020. doi: 10/gjxwd9.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968, doi: 10/c9zgc4.

[3] A. Sayar, S. Eken, and O. Öztürk, "Kd-tree and quad-tree decompositions for declustering of 2D range queries over uncertain space," *Frontiers Inf Technol Electronic Eng*, vol. 16, no. 2, pp. 98–108, Feb. 2015, doi: 10/gjtmr8.

[4] W. Wei, "Analysis of spatial database index technology," in *2010 2nd International Conference on Computer Engineering and Technology*, Apr. 2010, vol. 4, pp. V4-29-V4-32. doi: 10/d95v68.

[5] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, doi: 10/fnpv3t.

[6] L. Ni, F. Tian, Q. Ni, Y. Yan, and J. Zhang, "An anonymous entropy-based location privacy protection scheme in mobile social networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2019, no. 1, p. 93, Apr. 2019, doi: 10/gjtmsv.

[7] "SeldonIO," *GitHub*. https://github.com/SeldonIO (accessed Apr. 16, 2021).

[8] "Kubeflow," *Kubeflow*. https://www.kubeflow.org/docs/ (accessed Apr. 16, 2021).

[9] "Kubernetes Pods," *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/ (accessed May 03, 2021).

[10] "Google Colaboratory." https://colab.research.google.com/notebooks/intro.ipynb (accessed Apr. 28, 2021).

[11] "Minikube," *minikube*. https://minikube.sigs.k8s.io/docs/ (accessed Apr. 28, 2021).

[12] M. Inc, "MinIO | High Performance, Kubernetes Native Object Storage," *MinIO*. https://min.io (accessed Apr. 26, 2021).

[13] "Istio - Connect, secure, control, and observe services," *Istio*. /latest/ (accessed Apr. 26, 2021).

[14] "MicroK8s - Zero-ops Kubernetes for developers, edge and IoT | MicroK8s," *microk8s.io*. http://microk8s.io (accessed Apr. 26, 2021).

[15] "HDFS Architecture Guide." https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction (accessed May 07, 2021).

[16] "How LinkedIn customizes Apache Kafka for 7 trillion messages per day." https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages (accessed Apr. 28, 2021).

[17] Y. Fu, "Disaster Recovery for Multi-Region Kafka at Uber," *Uber Engineering Blog*, Dec. 21, 2020. https://eng.uber.com/kafka/ (accessed Apr. 28, 2021).

[18] “Kafka Producer — Confluent Documentation.” https://docs.confluent.io/platform/current/clients/producer.html (accessed Apr. 16, 2021).

[19] “Kafka Consumer — Confluent Documentation.” https://docs.confluent.io/platform/current/clients/consumer.html (accessed Apr. 16, 2021).

[20] “Kafka Streams API,” *Apache Kafka*. https://kafka.apache.org/0102/documentation/streams/ (accessed Apr. 29, 2021).

[21] “Confluent REST APIs — Confluent Documentation.” https://docs.confluent.io/platform/current/kafka-rest/index.html (accessed Apr. 16, 2021).

[22] “Kafka DSL API,” *Apache Kafka*. https://kafka.apache.org/11/documentation/streams/developer-guide/dsl-api.html (accessed Apr. 28, 2021).

[23] “Kafka Connect — Confluent Documentation.” https://docs.confluent.io/platform/current/connect/index.html (accessed Apr. 16, 2021).

[24] “Migrate from MirrorMaker to Replicator — Confluent Documentation.” https://docs.confluent.io/platform/current/multi-dc-deployments/replicator/migrate-replicator.html (accessed Apr. 16, 2021).

[25] D. Hardt <dick.hardt@gmail.com>, “The OAuth 2.0 Authorization Framework.” https://tools.ietf.org/html/rfc6749 (accessed Apr. 28, 2021).

[26] “Apache Airflow,” *Apache Airflow*. / (accessed Apr. 28, 2021).

[27] “MLflow - A platform for the machine learning lifecycle,” *MLflow*. https://mlflow.org/ (accessed Apr. 28, 2021).

[28] M. Behringer *et al.*, “Autonomic Networking: Definitions and Design Goals.” https://tools.ietf.org/html/rfc7575 (accessed Apr. 16, 2021).

[29] A. Clemm, “Intent-Based Networking - Concepts and Overview,” Jul. 08, 2019. https://tools.ietf.org/id/draft-clemm-nmrg-dist-intent-02.html#RFC7575 (accessed Apr. 16, 2021).

[30] M. Pham and D. B. Hoang, “SDN applications - The intent-based Northbound Interface realisation for extended applications,” in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, Jun. 2016, pp. 372–377. doi: 10/ghdjxv.

[31] ONF, “Intent NBI - Definition and Principles. ONF TR-523.” https://opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf (accessed Apr. 16, 2021).

[32] “Intent-Based Network - Developer Wiki - Confluence.” https://wiki.onap.org/display/DW/Intent-Based+Network#IntentBasedNetwork-UseCaseKeyData (accessed Apr. 16, 2021).

[33] “3GPP TR 28.812.” https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3553 (accessed Apr. 16, 2021).

[34] “3GPP TS 28.312.” https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3554 (accessed Apr. 16, 2021).