ENVELOPE

Evaluation and validation of connected mobility in real open systems beyond 5GS

Project deliverable D3.3

ENVELOPE Innovation Developments Report

HORIZON JU Innovation Actions | 101139048 |
ENVELOPE - HORIZON-JU-SNS-2023

Co-funded by the European Union

6GSNS

Deliverable administrative information

Dissemination level	PU - Public
Type of deliverable	**R — Document, report**
Work package	**WP3**
Title of the deliverable	**D3.3 – ENVELOPE innovation developments report**
Status – version, date	**Draft – V1.0, 31/12/2025**
Deliverable leader	**LUH**
Contractual date of delivery	**31/12/2025**
Submission date	**31/12/2025**
Keywords	**API implementation, experimentation as a service**

List of contributors

Name	Organization
Amr Rizk	**LUH**
Jannis Weil	**LUH**
Nehal Baganal	**LUH**
Anam Tahir	**LUH**
Dimitris Uzunidis	**NCSRD**
Vasilis Pitsilis	**NCSRD**
Harilaos Koumaras	**NCSRD**
Fofy Setaki	**OTE**
George Lyberopoulos	**OTE**
Zouzias Dimitris	**eBOS**

Pavlos Basaras	**ICCS**
Giorgos Drainakis	**ICCS**
Efstratios Myritzis	**ICCS**
Nikolaos Petropouleas	**LNVO**
Edoardo Bonetto	**LINKS**
Michele Petruzza	**LINKS**
Fabrizio Gatti	**TIM**
Gergely Kovacs	**CMS**
Ramon de Souza Schwartz	**TNO**
Alexandru Forrai	**SISW**
Marco Bazzani	**TEO**
Valeria Proietti	**TEO**
Oumayma Sghairi	**VICOM**
Apostolos Siokis	**IQU**
Christos Verikoukis	**ATH/ISI**
Davide Montagno Bozzone	**HPE**
Nicola di Pietro	**HPE**

Quality control

	Name	Organization	Date
Peer review 1	**Dimitris Zouzias**	**eBOS**	**09/12/2025**
Peer review 2	**Apostolos Siokis**	**IQU**	**08/12/2025**

Version History

Version	Date	Author	Summary of changes
0.1	29/10/2025	LUH	Initial draft with ToC
0.2	10/11/2025	All	Updated ToC
0.3	27/11/2025	All	Input from all partners
0.4	01/12/2025	LUH, All	Consolidated version
0.5	09/12/2025	All	Minor revision
0.6	09/12/2025	LUH, eBOS, IQU	Integrated internal review feedback
0.8	18/12/2025	All	Minor revision, added inter-PLMN
0.9	28/12/2025	ICCS	Full deliverable review and quality check
1.0	31/12/2025	ICCS	Deliverable submitted

Legal Disclaimer

Table of contents

List of figures

List of tables

Project executive summary

ENVELOPE aims to advance and open the reference 5G advanced architecture and transform it into a vertical-oriented one. It proposes a novel open and easy-to-use 5G-advanced architecture to enable a tighter integration of the network and the service information domains by

- exposing network capabilities to verticals,
- providing vertical information to the network; and
- enabling verticals to dynamically request and modify key network aspects,

all performed in an open, transparent, and easy-to-use, semi-automated way.

ENVELOPE will build APIs that act as an intermediate abstraction layer that translates the complicated 5GS interfaces and services into easy to consume services accessible by the vertical domain. The project will deliver an experimentation framework that will facilitate vertical services in accessing a series of innovations developed in the project, namely: edge computing with service continuity support (federation/migration), zero-touch management, multi-connectivity, dynamic slicing and predictive QoS.

ENVELOPE will deliver 3 large scale Beyond 5G (B5G) trial sites in Italy, Netherlands and Greece supporting novel vertical services, with advanced exposure capabilities and new functionalities tailored to the services' needs. Although focused on the Connected and Automation Mobility (CAM) vertical, the developments resulting from the use cases (UC) will be reusable by any vertical. The ENVELOPE architecture will serve as an envelope that can cover, accommodate, and support any type of vertical services. The applicability of ENVELOPE will be demonstrated and validated via the project CAM UCs and via several 3rd parties that will have the opportunity to conduct funded research and test their innovative solutions over ENVELOPE.

Social Media link:

@envelope-project

For further information please visit www.envelope-project.eu

Deliverable executive summary

Work Package 3 (WP3) is responsible for developing the activities related to the implementation of the ENVELOPE platform and its associated innovations. The ENVELOPE platform is a B5G System Experimentation as a Service (EaaS) platform that provides simplified and extended interactions with the 5G network.

This Deliverable D3.3 documents the results regarding the Tasks T3.3, T3.4, and T3.5, which build upon the work presented in the previous Deliverable D3.2 and comprise

- The development of CAM services and interfaces to enable a seamless interaction of these services with the B5GS trial sites (T3.3)
- The development of advanced Multi-access Edge Computing (MEC) functionalities to enable cross-domain coordination and improve efficiency (T3.4)
- The development of an AI framework that processes declarative intents for a zero-touch management of CAM services in large trial sites (T3.5).

This deliverable concludes the work of WP3 and the main development of the ENVELOPE platform. The integration into the trial sites and further testing will be continued as part of T4.2-T4.4 of WP4, leading to the large-scale trial site evaluation and demonstration in WP6.

List of abbreviations and acronyms

Acronym	Meaning
AF	Application Function
AMF	Access and Mobility Management Function
AnLF	Analytics Logical Function
ANE	ATSSS Layer - Network Side
ANN	Artificial Neural Network
APF	ATSSS Policy Function
ATSSS	Access Traffic Steering, Switching and Splitting
AUE	ATSSS Layer - UE Side
AV	Autonomous Vehicle
CAM	Connected and Automated Mobility
CAPIF	Common API Framework
COLMAP	Computer Vision Library for Multi-view Photogrammetry
CL	Closed Loop
DENM	Decentralized Environmental Notification Message
DoS	Denial-of-Service
DT	Digital Twin
DTD	Digital Twin Data
EaaS	Experimentation as a Service
EC	European Commission
ETSI	European Telecommunications Standards Institute
FGSM	Fast Gradient Sign Method
FTPS	File Transfer Protocol Secure
HLO	High-Level Orchestration
HO	Hand-over
IBN	Intent-Based Networking
ICP	Iterative Closest Point

IE	Infrastructure Elements
LBO	Local Break-Out
LLO	Low-Level Orchestration
LSTM	Long Short-Term Memory
MEC	Multi-access Edge Computing
MEO	MEC Orchestrator
ML	Machine Learning
MLP	Multi-Layer Perceptron
MtLF	Model Training Lifecycle Function
MTS	Multi-access Traffic Steering
MVS	Multi-View Stereo
NeRF	Neural Radiance Fields
NEF	Network Exposure Function
NWDAF	Network Data Analytics Function
OBU	On-Board Unit
OTA	Over-the-Air
PCI	Physical Cell Identity
PDU	Protocol Data Unit
PLMN	Public Land Mobile Network
PMF	Performance Measurement Function
PPO	Proximal Policy Optimization
PQoS	Predictive Quality of Service
QoD	Quality on Demand
QoS	Quality of Service
RAN	Radio Access Network
RL	Reinforcement Learning
RTT	Round-Trip Time
SfM	Structure from Motion
SIFT	Scale-Invariant Feature Transform

SMF	Session Management Function
SSB	Synchronization Signal Block
SSC	Session and Service Continuity
UC	Use Cases
UE	User Equipment
UPF	User Plane Function
URI	Uniform Resource Identifier
V2V	Vehicle-to-Vehicle
WP	Work Package
ZSM	Zero-touch network and Service Management
3GPP	3rd Generation Partnership Project

1 Introduction

1.1 Purpose of the deliverable

This deliverable documents the outcome of the activities carried out within WP3, focusing on the development and implementation of the ENVELOPE platform that aims to enable a simplified access to B5G system functionalities for vertical applications.

To achieve this objective, the activities of this deliverable build upon the Experimentation as a Service (EaaS) platform architecture established in Deliverable D3.1, as well as the network exposure capabilities documented in Deliverable D3.2. Utilizing this foundation, the present deliverable documents the outcome of the development and implementation of service-oriented capabilities within the ENVELOPE platform, including details regarding relevant innovations and system components within the reconfigurable B5G system that facilitate these services.

This deliverable further reports the development progress of Connected and Automated Mobility (CAM) applications at the three trial sites and describes how they leverage the interfaces and software components provided by the ENVELOPE platform to interact with the B5G system.

This deliverable serves as a reference for the ongoing development and integration activities in WP4 and paves the way for the large-scale WP6 experiments that will demonstrate and evaluate the capabilities of the ENVELOPE platform.

1.2 Relation with Deliverable 3.2

As ENVELOPE consortium we performed a logical separation of the APIs between i) those focused on fundamental network exposure capabilities, which are the ones related with QoS, Device Location and Performance Metrics, and ii) those related to advanced service-oriented capabilities including Predictive Quality of Service (PQoS), ATSSS multi-connectivity, MEC handover and edge computing continuum management. The latter are included and analyzed in this deliverable while the former have been analyzed in the previous WP3 deliverable (D3.2).

The main rationale behind this segregation within ENVELOPE is to follow the natural application development patterns where basic network access must be established before implementing exploitation of predictive analytics or cross-domain service migration. The division ensures focused deliverable scope while enabling seamless integration of increasingly sophisticated B5G innovations essential for connected mobility deployments. Each deliverable maintains clear boundaries while building towards ENVELOPE platform capabilities that support autonomous vehicle operations requiring both fundamental network access and advanced orchestration features. To facilitate understanding of this API distribution and provide clear reference for implementation activities, we tabulate in Table 1 the mapping of all ENVELOPE APIs, categorizing them by their nature (Native or CAMARA-compliant/CAMARA-style) and indicating their respective deliverable allocation.

Table 1 – ENVELOPE API Distribution across Deliverables D3.2 and D3.3. The table originates from Deliverable D3.2 and has been updated to reflect the developments of Deliverable D3.3.

API Name	Category	Deliverable
Quality of Service (QoS) / Quality on Demand (QoD) APIs		
AsSessionWithQoS	Native	3.2
Quality on Demand (QoD)	CAMARA	3.2
Device Location APIs		
MonitoringEvent	Native	3.2
MEC Location	Native	3.2
Location Retrieval	CAMARA	3.2
Device Location (Geofencing Subscriptions)	CAMARA	3.2
Devices In Area	CAMARA	3.2
Performance Metrics APIs		
AnalyticsExposure	Native	3.2
Performance Metrics (NWDAF)	CAMARA	3.2
Predicted Quality of Service (PQoS) APIs		
NWDAF Native APIs (AnalyticsSubscription, AnalyticsInfo Request, MLModelProvision)	Native	3.3
PQoS-Inference (PQoS-I)	ENVELOPE	3.3
PQoS-Training (PQoS-T)	ENVELOPE	3.3
Connectivity Insights	CAMARA	3.3
Predictive Connectivity Data	CAMARA	3.3
Access Traffic Steering, Switching, and Splitting (ATSSS) APIs		
ATSSS-T	ENVELOPE	3.3
ATSSS-Cu, ATSSS-Cp	ENVELOPE (internal)	3.3
Edge computing APIs		
ETSI MEC APIs	Native	3.3
aerOS APIs	Native	3.3
Kubernetes / Helm APIs	Native	3.3
Edge Cloud (Edge Application Management) APIs	CAMARA	3.3
ServiceParameter	Native	3.3
Edge Cloud (Traffic Influence) APIs	CAMARA	3.3

1.3 Intended audience

The dissemination level of D3.3 is public (PU), making this deliverable available to consortium members, European Commission (EC) Services and external stakeholders. This document primarily serves as a technical reference for all ENVELOPE beneficiaries, particularly partners involved in WP3 technical innovation activities and WP4 trial site integration efforts.

The document is designed for multiple audiences including project partners implementing ENVELOPE enablers and APIs, trial site operators deploying the B5G system infrastructure, vertical application developers who will consume the exposed APIs, external experimenters participating in ENVELOPE Open Calls, standardization bodies interested in B5G network exposure approaches and the broader research community working on B5G network programmability and vertical industry integration.

Additionally, this deliverable serves as a key reference for understanding how complex 3rd Generation Partnership Project (3GPP) network functions can be abstracted into simplified, standards-compliant interfaces that enable innovation in vertical domains without requiring deep telecommunications expertise.

1.4 Structure of this deliverable

The structure of this deliverable is organized as follows. Section 2 introduces the core ENVELOPE API framework, including a brief summary of previous work within WP3. The following sections document the project results regarding the development of the predicted Quality of Service (Task T3.3, Section 3), multi-connectivity (Task T3.3, Section 4), and edge computing (Task T3.4, Section 5) APIs. Section 6 details the results regarding zero-touch experiment management from Task T3.5. Section 7 elaborates on the inter-PLMN mobility process considered in ENVELOPE, and Section 8 summarizes the results regarding the development of the CAM applications that are enabled by ENVELOPE APIs and technical innovations. Finally, Section 9 summarizes the next steps and Section 10 concludes this deliverable.

2 ENVELOPE Platform

2.1 Overview

The ENVELOPE project advances the integration of B5G networks with vertical industries, particularly focusing on information exposure to CAM applications. This is made possible through the ENVELOPE enablers which are software components that first consume southbound *Native* APIs (Far-Edge APIs, Edge APIs and B5G Network APIs) through the Common API Framework (CAPIF) to interact with the underlying 5G network infrastructure and edge deployments capabilities. Then, these enablers expose northbound, simplified *ENVELOPE* APIs which abstract network complexities and combine network functionalities with edge capabilities to provide easy-to-consume interfaces to vertical services. ENVELOPE's architecture has been established and elaborated in Deliverable D2.2. The revised version from Deliverable D3.1 is shown in Figure 1.

The term "Native" in ENVELOPE is used to denote the technology-specific APIs and interfaces exposed by underlying B5G network functions and edge platforms. Within this deliverable, "Native APIs" refers exclusively to the underlying network exposure APIs that serve as southbound endpoints, e.g., 3GPP Network Exposure Function (NEF) APIs, 3GPP Network Data Analytics Function (NWDAF) APIs, and European Telecommunications Standards Institute (ETSI) Multi-access Edge Computing (MEC) APIs. This terminology is distinct from common usage of "native" in cloud-native or AI-native contexts. ENVELOPE enablers consume Native APIs and abstract their complexity into simplified northbound ENVELOPE APIs for vertical application consumption.

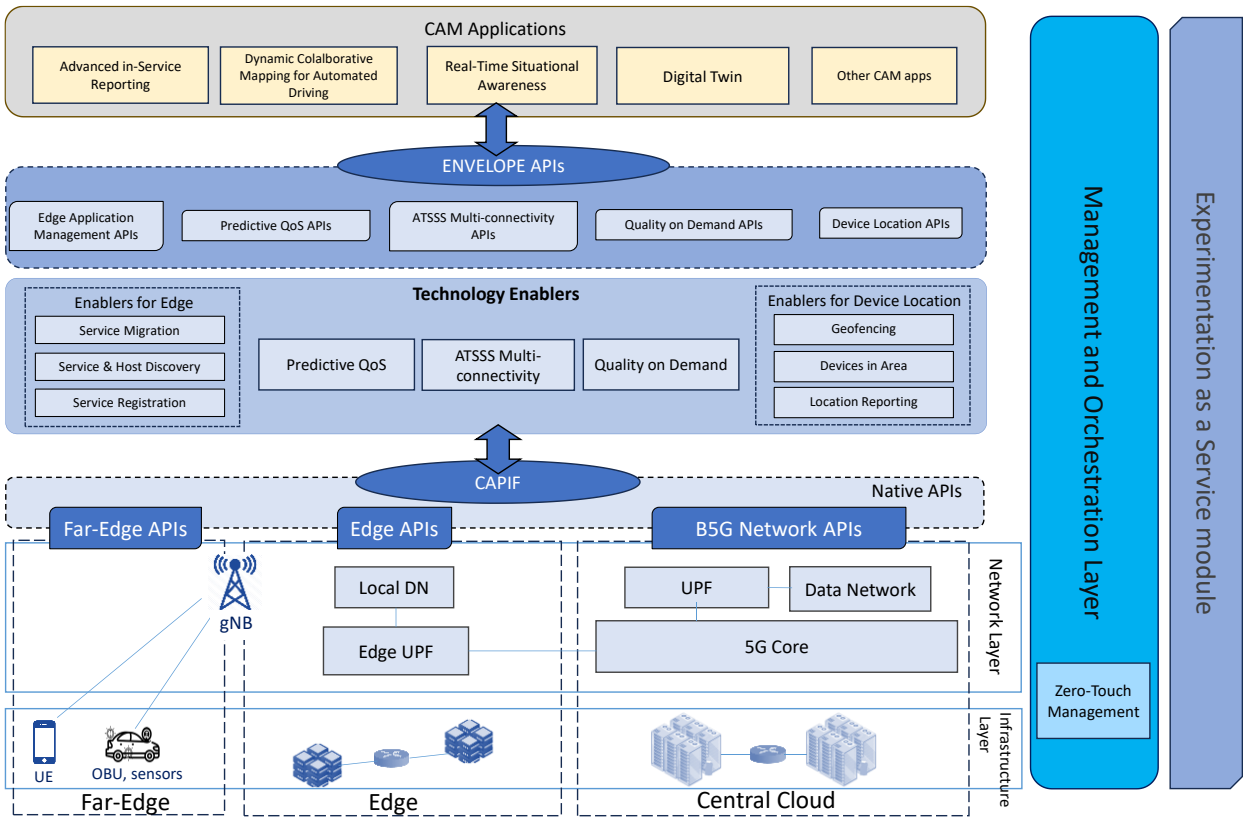

Figure 1 – The ENVELOPE architecture from D3.1.

2.2 Summary of Previous WP3 Deliverables

Deliverable D3.1 established a technical basis for the development of the ENVELOPE EaaS platform. It defined a common denominator to support various ENVELOPE enablers that jointly provide an abstraction layer on top of advanced B5G network functions for improved usability in vertical industry applications.

Based on this foundation, Deliverable D3.2 documented the developments of the Quality on Demand APIs, Device Location APIs, and the Performance Metrics APIs. The information exposed by these APIs can be leveraged by CAM applications and by other ENVELOPE enablers.

The relation of this Deliverable D3.3 with the previous Deliverable D3.2 is detailed in Section 1.2.

2.3 APIs in this Deliverable

The APIs documented in this deliverable focus on advanced service-oriented capabilities.

Table 2 provides an overview and the following paragraphs summarize the functionality of the APIs.

Table 2 – Native and ENVELOPE APIs included in this Deliverable.

API name	Category	Standards Compliance	Link	Section
Predicted Quality of Service (PQoS) APIs				
NWDAF Native APIs (AnalyticsSubscription, AnalyticsInfo Request, MLModelProvision)	Native	3GPP TS 23.288 and 29.520	3GPP TS 23.288 / 3GPP TS 29.520	3.2, 3.3
PQoS-Inference (PQoS-I)	ENVELOPE	Inspired by 3GPP TS 23.288	-	3.4
PQoS-Training (PQoS-T)	ENVELOPE	Inspired by 3GPP TS 23.288	-	3.5
Connectivity Insights	CAMARA	CAMARA Connectivity Insights	GitHub	3.6
Predictive Connectivity Data	CAMARA	CAMARA Predictive Connectivity Data	OpenAPI	3.7
Access Traffic Steering, Switching, and Splitting (ATSSS) APIs				
ATSSS-T	ENVELOPE	Exposes and implements a subset of ETSI MEC 015 with ENVELOPE-specific traffic steering modes	OpenAPI	4.2
ATSSS-Cu, ATSSS-Cp	ENVELOPE (internal)	Internal interfaces to control the multipath schedulers at the UE and network sides.	-	4.3
Edge Computing APIs				
ETSI MEC APIs	Native	ETSI GS MEC 010-2	ETSI spec	5.2.3.1
aerOS APIs	Native	ETSI NGSI-LD (version 1.6.1)	OpenAPI	5.2.3.2
Kubernetes / Helm APIs	Native	Open source Kubernetes and Helm library APIs	K8s specs / Helm specs	5.2.3.3
Edge Cloud (Edge Application Management) APIs	CAMARA	CAMARA Edge Application Management v0.9.3-wip (April 2025)	OpenAPI	5.2
ServiceParameter	Native	3GPP NEF compliant	OpenAPI	5.3
Edge Cloud (Traffic In-fluence) APIs	CAMARA	CAMARA Traffic Influence v0.9.0-rc.1 (April 2025)	OpenAPI	5.3

The Predicted Quality of Service (PQoS) APIs (see Section 3) provide functionality for predictive network analytics (e.g., prediction of end-to-end latency and throughput), including training and inference of prediction models. By abstracting 3GPP configurations and message structures, PQoS substantially simplifies the integration of predictive QoS capabilities into vertical applications.

The Access Traffic Steering, Switching, and Splitting (ATSSS) APIs (see Section 4) enable vertical services and applications to leverage multi-connectivity over 3GPP and non-3GPP networks for enhanced latency, reliability, and throughput. The implementation supports selected scheduling algorithms and traffic splitting using an intelligent Reinforcement Learning (RL) agent.

The Edge Computing APIs (see Section 5) provide functionality to manage the entire lifecycle of edge applications, ranging from the instantiation and optimized placement of edge applications to dynamic routing of UE traffic to the most suitable edge server in terms of the application's requirements. To improve usability, the ENVELOPE enabler abstracts three different native APIs (ETSI MEC, aerOS, K8s) into a single CAMARA-conform API.

In addition to API development, note that this deliverable documents the results on zero-touch experiment management (see Section 6) that will be leveraged to facilitate large-scale experiments at the three trial sites in order to evaluate the APIs developed and implemented within ENVELOPE. Furthermore, the realized mobility process for inter-PLMN handovers is documented in Section 7 and the development state of concrete CAM applications is presented in Section 8.

3 Predicted Quality of Service (PQoS)

3.1 Overview and Innovations

This section presents the overall design and the main components of the ENVELOPE Predictive QoS (PQoS) enabler, together with the API families through which it is exposed to vertical services. It outlines how the system architecture supports predictive network analytics, predictive model training and application-level integration, while also positioning ENVELOPE's PQoS approach with respect to existing standards and prior implementations.

3GPP specifications define the functions and interfaces required for network analytics exposure through the 5G network data analytics function (NWDAF). However, in practice, available implementations remain limited, often experimental and typically bound to specific vendor ecosystems. Most existing solutions focus on delivering static or historical analytics and do not provide fully compliant mechanisms for predictive QoS reporting, automated ML model provisioning, or application-friendly abstractions of the NWDAF API framework. Consequently, vertical services cannot easily integrate predictive network behavior into their operational logic, despite the potential benefits for service optimization and adaptation.

Within ENVELOPE, these gaps are addressed through the development of a **complete Predictive QoS framework that adheres to 3GPP architectural principles**, while adding the capabilities required for operational deployment. The main contributions can be summarized as follows: (a) We implemented a PQoS enabler capable of producing network parameter predictions based on both network-level and service-level metrics, using a combination of real-time monitoring data and historical information, (b) We provide support for automated model lifecycle operations, including the training of new enhanced Artificial Intelligence/Machine Learning (AI/ML) models or the re-training/tuning of existing ones, upon vertical service request and (c) These functionalities are exposed to third-parties via our developed ENVELOPE PQoS APIs to hide the complexity of native NWDAF configuration. These interfaces allow vertical applications to access PQoS services without the need to handle low-level NWDAF message structures, monitoring mappings, or 5G core configuration parameters. Overall, the ENVELOPE PQoS framework delivers a practical, standards-aligned and deployable solution that extends current 3GPP capabilities and significantly simplifies the integration of predictive network intelligence into vertical applications.

3.2 System Architecture

At a high level, the ENVELOPE PQoS architecture consists of three co-operating domains, as depicted in Figure 2.

The first domain includes the radio, core network and edge infrastructure. This encompasses user devices, the gNB, the 5G Core (AMF, SMF, UPF) and the edge-hosted application server(s). A dedicated Prometheus-based monitoring platform continuously collects network metrics and contextual information from all these entities, ranging from radio quality indicators and gNB load to User Equipment (UE) performance metrics and edge resource measurements, and aggregates them in our developed monitoring platform. This data pipeline is the foundation upon which PQoS inference and training processes operate.

The second domain implements the 3GPP-compliant NWDAF functions, responsible for both analytics inference and AI/ML model lifecycle management. Within this component, the Analytics Logical Function (AnLF) executes prediction tasks using a requested AI/ML prediction model, while the Model Training Lifecycle Function (MtLF) handles the creation, retrieval, or update of models. The NWDAF communicates with the monitoring infrastructure through southbound interfaces and exposes standardized APIs, i.e., *AnalyticsSubscription*, *AnalyticsInfo_Request* and *MLModelProvision*, as defined in 3GPP specifications (all data structures, subscription semantics and response formats follow 3GPP TS 23.288 and 29.520 guidelines).

The third architectural block is the PQoS Enabler at the AF layer, which acts as the entry point for vertical services. This component translates the rich and complex NWDAF API operations into simplified ENVELOPE Northbound APIs that are easier for application to consume. It manages authentication, subscription handling, parameter translation and delivery of notifications or inference results to the vertical. Through this abstraction, verticals gain access to predictive QoS functionality without needing to understand low-level NWDAF configuration options.

Together, these three layers provide an end-to-end pipeline (from data collection to analytics execution and application-level integration) enabling proactive QoS awareness in real-time network environments.

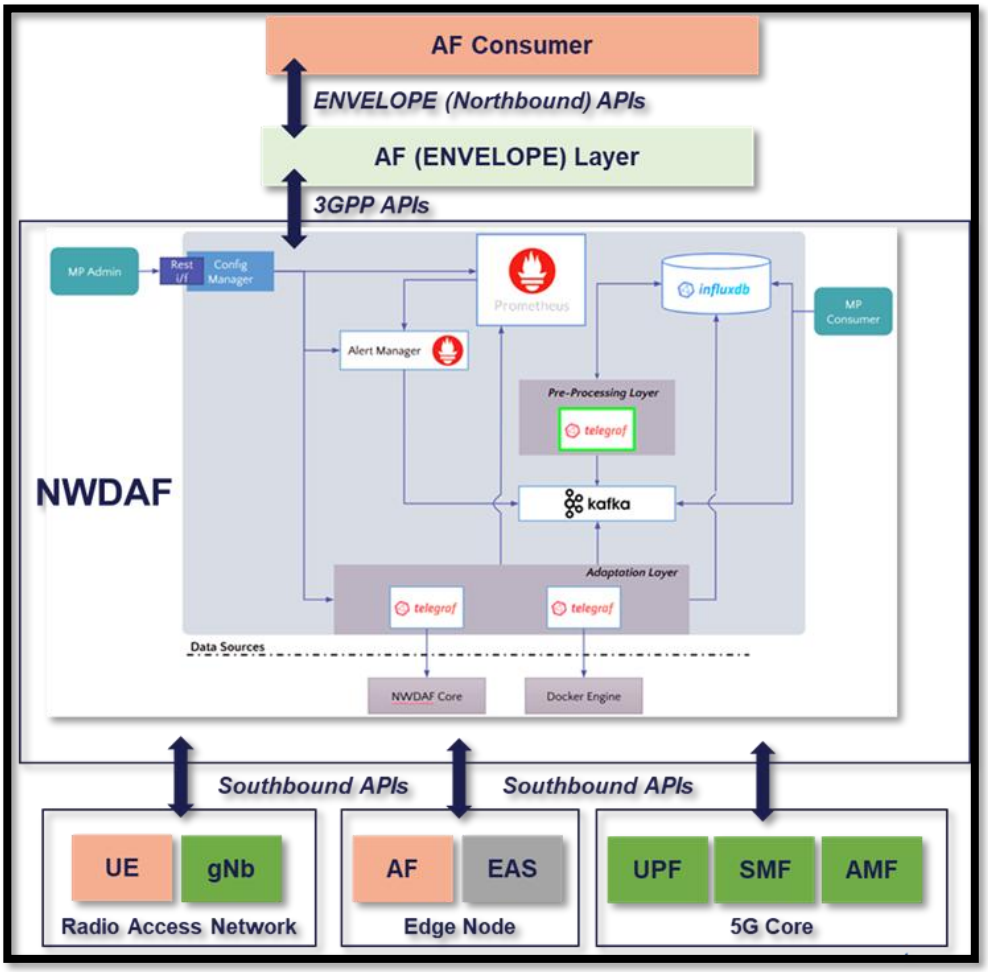

Figure 2 – High-level overview of the ENVELOPE PQoS architecture.

3.3 Functionality and API Overview

The PQoS enabler exposes its functionality through three distinct API families: (a) APIs for predictive inference (PQoS-I), (b) APIs for model training (PQoS-T) and (c) a set of CAMARA-compliant interfaces that offer network quality information following CAMARA's standardised service-exposure model (CAMARA Connectivity Insights and CAMARA Predictive Connectivity Data APIs).

The PQoS Inference (PQoS-I) API family delivers predictions regarding future QoS levels for both network-level (RAN/Core) and service-level (end-to-end) KPIs. Two AI/ML model categories are supported: (a) simple models, which operate on RAN/Core and application-level data only and (b) enhanced models, which additionally incorporate UE-specific information. Within these categories, the API supports several variants, including: (a) QosSustainability (SIMPLE-QS-PQoS) and Data Network Performance (SIMPLE-DNP-PQoS) for basic prediction services and (b) QS-PQoS, DNP-PQoS,and Hybrid (H-PQoS) for richer analytics that combine multiple input sources. Predictions can include uplink or downlink (end-to-end) throughput estimates as well as end-to-end latency forecasts, with horizons extending up to ten seconds. Both subscription-based and request-response exposure modes are available.

Complementing inference, the PQoS Training (PQoS-T) APIs allow vertical applications to initiate the development of predictive AI/ML models when no suitable ML model exists, in order to satisfy a given prediction request. This capability leverages the NWDAF MtLF and the underlying monitoring data to train new enhanced models (for QS, DNP, or Hybrid models), using an asynchronous publish-subscribe mechanism.

Finally, the PQoS enabler supports Connectivity Insights APIs, aligned with CAMARA specifications. These APIs introduce an additional capability layer where verticals can register application profiles with performance thresholds (such as latency or throughput) and subsequently query the network's confidence in meeting these requirements for particular devices. Subscription mechanisms are also available, enabling the continuous delivery of connectivity insight updates to subscribed services.

3.4 PQoS-I

3.4.1 Short description

The PQoS-Inference (PQoS-I) API family provides vertical applications with predicted network and service-level QoS metrics. These predictions enable applications to adapt proactively to anticipated network conditions. The API supports both near-real-time inference and threshold-based notifications, allowing flexibility in how QoS intelligence is consumed by verticals.

3.4.2 Relation with standards

PQoS-I is fully aligned with 3GPP specifications, while abstracting the underlying complexity for the application provider. In the backend, the enabler interacts with the NWDAF Analytics Logical Function (AnLF) for generating predictive analytics. This includes subscribing to relevant network and application-level metrics, performing inference using trained ML models and generating notifications when specified thresholds are crossed. All communication with the NWDAF align with 3GPP TS 29.520 APIs, including AnalyticsSubscription and AnalyticsInfo_Request.

3.4.3 Functional overview

The PQoS-I API supports two modes of operation: subscription-based notification mode and on-demand request-response mode, which are presented in Figure 3.

In the subscription-notification mode (Pub/Sub), applications can subscribe to QoS predictions for specific metrics, prediction methods, and threshold values. Once a subscription is created, the PQoS enabler subscribes to the AnLF, which continuously monitors input data from the network, performs periodic inference and triggers notifications when the defined network KPI thresholds are exceeded. Each subscription generates a unique CorrelationID, which allows the application to modify or delete existing subscriptions. Notifications are delivered to a callback endpoint defined by the application, confirming successful reception with standard HTTP response codes.

Supported subscription management operations include: (a) Add Subscription that creates new subscription, returning a CorrelationID, (b) Edit Subscription that modifies existing subscription using CorrelationID, (c) Get Subscriptions that retrieves all active subscriptions and (d) Delete Subscription that removes an existing subscription using the CorrelationID.

On the other hand, in the request-response (Req-Resp) mode, applications can directly request QoS predictions on-demand without creating a subscription. The API performs inference for the requested metric and returns the predicted values in the response. This mode is simpler and suited for ad-hoc queries, whereas the subscription mode is better for continuous monitoring and threshold-based notifications (In-Advance QoS Notifications – IQN messages).

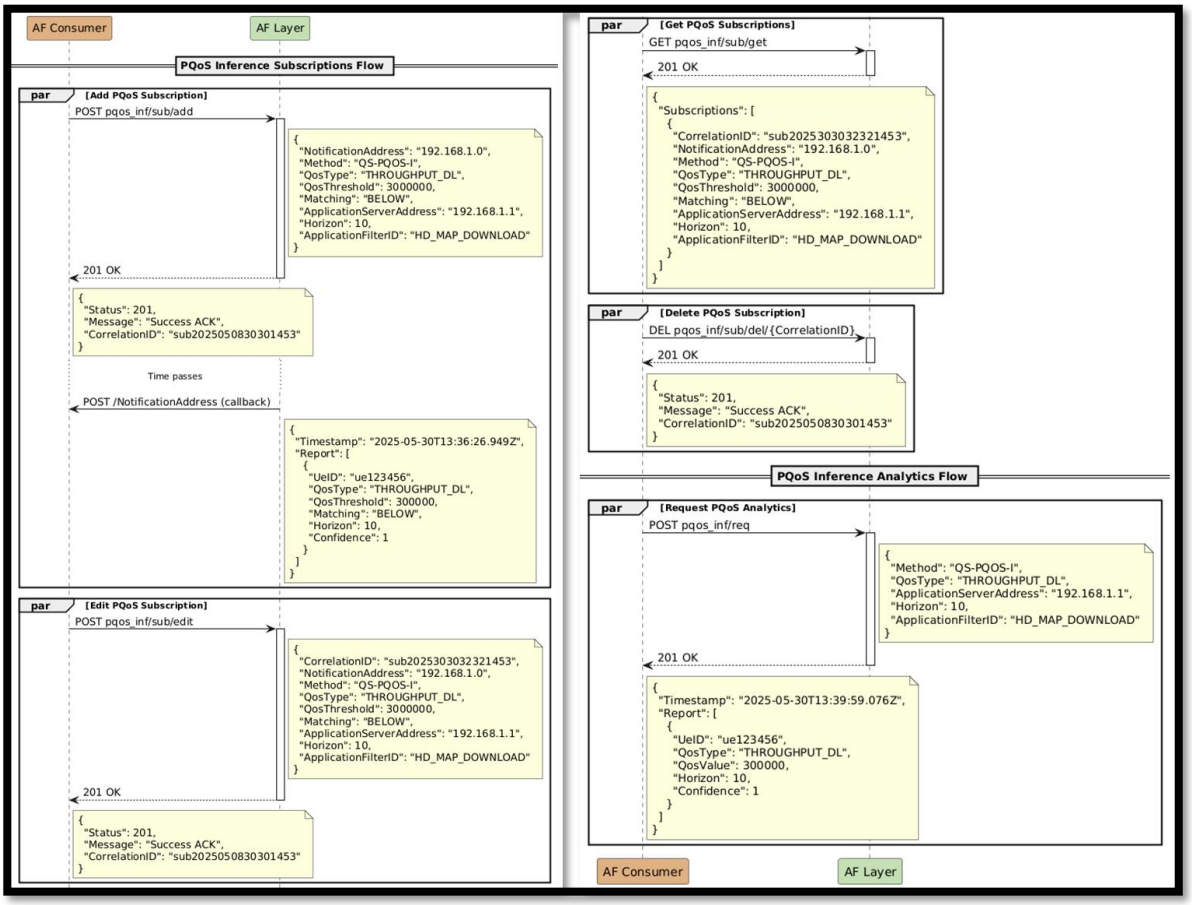

Figure 3 – PQoS-I Operation Modes.

3.4.4 Endpoints

The PQoS-I API exposes a set of RESTful endpoints that implement the two operational modes described in Table 3. These endpoints cover subscription creation and management, delivery of notification callbacks and on-demand prediction requests. All resources follow a consistent structure based on JSON payloads and authenticated access, ensuring predictable integration behavior across applications.

Table 3 – PQoS-Inference Endpoints.

Resource name	Resource URI	HTTP method	Description
Add Subscription	`/pqos_inf/sub/add`	POST	Create new subscription for QoS prediction
Edit Subscription	`/pqos_inf/sub/edit`	POST	Modify an existing subscription
Get Subscriptions	`/pqos_inf/sub/get`	GET	Retrieve all active subscriptions
Delete Subscription	`/pqos_inf/sub/del{CorrelationID}`	DELETE	Delete a subscription using CorrelationID
Request Prediction	`/pqos_inf/request`	POST	Request a one-time prediction (Req/Resp mode)

3.4.5 Request and response examples

POST /pqos_inf/sub/add

Request fields (PqosInfSubAdd data type)

- **NotificationAddress**: Server IPv6 address for receiving PQoS notifications.
- **Method**: Requested prediction method (SIMPLE-QS-PQOS-I, SIMPLE-DN-PQOS-I, QS-PQOS-I, DN-PQOS-I, H-PQOS-I).
- **QosType**: QoS parameter to be predicted (THROUGHPUT_DL, THROUGHPUT_UL, E2E_LATENCY).
- **QosThreshold**: Threshold QoS value that triggers a notification.
- **Matching**: Threshold matching logic (ABOVE or BELOW).
- **ApplicationServerAddress**: IPv6 address of the application server endpoint.
- **ApplicationFilterID**: (Optional) Filter predictions to UEs associated with the given application (only for DN-PQOS-I and H-PQOS-I).
- **Horizon**: Prediction horizon in seconds 1–10.

Response fields (PqosInfSubAddAck data type)

- **Status**: HTTP status code indicating success or failure.
- **Message**: Human-readable status description.

- **CorrelationID**: Unique subscription identifier returned when the subscription is successfully created.

Table 4 – Example for POST /pqos_inf/sub/add.

POST /pqos_inf/sub/add

```
{
  "NotificationAddress": "192.168.0.0",
  "Method": "QS-PQOS-I",
  "QosType": "THROUGHPUT_DL",
  "QosThreshold": 5000000,
  "Matching": "BELOW",
  "ApplicationServerAddress": "192.168.1.1",
  "ApplicationFilterID": "HD_MAP_DOWNLOAD",
  "Horizon": 5
}
```

Response

```
{
  "Status": 201,
  "Message": "Subscription created successfully",
  "CorrelationID": "12f4b880-aa21-4e79-a92d-9fcab124ab0d"
}
```

Callback Notification

```
POST /callback-endpoint
{
  "Timestamp": "2025-11-26T12:30:00Z",
  "Report": [
    {
      "UeID": "UE-01",
      "QosType": "THROUGHPUT_DL",
      "QosThreshold": 5000000,
      "Matching": "Below",
      "Horizon": 5,
      "Confidence": 0.87
    }
  ]
}
```

POST /pqos_inf/sub/edit

Request fields (PqosInfSubEdit data type)

- **CorrelationID**: Identifier of the subscription to be modified.
- **NotificationAddress**: Server IPv6 address for receiving PQoS notifications.
- **Method**: Requested prediction method (SIMPLE-QS-PQOS-I, SIMPLE-DN-PQOS-I, QS-PQOS-I, DN-PQOS-I, H-PQOS-I).
- **QosType**: QoS parameter to be predicted (THROUGHPUT_DL, THROUGHPUT_UL, E2E_LATENCY).
- **QosThreshold**: Threshold QoS value that triggers a notification.
- **Matching**: Threshold matching logic (ABOVE or BELOW).
- **ApplicationServerAddress**: IPv6 address of the application server endpoint.
- **ApplicationFilterID**: (Optional) Filter predictions to UEs associated with the given application (only for DN-PQOS-I and H-PQOS-I).
- **Horizon**: Prediction horizon in seconds 1–10.

Response fields (PqosInfSubAddAck data type)

- **Status**: HTTP status code indicating success or failure.
- **Message**: Human-readable status description.
- **CorrelationID**: Same subscription ID as provided in the request.

Table 5 – Example for POST /pqos_inf/sub/edit.

POST /pqos_inf/sub/edit

```
{
  "CorrelationID": "sub-0001",
  "NotificationAddress": "2001:db8::1",
  "Method": "QS-PQOS-I",
  "QosType": "E2E_LATENCY",
  "QosThreshold": 50,
  "Matching": "ABOVE",
  "ApplicationServerAddress": "2001:db8::100",
  "ApplicationFilterID": "App123",
  "Horizon": 3
}
```

Response

```
{
  "Status": 201,
  "Message": "Subscription updated successfully",
  "CorrelationID": "sub-0001"
}
```

GET /pqos_inf/sub/get

No Request fields

Response fields (PqosInfSubGet data type)

- **Subscriptions**: Array of active subscriptions.
  - **PqosInfSubEdit**: Each subscription object with full subscription details: CorrelationID, NotificationAddress, Method, QosType, QosThreshold, Matching, ApplicationServerAddress, ApplicationFilterID, Horizon.

Table 6 – Example for GET /pqos_inf/sub/get.

GET /pqos_inf/sub/get

Response

```
{
  "Subscriptions": [
    {
      "CorrelationID": "sub-0001",
      "NotificationAddress": "2001:db8::1",
      "Method": "QS-PQOS-I",
      "QosType": "E2E_LATENCY",
      "QosThreshold": 50,
      "Matching": "ABOVE",
      "ApplicationServerAddress": "2001:db8::100",
      "ApplicationFilterID": "App123",
      "Horizon": 3
    },
    {
      "CorrelationID": "sub-0002",
```

```
      "NotificationAddress": "2001:db8::2",
      "Method": "DN-PQOS-I",
      "QosType": "THROUGHPUT_UL",
      "QosThreshold": 2000000,
      "Matching": "BELOW",
      "ApplicationServerAddress": "2001:db8::101",
      "ApplicationFilterID": "App456",
      "Horizon": 4
    }
  ]
}
```

DELETE /pqos_inf/sub/del{CorrelationID}

Request parameters (URI query)

- **CorrelationID**: Subscription ID to be deleted.

Response fields (PqosInfSubAddAck data type)

- **Status**: HTTP status code.
- **Message**: Human-readable status description.
- **CorrelationID**: Identifier of the deleted subscription.

Table 7 – Example for DELETE /pqos_inf/sub/del{CorrelationID}.

DELETE /pqos_inf/sub/del{CorrelationID}
Response

```
{
  "Status": 201,
  "Message": "Subscription deleted successfully",
  "CorrelationID": "sub-0001"
}
```

POST /pqos_inf/request

Request fields (PqosInfReq data type)

- **Method**: PQoS prediction method (SIMPLE-QS-PQOS-I, SIMPLE-DN-PQOS-I, QS-PQOS-I, DN-PQOS-I, H-PQOS-I).
- **QosType**: QoS parameter for prediction (THROUGHPUT_DL, THROUGHPUT_UL, E2E_LATENCY).
- **ApplicationServerAddress**: IPv6 address of the application server endpoint.
- **Horizon**: Prediction horizon in seconds 1–10.
- **ApplicationFilterID**: (Optional) Restrict predictions to UEs for a specific application (only for DN-PQOS-I and H-PQOS-I).

Response fields (PqosInfReqResp data type)

- **Timestamp**: ISO 8601 timestamp of the data used for the prediction
- **Predictions**: List of predicted QoS values
  - **UeID**: ID of the UE for which the prediction applies
  - **QosType**: QoS parameter predicted (THROUGHPUT_DL, THROUGHPUT_UL, E2E_LATENCY)
  - **Horizon**: Prediction horizon in seconds
  - **QosValue**: Predicted KPI value (bps for throughput, ms for latency)
  - **Confidence**: Prediction confidence [0..1]

Table 8 – Example for POST /pqos_inf/request.

POST /pqos_inf/request

```
{
  "Method": "H-PQOS-I",
  "QosType": "THROUGHPUT_DL",
  "ApplicationServerAddress": "2001:db8::100",
  "Horizon": 5,
  "ApplicationFilterID": "App123"
}
```

Response

```
{
  "Timestamp": "2025-11-26T12:35:00Z",
  "Predictions": [
    {
      "UeID": "UE-01",
      "QosType": "THROUGHPUT_DL",
      "Horizon": 5,
      "QosValue ": [5100000, 5100000, 5100000, 5100000, 5100000]
      "Confidence": 0.91
    },
    {
      "UeID": "UE-02",
      "QosType": "THROUGHPUT_DL",
      "Horizon": 5,
      "QosValue ": [5100000, 5100000, 5100000, 5100000, 5100000]
      "Confidence": 0.85
    }
  ]
}
```

3.4.6 Error codes and error handling

The PQoS-I API returns standard HTTP response codes to indicate success or failure of a request. Handling these codes appropriately ensures robust integration with vertical applications:

- 201 – Request OK: The request was successfully processed. For subscription endpoints, a unique CorrelationID is returned for further management.
- 400 – Invalid Input: The request is malformed or contains invalid fields. Applications should verify the request payload, including required fields, data types, and value ranges.
- 401 – Unauthorized: Authentication failed due to missing or invalid credentials. Applications must provide a valid AuthToken in the headers.
- 403 – Forbidden: The client is authenticated but not authorized to perform the requested operation. Verify access rights and subscription permissions.

- 500 – Internal Server Error: The PQoS enabler encountered an unexpected condition. Applications may retry the request after a short delay.
- 501 – Not Implemented: The requested method or functionality is not supported by the server. Applications should avoid using unsupported methods.
- 503 – Service Unavailable: The PQoS service is temporarily unavailable. Applications should implement retry logic with exponential backoff.

Notes on handling notifications (subscription mode): Callback endpoints implemented by the API consumer should respond with 201 - Notification received OK upon successfully processing a QoS notification. Any other HTTP response codes returned by the consumer callback (e.g., 400, 401, 500) indicate failure to receive or process the notification, which may trigger retries depending on the configured logic of the PQoS server.

3.5 PQoS-T

3.5.1 Short description

The PQoS-Training (PQos-T) API allows applications to subscribe to and manage the lifecycle of predictive QoS model training services. It supports automated training of AI/ML models for selected QoS parameters, providing vertical applications with tailored models for inference. The API focuses on subscription-based training notifications.

3.5.2 Relation with standards

The PQoS-T API family is fully aligned with 3GPP standards while providing a simplified interface for vertical applications. Internally, it relies on the NWDAF's Model Training Lifecycle Function (MtLF) to manage the full lifecycle of AI/ML models, including their creation, training, validation, and updates. PQoS-T invokes MtLF to orchestrate these tasks without requiring the application to interact with the underlying NWDAF logic. In addition, the enabler leverages the NWDAF ML Model Provisioning function, which is responsible for storing, versioning and provisioning trained models for subsequent inference by PQoS-I services. By handling all interactions with these standard NWDAF functions internally, the PQoS-T API allows applications to request model training and receive notifications about newly available or updated models without needing to understand the complexity of NWDAF internal operations or configuration. This approach ensures full compliance with 3GPP specifications while providing a practical, deployable solution for application-specific predictive QoS model management.

3.5.3 Functional overview

The PQoS-T API supports subscription-based training notifications, as depicted in Figure 4. Applications can: (a) Subscribe to a training service for a given QoS parameter and prediction method. The PQoS-T server handles data collection and triggers model training, (b) Receive notifications when new models are trained or updated, through a callback endpoint provided by the API consumer, (c) Unsubscribe from training notifications using a unique CorrelationID. This enables applications to react to model updates without managing the full training lifecycle or accessing NWDAF directly.

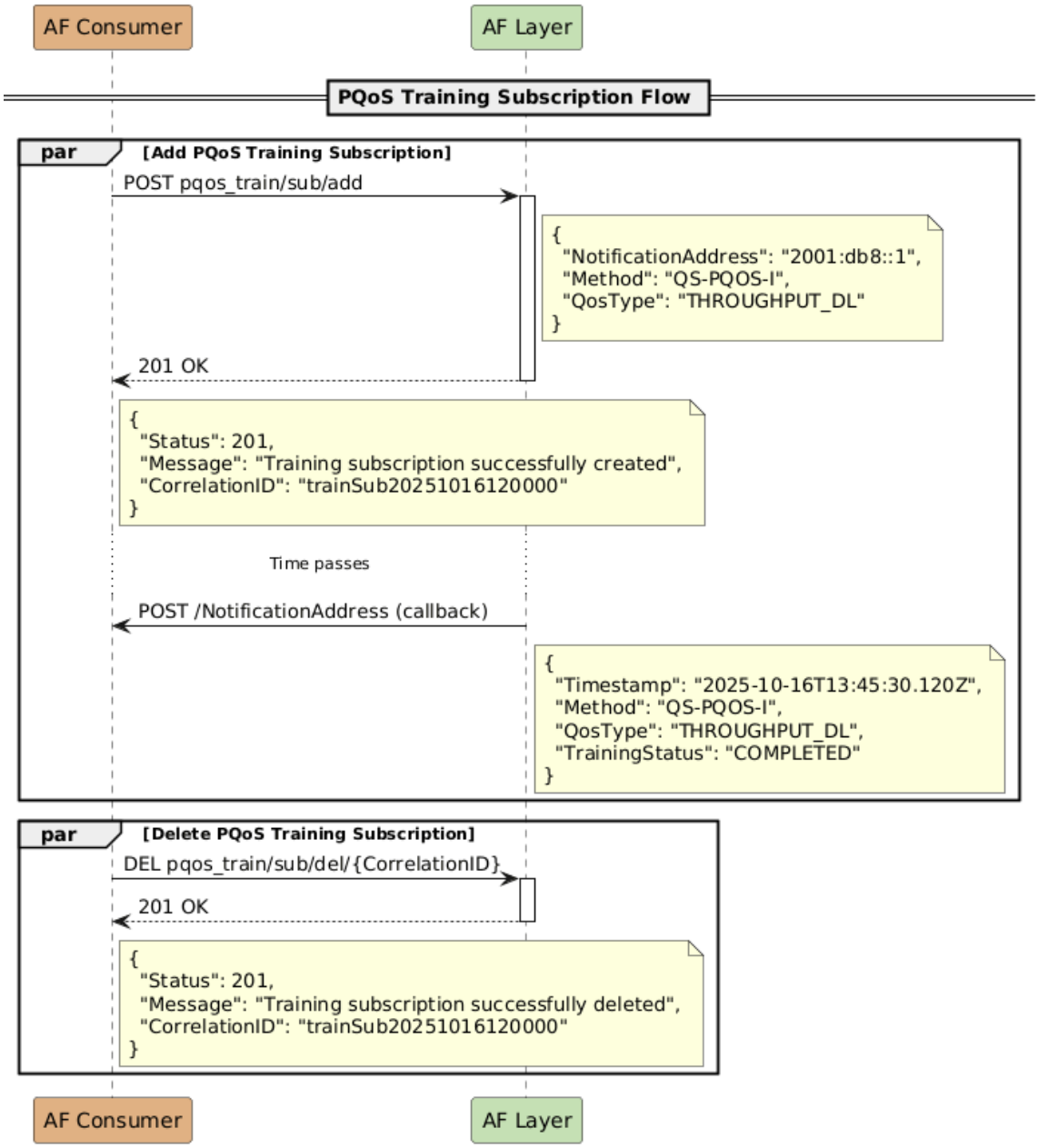

Figure 4 – PQoS-T Overview.

3.5.4 Endpoints

The PQoS-T APIs provide vertical applications with the ability to request the training of new or enhanced predictive QoS models, as well as manage subscriptions for training notifications. The endpoints support subscription management and allow applications to be notified once a model has been successfully trained and provisioned for use in inference. Table 9 summarizes the key PQoS-T resources, their URIs, HTTP methods and a brief description of their functionality:

Table 9 – PQoS-T endpoints.

Resource name	Resource URI	HTTP method	Description
PQoS-T Subscription Add	/pqos_train/sub/add	POST	Subscribe for a PQoS training service. Requests the training of a model for a specific metric and method.
PQoS-T Subscription Delete	/pqos_train/sub/del{CorrelationID}	DELETE	Remove an existing subscription for a PQoS training service using the CorrelationID.

3.5.5 Request and response examples

POST /pqos_train/sub/add

Request fields (PqosTrainSubAdd data type):

- **NotificationAddress**: IPv6 address of the server for training notifications.
- **Method**: PQoS method; supported values {QS-PQOS-I, DN-PQOS-I, H-PQOS-I}.
- **QosType**: QoS parameter for prediction; supported values {THROUGHPUT_DL, THROUGHPUT_UL, E2E_LATENCY}.

Response fields (PqosTrainSubAddAck data type):

- **Status**: HTTP status code.
- **Message**: Response description.
- **CorrelationID**: Subscription ID if successful, otherwise None.

Table 10 – Example for POST /pqos_train/sub/add.

POST /pqos_train/sub/add

```
{
  "NotificationAddress": "2001:db8::1",
  "Method": "QS-PQOS-I",
  "QosType": "THROUGHPUT_DL"
}
```

Response

```
{
  "Status": 201,
  "Message": "Subscription created successfully",
  "CorrelationID": "abcd1234-5678-90ef-gh12-3456ijklmnop"
}
```

Callback Notification

```
POST /callback-endpoint
{
  "Timestamp": "2025-11-26T14:30:00Z",
  "Method": "QS-PQOS-I",
  "QosType": "THROUGHPUT_DL",
  "TrainingStatus": "COMPLETED"
}
```

DELETE /pqos_train/sub/del{CorrelationID}

Request fields (via URI parameter):

- **CorrelationID**: Unique identifier of the subscription to be deleted

Response fields (PqosTrainSubAddAck data type):

- **Status**: HTTP status code returned
- **Message**: HTTP response description
- **CorrelationID**: Returns the same CorrelationID if deletion is successful; null otherwise

Table 11 – Example for DELETE /pqos_train/sub/del{CorrelationID}.

DELETE /pqos_train/sub/del{CorrelationID}
Response

<pre>{
  "Status": 201,
  "Message": "Subscription deleted successfully",
  "CorrelationID": "abc123-def456-ghi789"
}</pre>

3.5.6 Error codes and error handling

The PQoS-T API endpoints return standard HTTP status codes to indicate success or failure of the requested operation. Applications should handle these codes appropriately to ensure robust operation and proper recovery from errors. The main error codes are:

- 201 Created / OK: The request was successful. For subscription add, a CorrelationID is returned. For delete, the same CorrelationID is returned if the deletion succeeded.
- 400 Invalid input: The request was malformed, missing required fields, or contained invalid values. Clients should validate request fields before retrying.
- 401 Unauthorized: The request did not include a valid AuthToken or the token has expired. Clients should ensure proper authentication and reissue requests with a valid token.
- 403 Forbidden: The requesting client does not have sufficient permissions to perform the requested action (e.g., attempting to delete a subscription it does not own).
- 500 Internal server error: An unexpected server-side failure occurred. Clients may retry the request after a short delay.
- 501 Not Implemented: The requested endpoint or operation is not implemented by the server. Clients should avoid sending requests to unsupported resources.
- 503 Service unavailable: The server is temporarily unavailable, for example due to maintenance or high load. Clients should implement exponential backoff or retry logic to handle this condition.

For the Add subscription endpoint, notifications are sent asynchronously to the NotificationAddress. In this context, the same error codes may be returned in the acknowledgment or in response to the notification callback. Applications should ensure proper handling of failures both at subscription creation time and during notification delivery. This approach ensures that clients interacting with PQoS-T services can implement reliable error handling and recovery strategies for both training subscription management and notifications.

3.6 CAMARA Connectivity Insights

3.6.1 Short description

The CAMARA Connectivity Insights API family provides capabilities for defining application-specific network quality requirements, retrieving connectivity insights, and subscribing to notifications about network performance. These APIs support latency and UL/DL throughput thresholds for applications and allow real-time monitoring of network conditions for end-user devices. The implementation integrates with NWDAF functions to leverage predictive network intelligence.

3.6.2 Relation with standards

CAMARA APIs are aligned with the specifications defined by the CAMARA project. Internally, the implementation utilizes NWDAF functionality to obtain analytics and predictions, ensuring compliance with 3GPP network analytics principles while providing a simplified interface for vertical applications.

3.6.3 Functional overview

The CAMARA API family is divided into three main groups: (a) Application Profiles: allows creation, retrieval, update and deletion of application-specific network quality profiles that define thresholds for metrics such as latency and throughput, (b) Connectivity Insights: enables applications to request the predicted network performance against the defined thresholds for a specific device or application profile, returning the network's confidence in meeting the requested quality levels and (c) Subscriptions: supports creation and management of subscriptions to receive periodic notifications of connectivity insights, allowing applications to react to changing network conditions without continuous polling. The functionalities are presented in Figure 5.

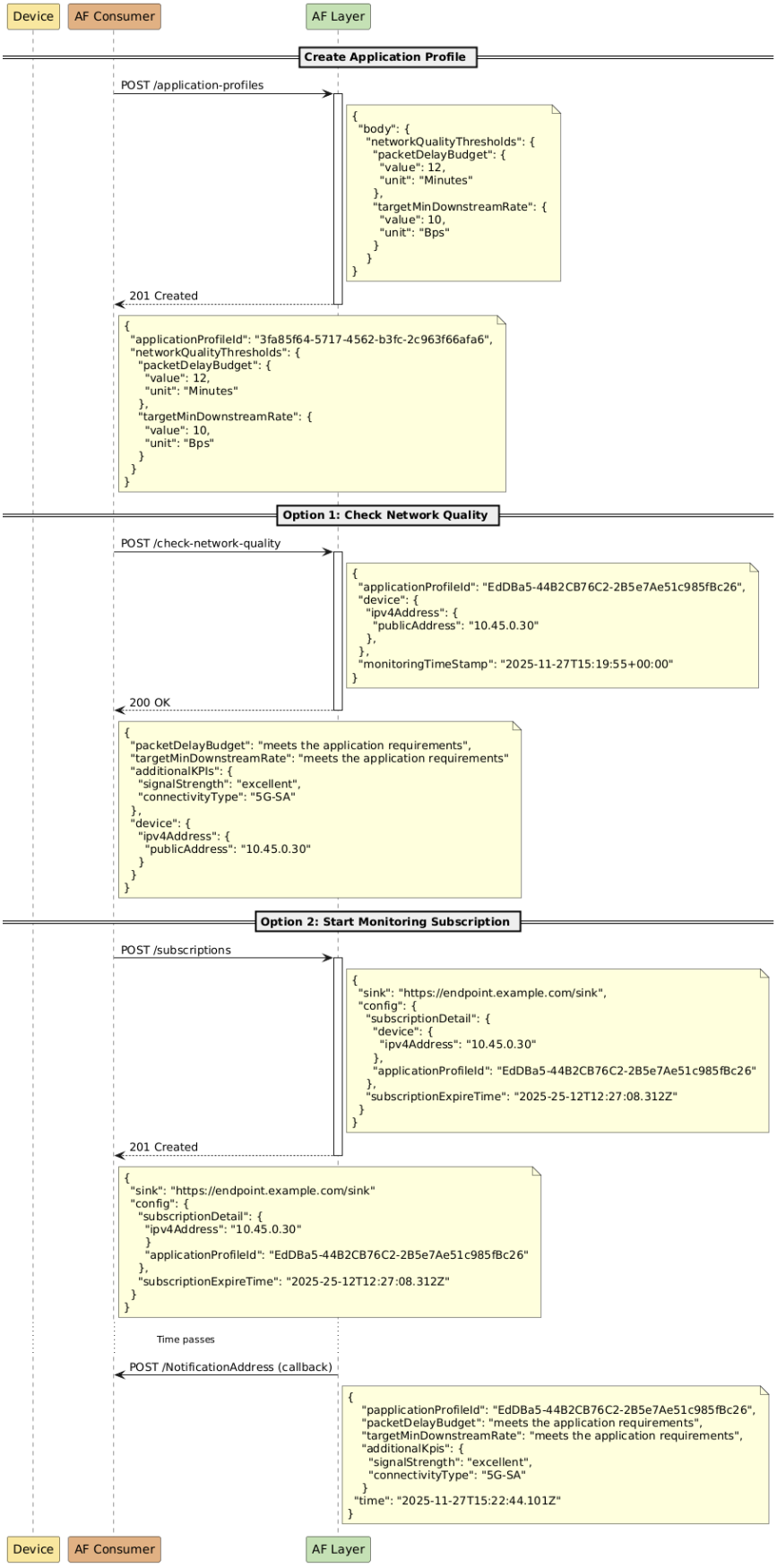

Figure 5 – CAMARA Connectivity Insights APIs overview.

3.6.4 Endpoints

The CAMARA Connectivity Insight API family exposes a set of RESTful resources that enable applications to manage network quality profiles, request connectivity insights, and subscribe to notifications about predicted network performance. The endpoints are presented in Table 12 below. Each endpoint is designed to provide a clear and standard-compliant interface for application developers, hiding the underlying NWDAF analytics complexity while supporting essential operations for predictive QoS and connectivity management.

Table 12 – CAMARA Connectivity Insights Endpoints.

Resource name	Resource URI	HTTP method	Description
Application Profile Create	/camara/application_profiles	POST	Create a new application profile with defined network quality thresholds.
Application Profile Update	/camara/application_profiles/{applicationProfileId}	PATCH	Update an existing application profile.
Application Profile Get	/camara/application_profiles/{applicationProfileId}	GET	Retrieve details of a specific application profile.
Application Profile Delete	/camara/application_profiles/{applicationProfileId}	DELETE	Remove a specific application profile.
Subscription Create	/subscriptions	POST	Create a subscription to receive periodic connectivity insights for a device.
Subscription Get All	/subscriptions	GET	Retrieve all active subscriptions for the client.
Subscription Get Single	/subscriptions/{subscriptionId}	GET	Retrieve a single subscription by ID.
Subscription Delete	/subscriptions/{subscriptionId}	DELETE	Delete a specific subscription.
Connectivity Insights Request	/check-network-quality	POST	Request predicted network quality for a device or application profile.

3.6.5 Request and response examples

POST /camara/application_profiles

Request fields (ApplicationProfile data type):

- **networkQualityThresholds.packetDelayBudget.value**: maximum acceptable one-way latency (e.g. in ms or sec)
- **networkQualityThresholds.packetDelayBudget.unit**: time unit (e.g. "ms" or "seconds")
- **networkQualityThresholds.targetMinDownstreamRate.value**: minimum required downlink throughput (in bps, Kbps, Mbps, Gbps, Tbps)

- **networkQualityThresholds.targetMinDownstreamRate.unit**: rate unit (e.g. bps, Kbps, Mbps, Gbps, Tbps)
- **networkQualityThresholds.targetMinUpstreamRate.value**: minimum required uplink throughput (in bps, Kbps, Mbps, Gbps, Tbps)
- **networkQualityThresholds.targetMinUpstreamRate.unit:** rate unit (e.g. bps, Kbps, Mbps, Gbps, Tbps)
- **networkQualityThresholds.packetLossErrorRate**: acceptable packet loss/error rate from 1 to 10 (The exponential power of the allowable error loss rate 10^(-N))
- **networkQualityThresholds.jitter.value**: maximum acceptable jitter (variation in packet delay)
- **networkQualityThresholds.jitter.unit**: time unit for jitter

Response fields (PqosTrainSubAddAck data type):

- **applicationProfileId**: unique identifier assigned to the new profile
- **networkQualityThresholds**: echo of the thresholds defined in the request (or normalized form)

Table 13 – Example for POST /camara/application_profiles.

POST /camara/application_profiles

```
{
  "networkQualityThresholds": {
    "packetDelayBudget": { "value": 100, "unit": "ms" },
    "targetMinDownstreamRate": { "value": 8000000, "unit": "bps" },
    "targetMinUpstreamRate": { "value": 1000000, "unit": "bps" },
    "packetLossErrorRate": 0.01,
    "jitter": { "value": 30, "unit": "ms" }
  }
}
```

Response

```
{
  "applicationProfileId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "networkQualityThresholds": {
    "packetDelayBudget": { "value": 100, "unit": "ms" },
    "targetMinDownstreamRate": { "value": 8000000, "unit": "bps" },
    "targetMinUpstreamRate": { "value": 1000000, "unit": "bps" },
    "packetLossErrorRate": 0.01,
    "jitter": { "value": 30, "unit": "ms" }
  }
}
```

POST /check-network-quality

Request fields:

- **applicationProfileId**: ID of an existing application profile (created via application-profiles API)
- **device**: object identifying the user device — one or more of: ipv4Address, ipv6Address, phoneNumber, networkAccessIdentifier
- **applicationServer**: object specifying the application server endpoint — can include ipv4Address and/or ipv6Address (and possibly port or other details)

Response fields (ConnectivityInsightsResult data type):

- **meetsRequirements**: boolean indicating whether network meets the threshold requirements of the given profile
- **predictedQuality**: object containing quantitative predicted metrics (e.g. latency, jitter, packetLoss, downlinkRate, uplinkRate)
- **confidence**: numeric confidence score (e.g. 0 to 1) indicating how reliable the prediction is

Table 14 – Example for POST /check-network-quality.

POST /check-network-quality

```
{
  "applicationProfileId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "device": {
    "ipv6Address": "2001:db8:85a3::8d3:1319:8a2e:370:7344"
  },
  "applicationServer": {
    "ipv6Address": "2001:db8:abcd::100"
  }
}
```

Response

```
{
  "packetDelayBudget": "meets the application requirements",
  "targetMinDownstreamRate": "meets the application requirements",
  "targetMinUpstreamRate": "meets the application requirements",
  "packetlossErrorRate": "meets the application requirements",
  "jitter": "meets the application requirements",
  "additionalKPIs": {
    "signalStrength": "excellent",
    "connectivityType": "5G-SA"
  },
  "device":{
    "ipv6Address": "2001:db8:abcd::100"
  }
}
```

POST camara/subscriptions

Request fields (SubscriptionRequest data type):

- **device**: object identifying the user device (ipv4/ipv6 address, phone number or NAI)
- **applicationServer**: object specifying the application server endpoint (ipv4/ipv6, port/range)
- **applicationProfileId**: existing application profile ID
- **sink**: callback URL where connectivity insight events will be delivered
- **protocol**: e.g. "HTTP" or other supported protocol for callbacks
- (Optional) **sinkCredential**: credentials or authentication details for callback delivery

Response fields (SubscriptionResponse data type):

- **subscriptionId**: unique identifier for the created subscription
- **status**: HTTP status code or a status indicator string
- **message**: human-readable description (e.g. “Subscription created successfully”)

Table 15 – Example for POST camara/subscriptions.

POST camara/subscriptions

```
{
  "protocol": "HTTP",
  "sink": "https://endpoint.example.com/sink",
  "sinkCredential": {
    "credentialType": "PLAIN"
  },
  "types": [
    "org.camaraproject.connectivity-insights-subscriptions.v0.network-quality"
  ],
  "config": {
    "subscriptionDetail": {
      "device": {
        "ipv4Address": {"publicAddress" : "10.45.0.30"}
      },
      "applicationProfileId": "12e2c376-700f-4e8c-ad03-3ad4292d8e6a"
    },
    "subscriptionExpireTime": "2025-25-12T12:27:08.312Z",
    "subscriptionMaxEvents": 20,
    "initialEvent": true
  }
}
```

Response

```
{
  "protocol": "HTTP",
  "sink": "https://endpoint.example.com/sink",
  "types": [
    "org.camaraproject.connectivity-insights-subscriptions.v0.network-quality"
  ],
  "config": {
    "subscriptionDetail": {
      "device": {
        "ipv4Address": {
          "publicAddress": "10.45.0.30"
        },
      },
      "applicationProfileId": "12e2c376-700f-4e8c-ad03-3ad4292d8e6a"
    },
    "subscriptionExpireTime": "2025-25-12T12:27:08.312000Z",
    "subscriptionMaxEvents": 20,
    "initialEvent": true
  },
  "subscriptionId": "73b45795-7e9b-4748-9c70-2ac1a8ddcd83",
  "startsAt": "2025-11-26T08:30:49.886428Z",
  "expiresAt": "2023-07-03T12:27:08.312000Z",
  "status": "ACTIVATION_REQUESTED"
}
```

Callback Notification

```
{
  "protocol": "HTTP",
  "sink": "https://endpoint.example.com/sink",
  "types": [
    "org.camaraproject.connectivity-insights-subscriptions.v0.network-quality"
  ],
  "config": {
    "subscriptionDetail": {
      "device": {
        "ipv4Address": {
          "publicAddress": "10.45.0.30"
        },
      },
      "applicationProfileId": "12e2c376-700f-4e8c-ad03-3ad4292d8e6a"
```

```
    },
    "subscriptionExpireTime": "2025-25-12T12:27:08.312000Z",
    "subscriptionMaxEvents": 20,
    "initialEvent": true
  },
  "subscriptionId": "73b45795-7e9b-4748-9c70-2ac1a8ddcd83",
  "startsAt": "2025-11-26T08:30:49.886428Z",
  "expiresAt": "2023-07-03T12:27:08.312000Z",
  "status": "ACTIVATION_REQUESTED"
}
```


3.6.6 Error codes and error handling

CAMARA APIs follow standard HTTP status codes:

- 201 Created / OK: Request was successfully processed. Subscriptions or profiles are created.
- 400 Invalid input: Request is missing required fields or contains invalid values. Applications should validate requests before retrying.
- 401 Unauthorized: The request is not properly authenticated; a valid token is required.
- 403 Forbidden: The client does not have permission to perform the requested operation.
- 404 Not Found: The specified resource (application profile, subscription) does not exist.
- 500 Internal server error: Unexpected server-side error. Retry logic may be applied.
- 503 Service unavailable: Temporary unavailability due to maintenance or load; clients should implement retry with backoff.

Notifications via subscription callbacks also return standard HTTP response codes to confirm delivery or indicate errors. Proper handling ensures robust integration with vertical applications.

3.7 CAMARA Predictive Connectivity Data

3.7.1 Short description

The CAMARA Predictive Connectivity Data API provides developers with the ability to predict what the connectivity quality level of the mobile network connectivity for a specific geographic area and time will be in the future. It uses, amongst others, historical and current data of the network to perform such prediction. This API is relevant for use cases such as tele-operation driving or drone operations, where connectivity predictions help identify safe zones to operate and to trigger a mobile device (e.g., vehicle or drone) route adaptation timely when necessary.

3.7.2 Relation with standards

This specification complies with the Predictive Connectivity Data API following CAMARA specification version 0.1.0 (September 2025), see exact git commit for the YAML specification implemented here. In addition, the current implementation complies with the 3GPP specification TS 29.522 v17.17.0 for the NEF AnalyticsExposure (*qosSustainInfos*) interface and also with 3GPP 29.520 v17.16.0 for the corresponding NWDAF AnalyticsInfo (*qosSustainInfos*) interface.

3.7.3 Functional overview

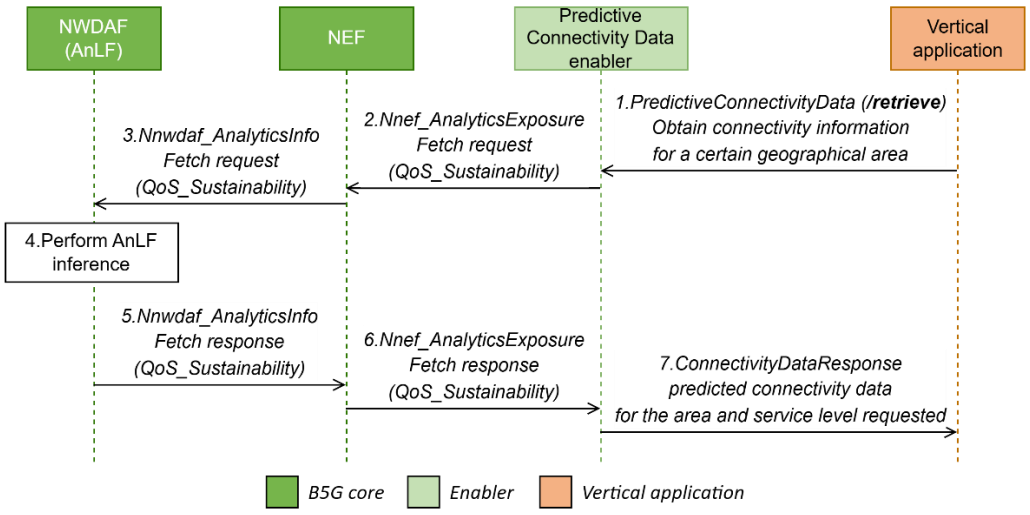

Figure 6 – Message flow for the Predictive Connectivity Data API.

The message flow procedure for the *Predictive Connectivity Data* API is shown in Figure 6. Steps 2-6 follow the 3GPP specifications TS 29.522 (NEF) AnalyticsExposure and TS 29.520 (NWDAF) AnalyticsInfo, both with corresponding analytics event set to *QoS Sustainability* (qosSustainInfos).

1. The vertical application sends a PredictiveConnectivityData (retrieve) request to obtain connectivity information for a specific geographical area. In this request, the requested service level, geographical area, and time interval for the prediction are included.
2. The Predictive Connectivity Data enabler sends an Nnef_AnalyticsExposure Fetch request (*QoS_Sustainability*) to the NEF.
3. The NEF sends an Nnwdaf_AnalyticsInfo Fetch request (*QoS_Sustainability*) to the NWDAF.
4. The NWDAF's sub-component AnLF (Analytics logical function) performs the AI model inference for the prediction request received.
5. The NWDAF returns an Nnwdaf_AnalyticsInfo Fetch response to the NEF.
6. The NEF forwards the response as Nnef_AnalyticsExposure Fetch response to the Predictive Connectivity Data enabler.
7. The Predictive Connectivity Data enabler processes the received information from the NEF and sends a ConnectivityDataResponse with predicted connectivity data for the requested area and service level back to the vertical application.

3.7.4 Endpoints

Resource name	Resource URI	HTTP method	Description
Connectivity Data	/retrieve	POST	Retrieves the connectivity for a concrete area, specifying the required service level that has to be provided.

3.7.5 Request and response examples

POST /retrieve

Request fields (*RetrieveConnectivityRequest* data type):

- **serviceLevel**: Describes the requested communication service level. The possible service levels are as follows: C2 (Command and Control), STREAM_4K (Streaming in 4K quality). More service levels may be added later in the project to support other use cases.
- **area**: description of the area where the service level prediction is requested for.
  - **areaType**: Type of this area. Only POLYGON is supported.
  - **boundary**: array with latitude and longitude points defining the area.
- **startTime**: Start date time of the prediction time window. It must follow RFC 3339 and must have time zone.
- **endTime**: End date time of the prediction time window. It must follow RFC 3339 and must have time zone.
- **networkType**: Specifies the type of connectivity for which to retrieve data. Only supported value is 5G.
- **precision**: Precision required of response cells. Precision defines a geohash level and corresponds to the length of the geohash for each cell. More information at Geohash system. If not included, the default precision level 7 is used. In case of using a not supported level by the network, the API returns the error response UNSUPPORTED_PRECISION.
- **height**: Requested prediction height in meters above ground level.
- **includeSignalStrength**: Optional parameter to request raw signal strength values in dBm for each layer.
- **sink**: The address where the API response will be asynchronously delivered, using the HTTP protocol.
- **sinkCredential**: A sink credential provides authentication or authorization information necessary to enable the delivery of events to a target.

Response fields (*ConnectivityDataResponse* data type):

- **layerThickness**: Thickness in metres of each layer.
- **timedConnectivityData**: Array of time intervals with connectivity data for the cells in the area.
  - **startTime**: Start date time of the prediction time window. It must follow RFC 3339 and must have time zone.
  - **endTime**: End date time of the prediction time window. It must follow RFC 3339 and must have time zone.
  - **cellConnectivityData**: Connectivity data for the different cells in a particular time intreval. This includes an array of representation data of a rectangular grid cell within the specified area, including its connectivity data with fields: geohash, layerConnectivities, and optional layerSignalStrengths.
- **status**: Represents the state of the response for the input polygon defined in the request, the possible values are: SUPPORTED_AREA, PART_OF_AREA_NOT_SUPPORTED, AREA_NOT_SUPPORTED, OPERATION_NOT_COMPLETED.

- **statusInfo**: Information about the status, mandatory when property status is OPERATION_NOT_COMPLETED for adding extra information about the error.
- **requestedHeight**: Optional height information in metres above ground level that was requested in the request. This property is only present if a specific height was requested.

Table 16 – Example for POST /retrieve

POST /retrieve

```
{
  "serviceLevel": "Teleoperation (TBD)",
  "area": {
    "areaType": "POLYGON",
    "boundary": [
      {"latitude": 50.735851,"longitude": 7.10066},
      {"latitude": 50.74,"longitude": 7.11},
      {"latitude": 50.73,"longitude": 7.12}
    ]
  },
  "startTime": "2025-01-03T11:00:00Z",
  "endTime": "2025-01-03T12:00:00Z",
}
```

Response

```
{
    "layerThickness": 10,
    "timedConnectivityData": [
        {
            "startTime": "2025-01-03T11:00:00Z",
            "endTime": "2025-01-03T12:00:00Z",
            "cellConnectivityData": [
                {
                    "geohash": "ezdmemd",
                    "layerConnectivities": [
                        "MC"
                    ]
                }
            ]
        }
    ],
    "status": "SUPPORTED_AREA"
}
```

3.7.6 Error codes and error handling

The status codes and their handling are implemented as specified in Predictive Connectivity Data API following CAMARA specification version 0.1.0 (September 2025), see exact git commit for the YAML specification implemented here.

3.8 Implementation Details

This section presents the overall design and the main components of the ENVELOPE Predictive QoS (PQoS) enabler, together with the API families through which it is exposed to vertical services. It outlines how the system architecture supports predictive network analytics, predictive model training, and application-level integration, while positioning ENVELOPE's PQoS approach with respect to existing standards and prior implementations.

The Monitoring Platform infrastructure leverages Prometheus for metric collection and Grafana for visualization. It collects standard 5G RAN and core network KPIs, including gNB radio indicators, AMF/SMF/UPF load, and UE performance metrics. Custom Prometheus exporters and plugins are deployed to capture over-the-top, application-level metrics. Data is time-series stored and pre-processed in Prometheus for compatibility with NWDAF ingestion. The platform operates in a fully containerized environment using Docker Compose, enabling rapid deployment, updates, and scalability. Pre-processing jobs transform raw metrics into formats suitable for analytics, applying slice-specific and service-level aggregation where required.

The NWDAF backend implements a subset of the 3GPP-specified functionality, focusing on key modules: AnLF, MtLF, subscription management, southbound data access and notification delivery. The AnLF exposes RESTful APIs compliant with 3GPP TS 29.520 for DN_PERFORMANCE, NF_LOAD, and NSI_LOAD events. It is implemented in GoLang using the Gorilla/mux library, with event data persisted in PostgreSQL. Southbound interactions retrieve metrics from the Prometheus monitoring platform. The MtLF handles ML model lifecycle operations, including training orchestration, versioning, metadata management, and subscription-based notification, enabling predictive analytics provisioning for vertical applications. Dockerized microservices ensure modular deployment, rapid scaling, and automated health checks.

The NWDAF originates from a legacy software component developed within the 6G Sandbox OpenCall project Analysat, which served as the foundational analytics framework. Within the ENVELOPE project, this NWDAF implementation has been extended to support the QOS_SUSTAINABILITY service, enabling dynamic quality-of-service monitoring and predictive control. Furthermore, it has been enhanced to invoke AI-based prediction services whenever a requested metric is related to a future time horizon. These extensions also integrate native AI training mechanisms through the MtLF, aligning with 3GPP-compliant APIs to ensure orchestration of model updates and analytics provisioning across network functions.

In regards to AI/ML pipelines, predictive modeling relies on Python and PyTorch, using Long short-term memory (LSTM) deep neural networks for time-series forecasting. Feature engineering combines prior state-of-the-art techniques[1] with insights from 3GPP TS guidelines[2] and follows both, academic and commercial best practices. The pipelines are fully dockerized, ensuring reproducibility, scalability, and fast deployment.

ENVELOPE exposes PQoS functionality to vertical services (AF Layer) via FastAPI-based asynchronous microservices. These APIs handle authentication, subscription management, and translation of NWDAF outputs into application-friendly formats. Local databases support fast query responses and caching. The abstraction layer shields vertical applications from low-level NWDAF message structures and core network configuration details, providing immediate access to

[1] Palaios, Alexandros, et al. "Machine learning for QoS prediction in vehicular communication: Challenges and solution approaches." *IEEE Access* 11 (2023): 92459-92477.

Miekkala, Topi, et al. "Nordicdat: A cross-border predictive qos dataset." *GLOBECOM 2024-2024 IEEE Global Communications Conference*. IEEE, 2024.

[2] 3GPP. (2022). *Architecture enhancements for 5G System (5GS) to support network data analytics services (TS 23.288, Release 17)*. ETSI.

predictive QoS insights. The entire API stack is containerized for seamless integration and continuous deployment using Docker Compose.

Deployment-wise the full ENVELOPE PQoS stack (including Monitoring Platform, NWDAF backend, AI/ML pipelines, and API exposure) is deployed using Docker Compose, enabling modular container orchestration, automated health checks, and rapid software updates. This approach allows independent scaling of components while maintaining reproducibility and portability across testbeds or operational environments.

3.9 Lessons Learned

The API definition and development phase of the ENVELOPE PQoS enabler provided valuable insights into designing application-facing interfaces that expose complex 3GPP NWDAF functionality to vertical services. One key learning was the importance of abstraction: simplifying data models and API calls enabled vertical applications to access predictive QoS metrics without needing to understand low-level NWDAF structures or 5G core configurations. Designing northbound APIs using asynchronous frameworks (FastAPI) facilitated easy integration, reduced response latencies, and allowed for intuitive subscription and notification mechanisms for predictive analytics. The abstraction layer successfully decoupled vertical services from the underlying network complexities, proving essential for maintainability and adoption.

Backend communication relied heavily on microservice architecture, which allowed for compartmentalization of functionality between the PQoS enabler, NWDAF analytics, and AI/ML pipelines. This approach simplified updates and redeployments, as individual services could be upgraded or restarted without impacting the entire system. Dockerized deployments further accelerated iteration cycles and ensured reproducibility across development and testbed environments.

However, several limitations emerged during implementation. Integration with 5G RAN and core network plugins was constrained by the metrics exposed by specific hardware/software stacks, such as Open5GS and Amarisoft RAN, which provide a limited set of KPIs. AI/ML pipelines faced trade-offs between model complexity and inference latency; very deep or wide models increased prediction accuracy marginally but substantially delayed response times. Furthermore, 3GPP specifications do not define per-UE predictive mechanisms, focusing instead on cell- or slice-level metrics, necessitating custom adaptations to deliver per-UE insights. Measurement collection times posed additional challenges, requiring backend optimizations and workaround techniques to ensure timely data for inference. Maintaining multiple model variants also revealed scalability issues, as each model required individualized configurations, complicating automation when supporting dozens or hundreds of simultaneous model instances.

Next steps focus on validating the complete end-to-end PQoS pipelines within real testbeds and trial sites (Greek and Dutch). This includes evaluating predictive QoS API performance metrics such as inference latency, prediction accuracy, and API reliability under realistic network conditions. Comparative studies of different model variants will assess trade-offs between training and inference time, computational cost, and predictive accuracy. Finally, MLOps pipelines will be stress-tested in real-time scenarios, examining how automated model provisioning, retraining, and deployment behave under operational load when vertical services request new models. These evaluations will provide critical insights for scaling PQoS enablers for deployment across diverse verticals and network environments.

4 ATSSS Multi-Connectivity

4.1 Overview and Innovations

This section provides an overview of the developments on Access Traffic Steering, Switching and Splitting (ATSSS). Figure 7 provides an overview of the ATSSS enabler. ENVELOPE introduces three APIs to manage multi-connectivity over two links, 3GPP and non-3GPP network paths, namely *ATSSS-T* (exposed to the application server), *ATSSS-Cu* (internal), and *ATSSS-Cp* (internal). These APIs facilitate the interaction between the following components:

- **CAM App Server:** Requests multi-connectivity from the ATSSS policy function via the *ATSSS-T* API. This request includes the specification of a scheduling rule. As part of this deliverable, minimum RTT and intelligent load balancing rules have been implemented.
- **ATSSS Policy Function (APF):** Resides in the core network and manages the ATSSS session between the CAM App Server and the CAM App Client.
- **ATSSS Layer** on the **UE (AUE)** and **Network (ANE)** sides**:** The client and the server establish an MPQUIC session via two MPQUIC service instances. The AUE resides on the UE and the ANE is integrated into the core network. These service instances provide an ENVELOPE ATSSS software layer. Each instance comprises an MPQUIC scheduler that adheres to the rules received from the APF via ATSSS-Cu (AUE) and ATSSS-Cp (ANE). The AUE controls the uplink traffic, and the ANE controls the downlink traffic. If the load-balancing ATSSS rule is active, the AUE further uses *ATSSS-Cu* to send its desired traffic ratios to the APF. The traffic ratios are determined by a **Reinforcement Learning (RL) agent** that makes decisions based on the locally observed network conditions.
- **CAM App Client:** The client connects with the CAM App Server through the ATSSS layer.

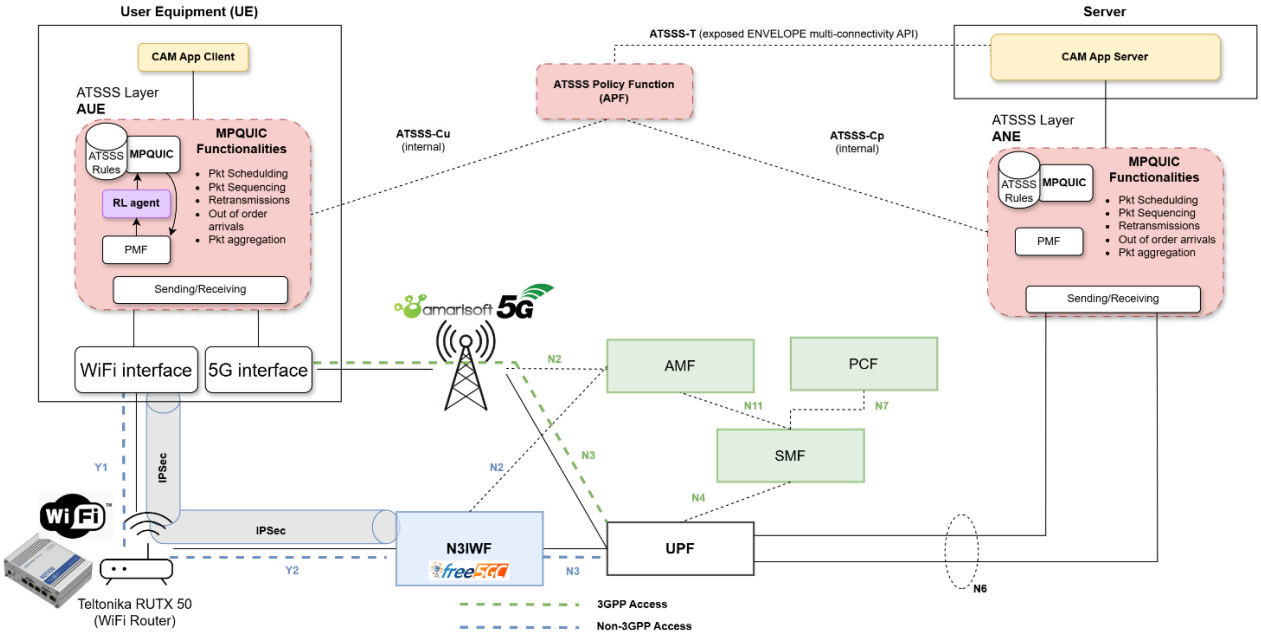

Figure 7 – High-level overview of ATSSS. The ATSSS-T API facilitates MPQUIC session creation, and the internal ATSSS-Cu and ATSSS-Cp control the MPQUIC service instances on the UE (AUE) and in the Network (ANE).

ENVELOPE ATSSS provides the following innovations over existing frameworks and standards:

- ATSSS-T implements a subset of the Multi-access Traffic Steering (MTS) service defined in ETSI GS MEC 015. It provides an abstraction layer to steer traffic between 3GPP and non-3GPP networks and is designed for seamless integration with CAM app services. The focus on traffic steering between two network paths eases implementation and usability of multi-connectivity in the CAM application context.
- The APF expands upon the functionality of MTS by facilitating communication to the AUE and ANE. While MTS is designed to manage multi-connectivity sessions on an edge server, the APF can manage multiple MPQUIC services running at different locations in the core network and on the UE.
- In addition to the CAM app server specifying scheduling rules, the ATSSS-Cu API enables UE-side control of the scheduler. ENVELOPE proposes an intelligent RL agent that runs on the UE and controls the ratio between 3GPP and non-3GPP traffic. An RL environment abstracts over the real underlying system and enables easy training and deployment of learning-based methods. Their goal is to reduce end-to-end latency and increase throughput over multi-path schedulers that use manually designed heuristics.

The next sections introduce the individual ENVELOPE ATSSS APIs. Afterwards, we provide implementation details, including the functionality of the RL agent.

4.2 ATSSS-T API

4.2.1 Short description

The ATSSS-T API is exposed to the vertical service and allows it to request and control a multi-connectivity session. The vertical service implements an external Application Function (AF) that connects to the ATSSS policy function (APF) through ATSSS-T.

4.2.2 Relation with standards

ATSSS-T is inspired by the Multi-access Traffic Steering (MTS) service defined in the ETSI MEC 015 specification. ATSSS-T implements a subset of MTS and is tailored for CAM applications by focusing on two network paths with 3GPP and non-3GPP traffic.

4.2.3 Functional overview

ATSSS-T is responsible for multi-connectivity session management (see Table 17). Figure 8 illustrates the interaction between the server and the APF enabled through ATSSS-T.

Table 17 – Summary of ATSSS-T.

API name	Description	Modules Involved	Use Cases
ATSSS-T	Enables a vertical service to request multi-connectivity via the APF.	APF, CAM App Server	GR-UC-1

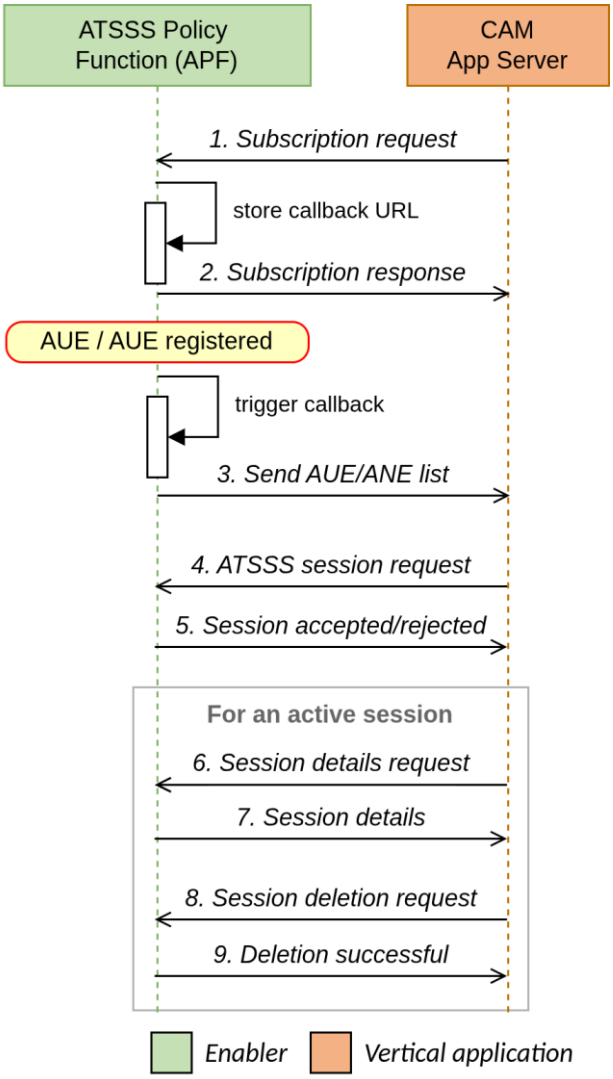

Figure 8 – Message flow for the ATSSS-T API.

The session establishment procedure is as follows:

1. The CAM app server (vertical) sends a subscription request to the APF for receiving updates whenever a new AUE/ANE for this service registers with the APF.
2. The APF confirms the subscription.
3. Upon changes, the vertical receives a list of registered AUEs and ANEs from the APF.
4. The vertical sends a session request to the APF through ATSSS-T. This request specifies the involved AUE, ANE, and the ATSSS rule the vertical wants to use to manage traffic.
5. The APF indicates whether an application of the requested rule is feasible or not and activates the session if activation is possible.
6. Upon creation of the session, the vertical can request the session details (can also be retrieved asynchronously via a callback at session creation time).
7. The APF responds with the session details, including the selected rule and active settings (which may have been manipulated by the UE in the meanwhile).
8. The vertical may send a request to delete/close the session. If the APF receives no deletion request, the session automatically expires after a fixed time.
9. The deletion of the session is confirmed by the APF.

4.2.4 Endpoints

ATSSS-T exposes three endpoints on the APF, which are summarized in Table 18. The following section 4.2.5 details the fields of respective requests and responses.

Table 18 – Overview of ATSSS-T endpoints.

Resource name	Resource URI	HTTP method	Description
ATSSS-T	atsss-t/sub	POST	Subscribe to ATSSS registration notifications
ATSSS-T	atsss-t/session	POST	Requests a new ATSSS session with multi-connectivity
ATSSS-T	atsss-t/session/{sessionID}	GET	Retrieves details of an active ATSSS session, including applied steering rules
ATSSS-T	atsss-t/session/{sessionID}	DELETE	Ends an ongoing ATSSS session

4.2.5 Request and response examples

POST atsss-t/sub

Request fields

- **serviceID**: the identifier of the service provided by the vertical
- **callback**: a callback URL that will be used to notify the vertical about registrations

The direct response by the APF only notifies the vertical about the successful subscription.

The asynchronous callback response comprises the fields

- **aueList:** list of AUEs registered for this service that are not part of an ATSSS session yet
- **aneList:** list of ANEs registered for this service that are not part of an ATSSS session yet

Table 19 – Example for GET atsss-cu/sub.

Vertical➔ APF: POST atsss-t/sub

```
{
  "serviceID": "dummyService",
  "callback": "dummyServiceAddress/callback",
}
```

Response: 201 Created

APF ➔ Vertical: POST dummyServiceAddress/callback (asynchronous callback)

```
{
  "aueList": ["dummyAUE"],
  "aneList": ["dummyANE"],
}
```

POST atsss-t/session

Request fields

- **serviceID:** Unique identifier of the requesting vertical service.
- **callbackUri:** Optional. Can be provided by the vertical for allowing the APF to actively forward session updates (triggered by the AUE) to the CAM app server. The content sent via the callback is identical to the response for GET atsss-t/session/{sessionID}.
- **flowFilter:** Traffic flow criteria
  - **aueID:** Identifier of the AUE that should be part of the ATSSS session (e.g., IP address and port of the UE)
  - **aneID:** Identifier of the ANE that should be part of the ATSSS session (e.g., IP address and port of the requesting server)
  - **protocol:** Underlying transport protocol (e.g., TCP, UDP)
- **steeringPolicy:** Defines the traffic steering policy for multi-connectivity
  - **method:** Traffic-steering method. Possible values are “minRTT” to prioritize access with the lowest RTT, and “load-balancing” to distribute traffic according to the split ratio
  - **splitRatio:** traffic distribution (access percentages)
    - **3GPP**: percentage being sent over 3GPP
    - **N3GPP**: percentage being sent over non-3GPP
- **duration:** Duration of the session in seconds

Response fields

- **sessionID:** Unique identifier of the instantiated ATSSS session
- **serviceID:** Identifier of the requesting vertical service
- **status:** Status of the session. Possible values: accepted, rejected, deleted

Table 20 – Example for POST atsss-t/session.

POST atsss-t/session

```
{
  "serviceID": "dummyService",
  "flowFilter": {
    "aueID": "dummyAUE",
    "aneID": "dummyANE",
    "protocol": "UDP",
  },
  "steeringPolicy": {
    "method": "load-balancing",
    "splitRatio": {
      "3GPP": 30,
      "N3GPP": 70,
    }
  },
  "duration": 86400,
}
```

Response: 201 Created

```
{
  "sessionID": "dummySession",
  "serviceID": "dummyService",
```

```
  "status": "accepted"
}
```

GET atsss-t/session/{sessionID}

No request fields

Response fields

- **sessionID**: Unique identifier of the instantiated ATSSS session
- **serviceID**: Identifier of the requesting vertical service.
- **startTime**: Timestamp indicating when the session started
- **status**: Status of the session. Possible values: active, failed, ongoing
- **accessNetworks**: list of access networks involved in this session
    - **accessId**: Unique identifier for the access network connection
    - **accessType**: Numeric value (0 to 255) corresponding to specific type of access network connection according to ETSI MEC 015 MtsCapabilityInfo:
        - 0 = Unknown.
        - 1 = Any IEEE802.11-based WLAN technology.
        - 2 = Any 3GPP-based Cellular technology.
        - 3 = Any Fixed Access.
        - 11 = IEEE802.11 a/b/g WLAN.
        - 12 = IEEE 802.11 a/b/g/n WLAN.
        - 13 = IEEE 802.11 a/b/g/n/ac WLAN.
        - 14 = IEEE 802.11 a/b/g/n/ac/ax WLAN (Wi-Fi 6).
        - 15 = IEEE 802.11 b/g/n WLAN.
        - 31 = 3GPP GERAN/UTRA (2G/3G).
        - 32 = 3GPP E-UTRA (4G/LTE).
        - 33 = 3GPP NR (5G).
- **steeringPolicy**: Defines the traffic steering policy for multi-connectivity
    - **method**: Active Steering method, can be “minRTT” or “load-balancing”
    - **splitRatio**: Traffic distribution (access percentages)
        - 3GPP: percentage being sent over 3GPP
        - N3GPP: percentage being sent over non-3GPP
- **duration**: Duration of the session in seconds

Table 21 – Example for GET atsss-t/session/{sessionID}.

GET atsss-t/session/dummySession

Response: 200 OK

```
{
  "sessionID": "dummySession",
  "serviceID": "dummyService",
  "startTime": 2025-12-24T18:02:06.123456,
  "status": "active",
  "accessNetworks": [
    {
      "accessId": "wifi6",
      "accessType": 14,
    },
    {
      "accessId": "5g",
```

```
      "accessType": 33,
    }
  ],
  "steeringPolicy": {
    "method": "load-balancing",
    "splitRatio": {
      "3GPP": 30,
      "N3GPP": 70,
    }
  },
  "duration": 86400,
}
```

DELETE atsss-t/session/{sessionID}

No request fields

Response fields are identical to the response of the session creation (see POST atsss-t/session)

Table 22 – Example for DELETE atsss-t/session/{sessionID}.

DELETE atsss-t/session/dummySession
Response: 200 OK

```
{
  "sessionID": "dummySession",
  "serviceID": "dummyService",
  "status": "deleted"
}
```

4.2.6 Error codes and error handling

The error codes are implemented as specified in the ATSSS(-T) specification document available on the ENVELOPE website.[3] Note that the “status” field of a non-error response for a valid request may also indicate that a session could not be established.

4.3 ATSSS-Cu and ATSSS-Cp APIs

4.3.1 Short description

The ATSSS-Cu and ATSSS-Cp APIs are internal and not exposed to the vertical service. They are responsible for forwarding traffic steering rules to the MPQUIC services residing at the UE (AUE, via ATSSS-Cu) and the core network (ANE, via ATSSS-Cp). If the “load-balancing” traffic steering rule has been selected, ATSSS-Cu further allows the AUE to update the traffic ratio. The traffic ratio determines the traffic split between 3GPP and N3GPP networks for both up- and downlink.

4.3.2 Relation with standards

There is no known related standard. MTS of ETSI MEC 015 is intended for managing multi-connectivity sessions on a single host and does not model communication with the UE. The APF can be interpreted as a specific MTS service that exposes traffic flow management functionality to the UE via the ATSSS-Cu API.

[3] https://envelope-project.eu/wp-content/uploads/2025/11/API-specification-ATSSS.pdf

4.3.3 Functional overview

ATSSS-Cu and ATSSS-Cp are responsible for forwarding the active rule of the APF to the corresponding MPQUIC services running on the AUE and ANE. Figure 9 and Figure 10 illustrate the interaction between the components.

API name	Description	Modules Involved	Use Cases
ATSSS-Cu	Enables an AUE to retrieve and manipulate multi-connectivity traffic steering rules.	APF, AUE	GR-UC-1
ATSSS-Cp	Enables an ANE to retrieve multi connectivity traffic steering rules.	APF, ANE	GR-UC1

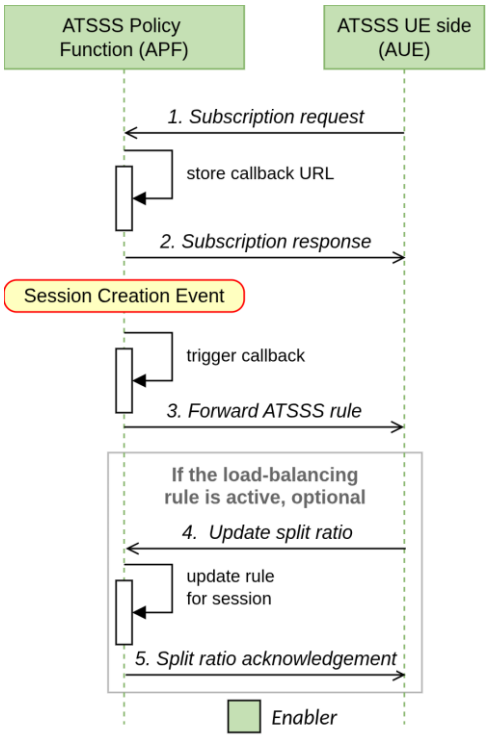

Figure 9 – Message flow for the internal ATSSS-Cu API between APF and AUE.

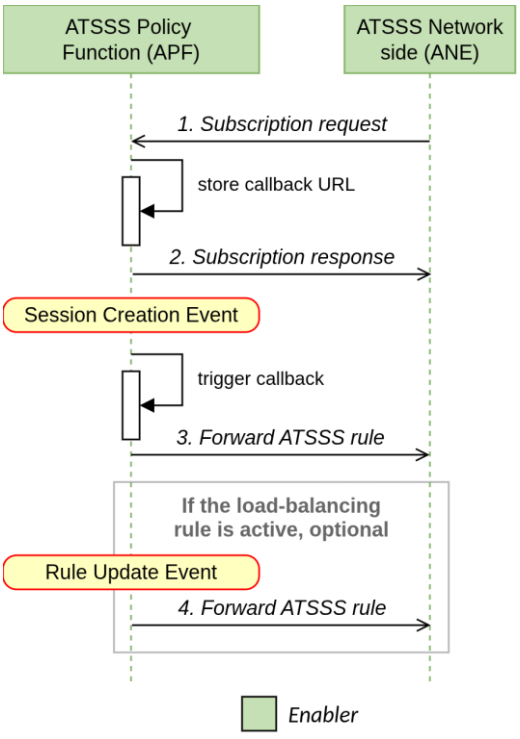

Figure 10 – Message flow for the internal ATSSS-Cp API between APF and ANE.

Both APIs provide functionality for the respective MPQUIC instances to establish a direct session and retrieve the active ATSSS rule. First, the AUE and ANE send a subscription request to the APF (see steps 1 and 2 in Figure 9 and Figure 10). The APF registers both MPQUIC service instances and stores their callback URLs internally. Upon a session creation request by the CAM app server, the APF checks whether the requested AUE and ANE have registered. If that is the case, the session can be created and the APF triggers the callbacks to forward the ATSSS rule to the AUE and ANE (step 3 in Figure 9 and Figure 10).

If the load-balancing rule is active, the ATSSS-Cu API further allows the MPQUIC instance on the UE side (AUE) to set the traffic split ratio (step 4 in Figure 9), which is acknowledged by the APF

(step 5 in Figure 9). A successful update of the traffic split ratio triggers a rule update event, resulting in the APF forwarding the updated rule to the ANE (step 4 in Figure 10).

4.3.4 Endpoints

ATSSS-Cu and ATSSS-Cp expose three endpoints on the APF, as summarized in Table 23.

Table 23 – Overview of ATSSS-Cu and ATSSS-Cp endpoints.

Resource name	Resource URI	HTTP method	Description
ATSSS-Cu	atsss-cu/sub	POST	Allows an AUE to subscribe to ATSSS rules updates for a service
ATSSS-Cu	atsss-cu/ratio/{aueID}	POST	Update traffic ratio in “load-balancing” mode
ATSSS-Cp	atsss-cp/sub	POST	Allows an ANE to subscribe to ATSSS rules updates for a service

4.3.5 Request and response examples

POST atsss-cu/sub

Request fields

- **aueID**: a unique identifier of the requesting AUE
- **serviceID**: the identifier of the service for which this AUE provides ATSSS functionality
- **callback**: a callback URL that will be used to notify the AUE about session creation

The direct response by the APF only notifies the AUE about the successful subscription.

The asynchronous callback response comprises the fields **sessionID**, **serviceID**, **startTime**, **status**, **steeringPolicy**, and **duration** analogous to the atsss-t/session/{sessionID} response fields of Section 4.2.5). In addition, the **aneID** field provides an identifier of the ANE that participates in the session, allowing the AUE to establish and maintain a MPQUIC connection with the ANE. If the AUE is involved in multiple sessions, the sessions details are provided in a list. If a session has been closed, the session is removed from the list. If the AUE is involved in no session or the previous session has been deleted, the response is empty. The APF may substitute the real sessionID and provide the AUE with a unique sessionID that only itself can map to the sessionID known to the CAM app server (that is used to manage the session).

Table 24 – Example for GET atsss-cu/sub.

AUE ➔ APF: POST atsss-cu/sub

```
{
  "aueID": "dummyAUE",
  "serviceID": "dummyService",
  "callback": "dummyAUE/callback",
}
```

Response: 201 Created
APF ➔ AUE: POST dummyAUE/callback (asynchronous callback)

```
{
  "sessionID": "dummySession",
  "serviceID": "dummyService",
  "startTime": 2025-12-24T18:02:06.123456,
  "status": "active",
  "steeringPolicy": {
    "method": "load-balancing",
    "splitRatio": {
      "3GPP": 30,
      "N3GPP": 70,
    }
  },
  "duration": 86400,
  "aneID": "dummyANE",
}
```

POST atsss-cu/ratio/{aueID}

Request fields

- **ratio**: the percentage of traffic that should be sent over the 3GPP network path

There are no additional response fields. A successful request is acknowledged.

Table 25 – Example for POST atsss-cu/ratio/{aueID}

POST atsss-cu/ratio/dummyAUE
{ "ratio": 45 }
Response: 200 OK

POST atsss-cp/sub

This endpoint behaves almost identical to atsss-cu/sub. The only differences are that the request does not contain the aueID field but an **aneID** field (its own identifier) and that the asynchronous response of the APF does not contain the aneID field but an **aueID** field (the identifier of the AUE that is part of the respective active session).

4.3.6 Error codes and error handling

The error codes of ATSSS-Cu and ATSSS-Cp are identical to the error codes of ATSSS-T.

4.4 Implementation Details

This section provides an overview of the implementation details of the ATSSS Policy Function, PMF, MPQUIC, and the RL agent. To enable flexible deployment, all software components are encapsulated in docker containers.

4.4.1 ATSSS Policy Function

The ATSSS Policy Function (APF) coordinates the interaction between all components involved in the multi-connectivity session. It hosts the three ATSSS APIs, namely ATSSS-T, ATSSS-Cu and ATSSS-Cp. The APF is implemented using Python and the FastAPI framework.[4]

The APF maintains a dynamic registry of all ATSSS-capable UE (AUE) and Core Network (ANE) systems. Each AUE and ANE registers with the APF when it becomes available and de-registers (or is removed by timeout) when it is no longer reachable. For each registered entity, the APF stores at least the following information:

- Unique identifier (AUE ID or ANE ID)
- Network addresses of the UE / Core

This registry is used by the APF to validate the feasibility of ATSSS session requests and to map a vertical service session to a concrete pair of AUE and ANE.

When the CAM Application Server wants to start an ATSSS session with some particular UE, it first sends a session establishment request via ATSSS-T that includes the selected endpoints and the desired steering policy (e.g., minRTT or load-balancing).

Based on its internal state, the APF decides whether the request can be fulfilled (i.e., when the respective AUE and ANE are registered and thus support ATSSS multi-connectivity). If this is the case, the APF creates a new session and notifies the AUE and ANE involved in this connection.

Multipath connectivity is achieved with MPQUIC, which manages two network paths between the AUE residing at UE and the ANE residing in the core network. The ANE is connected bidirectionally with the vertical service.

Once the CAM Application Server starts a session, the AUE receives a session-start trigger together with the required configuration (ANE address, ATSSS policy, etc.) via ATSSS-Cu. If the chosen policy is load-balancing, the AUE invokes the RL agent to compute the traffic split ratio that determines how packets are distributed across the available paths. If the chosen policy is minRTT, the AUE obtains path metrics for both accesses (e.g., from the PMF) and selects the path with the lowest RTT for transmission. For all policies, scheduling decisions are taken at a fine granularity and can be adapted frequently based on the current network conditions (i.e., more dynamic networks lead to more frequent decision-making).

Similarly, the ANE is notified about the start of the session via ATSSS-Cp. For uplink traffic, its main role is to collect packets received from the AUE over both paths, reassemble them into a single stream, and forward this stream to the CAM Application Server. For downlink traffic, it places packets on the paths according to the scheduling decisions taken at the AUE. To this end, the ANE continuously receives the AUE's current scheduling configuration from the APF via ATSSS-Cp, ensuring consistent steering behavior in both uplink and downlink.

[4] https://fastapi.tiangolo.com/

Multipath QUIC in ENVELOPE is implemented using a lightweight C library developed in the project that wraps PicoQUIC[5] and exposes a simplified client-server API. The library uses an internal packet queue. When data is enqueued, a dedicated network thread configures the send context, copies the data into a PicoQUIC buffer, and hands it to PicoQUIC's scheduler, which processes it via its callback-based, event-driven API. It supports both IETF MP-QUIC (using PicoQUIC's native multipath extensions) and a custom multipath mode based on multiple coordinated PicoQUIC instances.

4.4.2 PMF

Path scheduling is crucial over heterogeneous links, often suffering head-of-line (HoL) blocking. As wireless links can vary rapidly, and outages on one path may last several seconds, packets on a slower path often lag behind those on a faster path, leading to out-of-order arrivals that degrade end users' QoE. Therefore, real-time monitoring of each access link's performance is of prior importance for effective traffic steering, switching and splitting. 3GPP introduces the PMF which provides per-path monitoring by continuously measuring metrics such as RTT, packet loss and access availability/unavailability support [3GPP TS 23.501 - Section 5.32.5].

In the context of the ENVELOPE, we provide a PoC implementation of the PMF procedures to support RTT and packet loss Packet Loss Rate (PLR). Specifically, the UE-initiated and UPF-initiated RTT procedures will allow both endpoints to continuously evaluate path latency which is essential for delay-sensitive steering modes such as minRTT. Similarly, the UE-initiated and network-initiated PLR measurement procedures are developed to track uplink and downlink packet loss, which may be used to determine path congestion conditions.

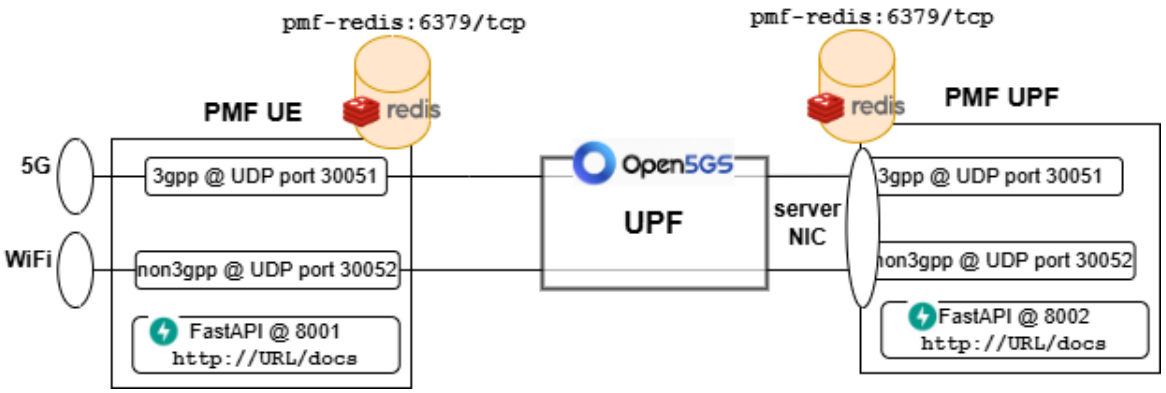

Figure 11. PoC architecture of PMF

Figure 11 illustrates the PoC architecture used to implement the 3GPP PMF logical components and the related procedures. The architecture is lightweight and modular based on docker containers, allowing PMF logic to be integrated alongside any existing 5G core without intrusive changes to the user-plane data path. At a high level, the architecture consists of two main components: PMF UE and PMF UPF, interconnected over both 3GPP and non-3GPP access networks, through the PDU sessions over UPF.

[5] https://github.com/private-octopus/picoquic

The PMF UE component represents the user-side PMF logic responsible for initiating and responding to per-path monitoring procedures. It is collocated with the UE and maintains independent communication channels for each access type:

- 3GPP access: UDP port 30051
- non-3GPP access (e.g., Wi-Fi): UDP port 30052

These UDP channels are used to exchange PMF probe packets for RTT and packet loss measurements. Measurement results are locally stored and exposed via a FastAPI service (port 8001), enabling observability, debugging, and integration with higher-layer experimentation logic. To support stateful and continuous monitoring, the PMF UE leverages a Redis datastore, which maintains per-path statistics such as RTT samples, packet counters, and loss measurements.

Likewise, the PMF UPF component implements the network-side PMF logic and is logically collocated with the UPF (i.e., a separate container attached to the same docker network of UPF). It supports network-initiated procedures and listens on UDP ports 30051 and 30052 for PMFP messages incoming from the UE. This symmetry allows RTT and PLR measurements to be performed independently in the uplink and downlink directions, in line with 3GPP specifications. The PMF UPF maintains its own Redis instance to store measurement state and historical metrics, enabling correlation with UE-side observations. A FastAPI service (port 8002) exposes PMF statistics and control endpoints.

This architecture enables continuous, bidirectional per-path monitoring with minimal overhead (e.g., UDP payloads up to 16 bytes), providing the real-time visibility required to mitigate HoL blocking and support adaptive traffic steering over heterogeneous access networks.

4.4.3 Reinforcement Learning Agent

The Reinforcement Learning (RL) agent is implemented in Python using the RLlib library.[6] The library provides scalable implementations of state-of-the-art deep RL algorithms and models, as well as sensible default configurations.

The goal of the ATSSS RL agent is to adapt the traffic ratio depending on the current network conditions with the aim of improving the network's performance in comparison to fixed schedulers. The concrete target metrics for the agent are defined by the concrete CAM application. As an initial feasibility check, we decided to minimize the weighted round trip time while penalizing ratio changes for increased stability.

To adapt its decisions to the current network conditions, the agent receives data from the Performance Measurement Function (PMF) about each network path, including the current round-trip times and packet loss rates.

We selected the Proximal Policy Optimization (PPO) algorithm to train the agent. While PPO is a well-established deep RL algorithm, RLlib allows experimenting with different training algorithms with little effort. The agent receives the output of the PMF as an observation and outputs a continuous action value in the interval [0, 1]. This action is the splitting ratio that is sent to the APF via ATSSS-Cu. An intelligent agent should be able to not only react to the most recent measurements, but also to detect trends and anticipate changes in the network. To enable such

[6] https://docs.ray.io/en/latest/rllib/index.html

functionality, the agent can leverage a Long Short-Term Memory (LSTM) cell to aggregate information about previously received observations within its internal state. However, this approach requires processing all observations sequentially, which could lead to a high overhead. A stateless alternative would be to use a finite history of previous observations as input to the model. However, this could increase the costs per invocation of the model.

The plan is to leverage the flexibility of RLlib and investigate the effect of different model architectures and training algorithms on the agent's performance in concrete CAM scenarios.

4.5 Lessons Learned

The specification and development of the ATSSS APIs were an iterative process. We started with an initial definition of ATSSS-T and hard-coded information like the IP addresses of the MPQUIC service instances at the UE and the core network. This allowed us to quickly demonstrate the technical feasibility, but initially we had not defined a common procedure of how such information is exchanged between the individual ATSSS components. Initial implementations of ATSSS-Cu and ATSSS-Cp were synchronous, but we eventually selected an asynchronous design for improved efficiency. As the RL agent depends on the PMF, we first created a dummy PMF that utilizes regular ping commands to estimate the latency and forward this information to the agent. Retrospectively, we think this was a good approach, as it allowed us to work on the RL part independent of the other ATSSS components.

5 Edge Computing

5.1 Overview and Innovations

The edge computing solutions developed in the project have been designed to enable seamless management and orchestration of applications across distributed telco edge cloud environments. At the heart of the developed solutions is the CAMARA Edge Application Management API. This API abstracts the complexity of heterogeneous edge infrastructures, allowing for the discovery of available edge zones, onboarding and instantiation of applications, and dynamic migration (handover) of services as users or devices move geographically. The system supports not only the deployment and lifecycle management of applications but also their federation across multiple telco domains, ensuring interoperability and resource sharing between different operators and platforms. Another API developed is the Traffic Influence API, which can be used to optimize the UE data traffic to the most appropriate edge application instance in the edge cloud infrastructure.

These solutions follow an architecture which is modular and cloud-native, thereby leveraging containerized microservices and adapters that translate the CAMARA APIs into the native APIs of underlying platforms, namely, ETSI MEC, AerOS, and Liqo Kubernetes-based multi-cluster orchestrators. This northbound abstraction and southbound translation mechanism ensures that application developers interact with a consistent API layer, regardless of the specific technologies deployed at each edge site. The system further incorporates advanced orchestration capabilities, enabling automated, context-aware deployment and migration of applications based on real-time network intelligence and user mobility. This is particularly critical for latency-sensitive and mission-critical services, such as those in connected mobility and automotive domains, where service continuity and performance must be maintained as devices traverse different network zones.

5.2 Edge Application Management API

5.2.1 Short description

CAMARA Edge Application Management APIs enable developers and application providers to manage the full lifecycle of edge applications within a distributed telco edge cloud environment. These APIs support: discovery of available edge zones, onboarding of applications, application instantiation in optimal edge cloud zones based on computing and network requirements, and also the handover (migration) of application across edge zones. Application deployment at the edge ensures low latency, service reliability, and improved performance for advanced services in automotive and other domains, while leveraging the strengths of traditional cloud combined with mobile network operator expertise for operational efficiency and cost savings.

5.2.2 Relation with standards

This specification complies with the CAMARA Edge Application Management specification version v0.9.3-wip (April 2025), see exact git commit for the YAML specification implemented here.

5.2.3 Functional overview

The CAMARA Edge Application Management API abstracts specific implementation choices with respect to the underlying standardisation specifications (e.g., ETSI MEC) and/or open-source solutions (e.g., Kubernetes-based). In this section, the different underlying edge enablers are described including details of how the Edge Application Management API is translated to each enabler's southbound native APIs.

5.2.3.1 ETSI MEC-based enabler

In the Italian site, the native API into which the ENVELOPE APIs are translated are compliant to the ETSI MEC standard. In particular, the following services have been implemented:

- **Edge Application Package Management:** This service handles the lifecycle of MEC application packages that include all necessary files, metadata, and artifacts (e.g., VM or container images) needed to deploy an application on the MEC system. It facilitates onboarding, updating, and removal of application packages to/from the MEC environment, ensuring the applications are ready for instantiation and deployment on edge hosts. An example of how the Edge Enabler interact with the underlying Edge Host via ETSI MEC API is depicted in Figure 12, which describe how an application package is onboarded in the MEC HOST

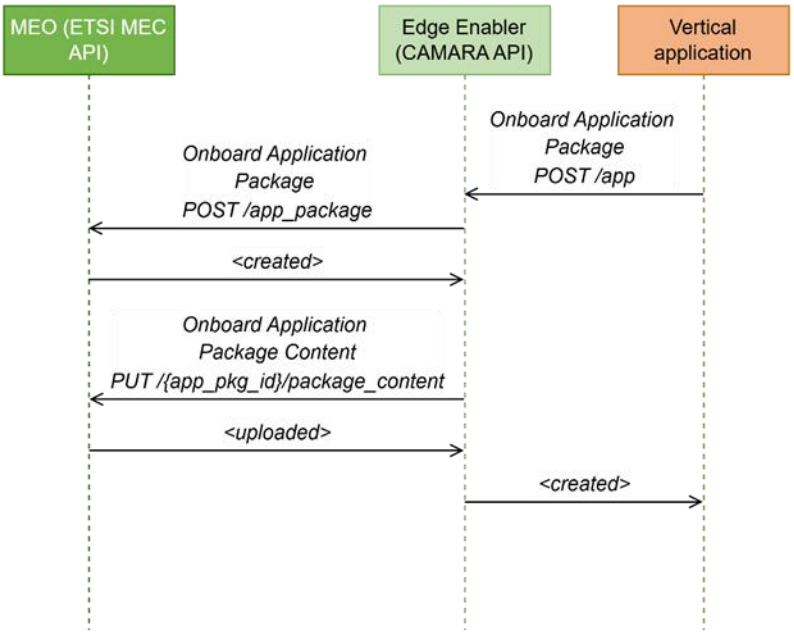

Figure 12 – Sequence diagram to onboard application packages through the Edge Enabler.

The diagram illustrates a key interaction sequence among the Edge Host (using the ETSI MEC API), the Edge Enabler (using the CAMARA API), and the Vertical Application in the context of onboarding an edge application package. The sequence highlights the transformation and bridging of APIs between different components in the ENVELOPE architecture. Initially, the Vertical Application sends a request to onboard an application package to the Edge Enabler via a POST /app call. The Edge Enabler then forwards this request to the Edge Host using the ETSI MEC native API (POST /app_package). Upon successful creation acknowledgment, the Vertical Application

proceeds to upload the application package content through a PUT request that includes the application package ID. This process ensures that the application package content is securely transferred and registered on the Edge Host, encapsulating the complexity of interfacing multiple APIs and abstracting it to support standardized ETSI MEC operations.

- **Edge Application Instance Management**: This service manages the instantiation, running state, termination, and lifecycle events of MEC application instances on edge hosts. It coordinates where and when applications execute within the edge system, optimizing resource use and service availability based on application requirements and context. An example of interaction between the Vertical Application and the Edge Enabler to instantiate an application in a MEC Host is depicted in Figure 13.

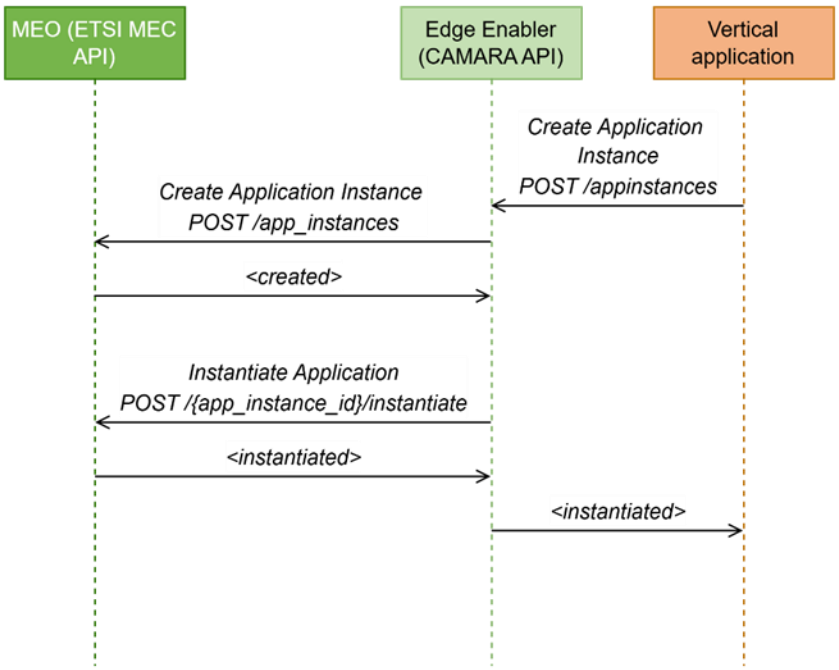

Figure 13 – Interaction between Vertical Application and Edge Enabler to instantiate an application in a MEC Host.

This sequence diagram details the process of creating and instantiating an application instance across three layers: the Vertical Application, the Edge Enabler via the CAMARA API, and the Edge Host conforming to the ETSI MEC API. Initially, the Vertical Application requests the creation of an application instance by sending a POST /app_instances call to the Edge Enabler. The Edge Enabler acts as a mediator and transforms this request into a corresponding POST /app_instances call to the Edge Host, following the native ETSI MEC API. Once the application instance is successfully created, an acknowledgment is returned back through the same chain. Next, the Vertical Application triggers the instantiation of the application instance by sending a POST to the specific instance instantiate endpoint (POST /{app_instance_id}/instantiate) at the Edge Enabler level, which forwards this request to the Edge Host. The Edge Host then proceeds with the actual application instance instantiation, enabling the application to run on the edge environment. Confirmation of instantiation is propagated upstream to notify all involved entities.

- **Federation of Multiple Edge Systems:** This service enables collaboration and interoperability among multiple MEC systems deployed across different locations or administrative domains. Federation allows sharing of resources, services, and application mobility across edge systems, thus extending the reach and capabilities of edge applications beyond a single MEC deployment.

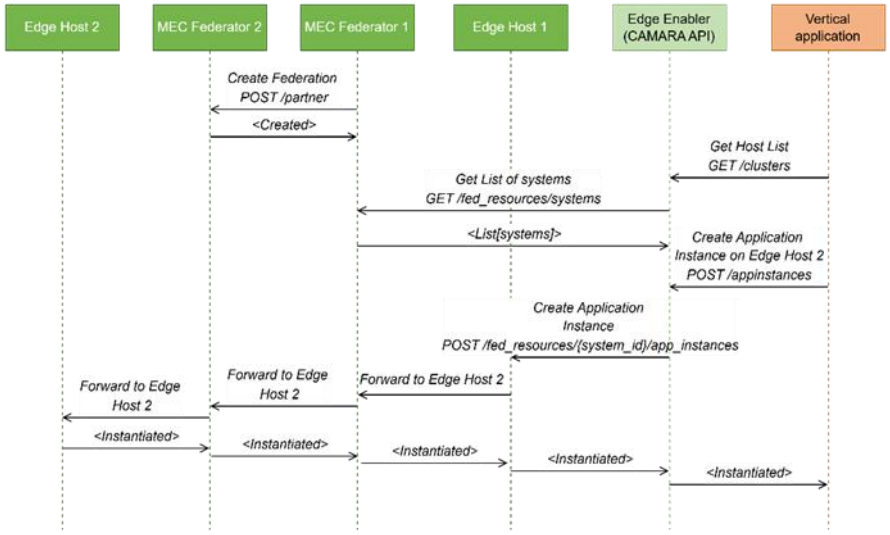

Figure 14 – Interaction between Vertical Application and Edge Enabler to instantiate an application in a MEC Host in a Federated environment.

Figure 14 illustrates the service orchestration involved in federating multiple edge systems and managing distributed application instances. The process begins with the creation of a federation partner via a POST /partner call initiated by MEC Federator 1 to MEC Federator 2, which confirms the federation creation. Next, Edge Host 1 queries the federated system list with GET /fed_resources/systems to discover available edge systems and their resources, sharing the list with the Edge Enabler. The Vertical Application, via the Edge Enabler (CAMARA API), retrieves host clusters information using GET /clusters. Based on this information, it initiates the creation of an application instance on a specific remote Edge Host (Edge Host 2) by posting an application instance creation request to the federated system endpoint (POST /fed_resources/{system_id}/app_instances). This request is forwarded through MEC Federator components and ultimately reaches Edge Host 2. After the instantiation of the application instance on Edge Host 2, the instantiation confirmation propagates back through the federation layers and the Edge Enabler to the Vertical Application.

5.2.3.1.1 Native API Details

This section describes the key native API endpoints grouped by functional areas (Application Package Management, Application Instance Management, Federation, and Application Mobility Services). Each area provides standard RESTful operations supporting lifecycle management such as creation, querying, updating, and deletion of resources.

5.2.3.1.1.1 Application Package Management API

Table 26 lists the endpoints for the Application Management Native API. The API root is always */app-pgkm/v1.*

Table 26 – Overview of Application Package Management API endpoints.

Endpoint name	Resource URI	HTTP method	Description
Query Onboarded App Packages	/onboarded_app_packages	GET	Queries information about onboarded application packages with optional filters and field selections
Create Onboard Application Package Resource	/onboarded_app_packages	POST	Creates a resource for onboarding an application package with necessary metadata.
Fetch Onboarded App Package Content	/onboarded_app_packages/{appPkgId or appDId}/package_content	POST	Retrieves the content of an onboarded application package identified by appPkgId or appDId. Supports partial content range requests.
Upload Application Package Content	/onboarded_app_packages/{appPkgId or appDId}/package_content	PUT	Uploads content for an application package identified by appPkgId or appDId. Accepts multipart/form-data with package file.
Get Individual App Package Details	/onboarded_app_packages/{appPkgId}	GET	Retrieves detailed information of a specific app package resource by its ID.
Delete Individual App Package	/onboarded_app_packages/{appPkgId}	DELETE	Deletes a specific app package resource from the system by its ID.
Update Operational State of App Package	/onboarded_app_packages/{appPkgId}	PATCH	Modifies the operational state of an individual app package resource.
Read AppD Content of Onboarded Package	/onboarded_app_packages/{appDid}/appD	GET	Retrieves the application descriptor content of an onboarded package by appDId or appPkgId, with options for filtering and attribute inclusion/exclusion.

5.2.3.1.1.2 Application Instance Management API

Table 27 lists the endpoint for the Application Instance Native API. The API root is always /app_lcm/v1.

Table 27 – Overview of Application Instance Management API endpoints.

Endpoint name	Resource URI	HTTP method	Description
Query Application Instances	/app_instances	GET	Queries information relating to onboarded application instances in the MEO.
Create Application Instance	/app_instances	POST	Creates an application instance resource
Terminate Application Instance	/app_instances/{appInstanceId}/terminate	POST	Terminates an application instance identified by appInstanceId (UUID).
Change Operational State of Application Instance	/app_instances/{appInstanceId}/operate	POST	Changes the operational state (start/stop) of an application instance identified by appInstanceId.
Instantiate Application Instance	/app_instances/{appInstanceId}/instantiate	POST	Instantiates (creates and starts) an application instance identified by appInstanceId.
Delete Application Instance	/app_instances/{appInstanceId}	DELETE	Deletes an individual application instance resource identified by appInstanceId.
Terminate Service Instance	/app_instances/{appInstanceId}	POST	Terminates a service instance identified by serviceInstanceId. Optional boolean query parameter to force termination.

5.2.3.1.1.3 MEC Federation

Table 28 lists the endpoint for the MEC Federation Native API.

Table 28 – Overview of MEC Federation API endpoints.

Endpoint name	Resource URI	HTTP method	Description
Create Federation	/partener	POST	Creates a one-direction federation with another operator platform using a FederationRequest. On success,

			returns federation meta-information and headers indicating the new resource location. Supports a callback (onPartnerStatusEvent) for notifying the partner of changes in federation status, metadata, zones, endpoints, network codes, and service APIs.
Get Federation Details	/{federationContextId}/partner	GET	Retrieves details about federation context with the partner operator platform, including offered zones, network codes, edge discovery, LCM service, and platform capabilities.
Update Federation	/{federationContextId}/partner	PATCH	Updates parameters associated with an existing federation (such as mobile/fixed network codes, OPS or APP policies).
Delete Federation	/{federationContextId}/partner	DELETE	Removes an existing federation context by federationContextId. Returns success (200) if deleted, or error responses similar to above
Get Federation Context ID	/fed-context-id	GET	Retrieves the existing federationContextId with the partner operator platform. Returns federationContextId (and Location header).
Get Federation Health	/{federationContextId}/health	GET	Gets health status info for the federation context with the partner operator platform.
Renew Federation	/{federationContextId}/renew	POST	Renews the existing federation context by federationContextId. Returns updated FederationContextId, federationRenewalDate, and federationExpiryDate.

5.2.3.2 AerOS-based continuum enabler

5.2.3.2.1 Short description

AerOS is an edge–cloud continuum platform developed under the aerOS project[7] that acts as a meta-operating system to intelligently manage and integrate distributed computing and network resources into a seamless computing continuum. It enables the federation and abstraction of geographically dispersed resources, allowing dynamic service allocation upon request, while exposing standardized NGSI-LD–based APIs for querying resource availability and orchestrating hyper-distributed containerized applications across edge and cloud. The aerOS APIs operate on the southbound interface of the ENVELOPE platform, enabling external components and applications to interact with aerOS for querying resource availability, initiating service deployments and managing orchestration lifecycles throughout the distributed infrastructure. Within the Greek site, the aerOS APIs are utilized to support 5G inter-PLMN mobility scenarios by orchestrating service migration between two administrative domains that form part of a federated aerOS continuum. While the UE handover between PLMNs is performed at the 5G network layer, aerOS complements this process by ensuring application-level continuity. This is achieved by redeploying the relevant services in the target aerOS domain following the UE's network transition. Both domains expose their compute and network resources through the aerOS federation layer, enabling unified management and transparent external access via standardized APIs. In this context, the aerOS enabler reacts to mobility events by initiating dynamic service redeployment based on the UE's current PLMN, thereby ensuring context-aware orchestration and seamless service continuity across the continuum.

5.2.3.2.2 Relation with standards

This specification complies with the ETSI NGSI-LD standard (version 1.6.1) for context information management and semantic interoperability. The aerOS APIs adopt TOSCA Simple YAML 1.3 for service descriptor specification and NGSI-LD query expressions follow the ETSI GS CIM 009 specification for flexible and standardized resource discovery across the continuum infrastructure.

5.2.3.2.3 Functional overview

The following sequence diagrams illustrate the key interaction flows within the aerOS platform, focusing on secure access, service orchestration and resource discovery across the edge–cloud continuum. aerOS employs a federated architecture where each domain exposes its capabilities through a secure API gateway, validates user credentials via a shared identity management system, and coordinates service deployments through a multi-layer orchestration stack. The diagrams capture the main operational stages: authentication through Keycloak, request mediation by the API Gateway, high-level orchestration using TOSCA descriptors, distributed allocation across local and remote domains, and retrieval of federated resource information through Orion-LD. Collectively, they provide a clear view of how aerOS processes user requests end-to-end and how the various components collaborate to deliver secure, cross-domain continuum services.

[7] https://aeros-project.eu/

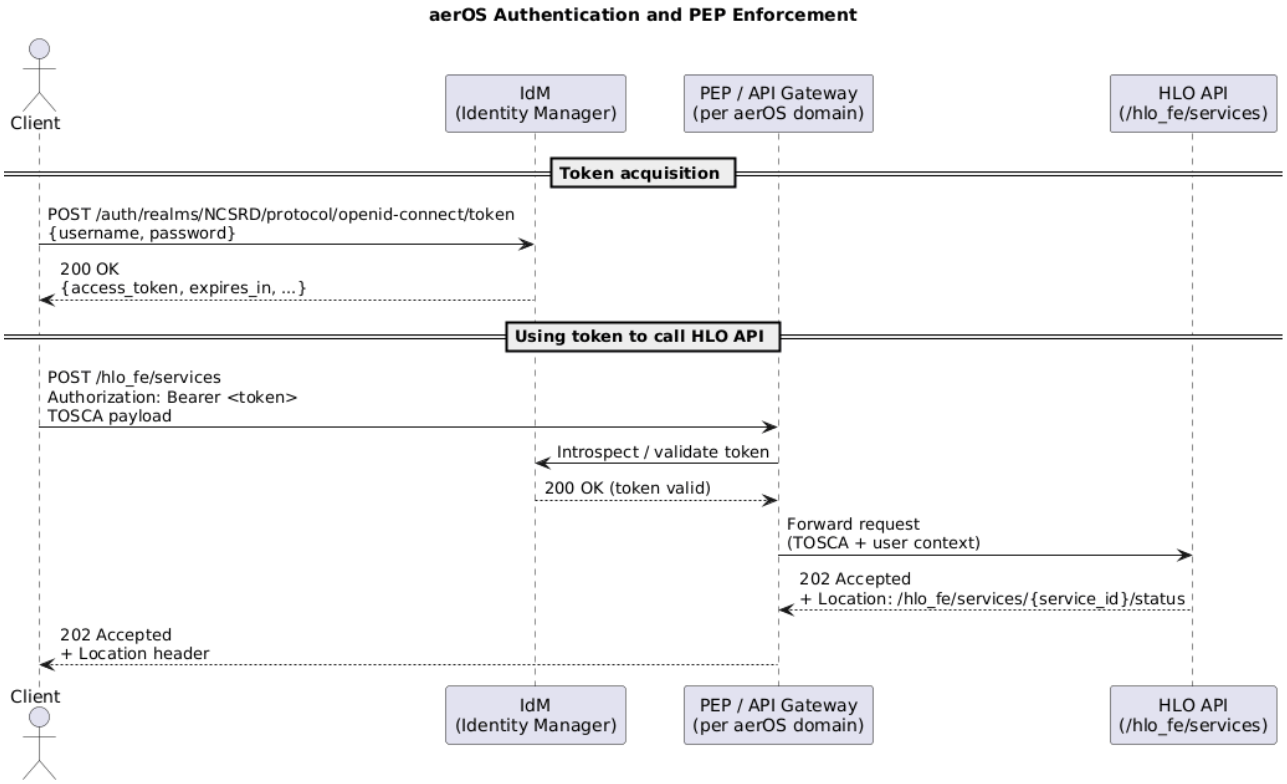

Figure 15 – Sequence diagram of the aerOS authentication and PEP enforcement.

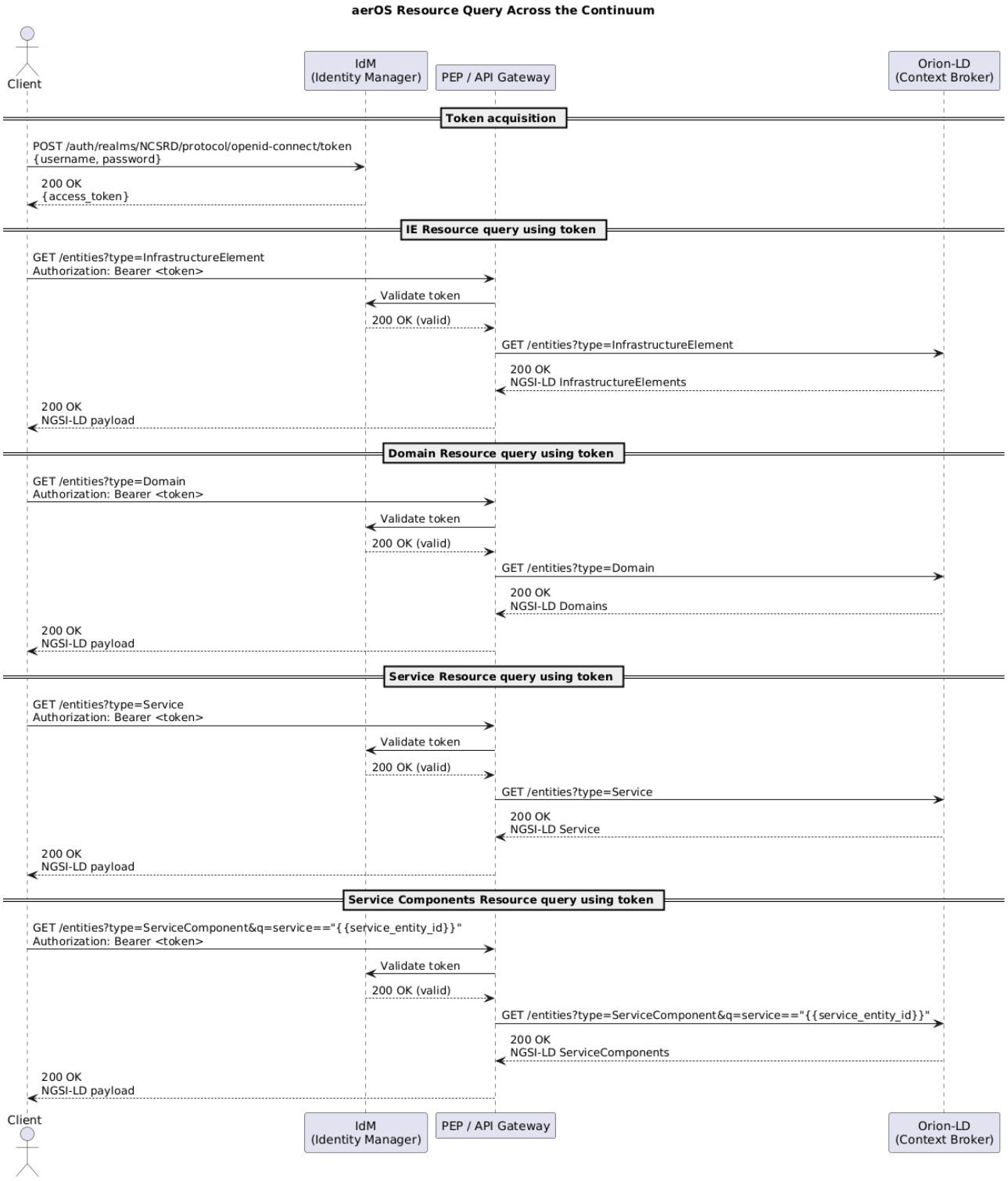

Figure 16 – Sequence diagram of aerOS resource query across the continuum.

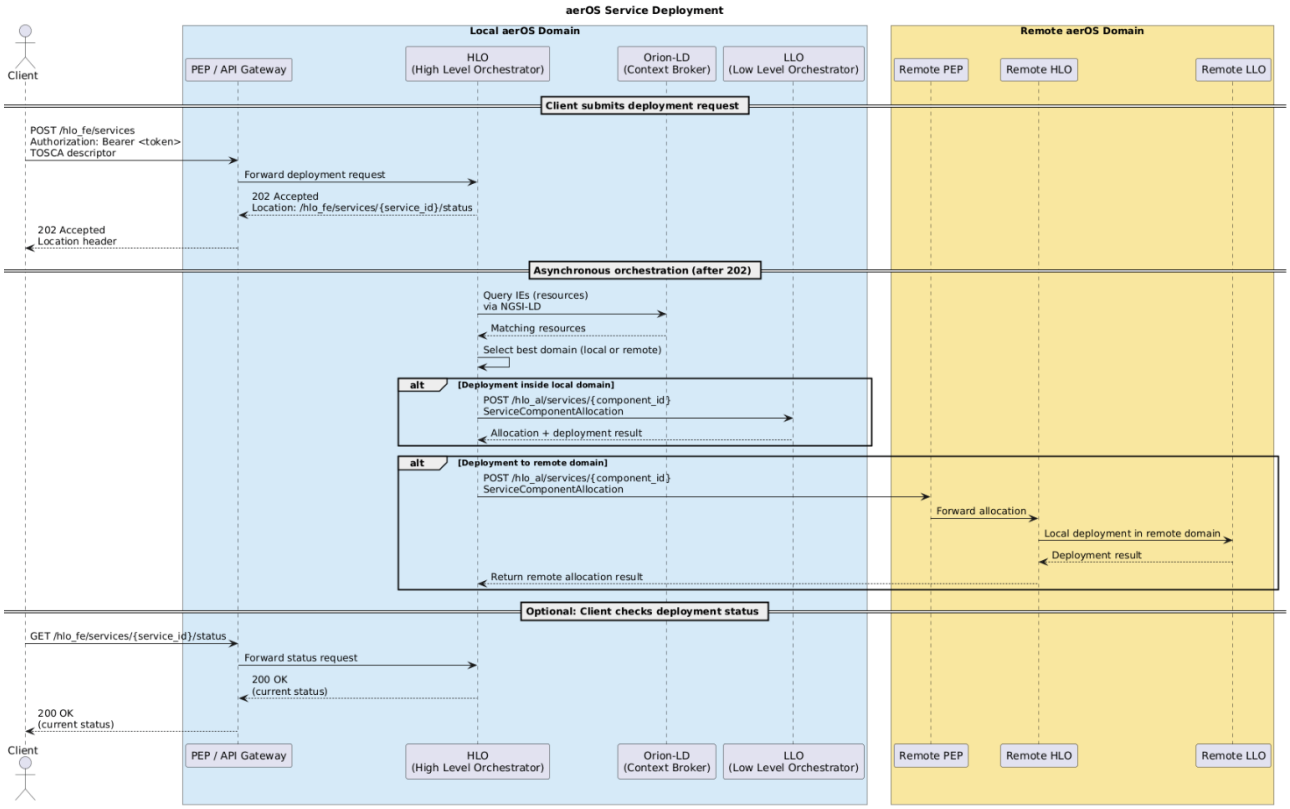

Figure 17 – Sequence diagram of aerOS service deployment.

5.2.3.2.3.1 Onboarding and Instantiation Workflow

In CAMARA, onboarding and deployment are distinct steps: /apps endpoints register application definition while /appinstances endpoints manage actual execution. aerOS, by contrast, does not distinguish onboarding from instantiation; both are implicitly combined in the creation of a service exposed in the /hlo_fe/services/{service_id} endpoint. To reconcile this mismatch, the CAMARA adapter implements internal storage for onboarded manifests, assigning each a CAMARA-compliant AppId. No call is made to aerOS during onboarding. When a deployment request arrives the adapter retrieves the manifest, from local store, gets also CAMARA deployment information, concerning selected zone for deployment, and generates a TOSCA descriptor which carries all information required for an aerOS based deployment. Accordingly, this TOSCA descriptor is submitted for allocation over aerOS-operated continuum via the /hlo_fe/services/{service_id} API endpoint. Additionally, adapter generates a CAMARA AppDeploymentId (or AppInstanceId), mapping the aerOS service identifier to CAMARA's lifecycle model. In this way, CAMARA's two-phase workflow is maintained even though aerOS provides only a single orchestration entry point.

Below we demonstrate how an application described through the CAMARA AppManifest is transformed into a TOSCA service descriptor ready for aerOS deployment. The following CAMARA manifest captures a simple containerized application based on nginx:


```
{
  "appId": null,
  "name": "NginxApplication",
  "appProvider": "P_123e4567_e89b_12d3_a456_426614174000",
  "version": "1.0.0",
  "packageType": "CONTAINER",
  "operatingSystem": null,
  "appRepo": {
    "type": "PUBLICREPO",
    "imagePath": "docker.io/library/nginx:latest",
    "userName": null,
    "credentials": null,
    "authType": null,
    "checksum": null
  },
  "requiredResources": {
    "infraKind": "kubernetes",
    "applicationResources": {
      "cpuPool": {
        "numCPU": 1,
        "memory": 1024,
        "topology": {
          "minNumberOfNodes": 1,
          "minNodeCpu": 1,
          "minNodeMemory": 1024
        }
      },
      "gpuPool": null
    },
    "isStandalone": true,
    "version": null,
    "additionalStorage": null,
    "networking": null,
    "addons": null
  },
  "componentSpec": [
    {
      "componentName": "Nginx_VDU",
      "networkInterfaces": [
        {
          "interfaceId": "nginx_port",
          "protocol": "TCP",
          "port": 80,
          "visibilityType": "VISIBILITY_INTERNAL"
        }
      ]
    }
  ]
}
```


When this manifest is submitted for deployment, the transformation layer first derives a TOSCA node template from the defined component. The name Nginx_VDU becomes the TOSCA node name, and the type defaults to a container-oriented node such as tosca.nodes.Container.Application.

```
node_templates:
  Nginx_VDU:
    type: tosca.nodes.Container.Application
```

The container image docker.io/library/nginx:latest is dissected into its repository (docker.io/library) and image file (nginx:latest), forming a TOSCA artifact. Because the manifest declares the image source as PUBLICREPO, the artifact is marked as non-private.

```
artifacts:
  application_image:
    file: nginx:latest
    type: tosca.artifacts.Deployment.Image.Container.Docker
    repository: docker.io/library
    is_private: false
```

Network interface definitions, such as the internal TCP endpoint on port 80, are converted into TOSCA network requirements, including port definitions and visibility rules.

```
requirements:
  - network:
      properties:
        ports:
          nginx_port:
            properties:
              protocol: [tcp]
              source: 80
        exposePorts: false
```

Resource expectations in the manifest, notably memory and CPU requirements, are rewritten as host capability filters using TOSCA's node_filter mechanism. The deployment zone chosen during instantiation is inserted as a domain_id filter, ensuring that the service is orchestrated within the intended aerOS domain.

```
- host:
    node_filter:
      capabilities:
        - host:
            properties:
              cpu_usage:
                less_or_equal: 0.4
              cpu_arch:
                equal: x64
              mem_size:
                greater_or_equal: "1024"
              realtime:
                equal: false
              energy_efficiency:
                greater_or_equal: "0"
              green:
                greater_or_equal: "0"
              domain_id:
                equal: urn:ngsi-ld:Domain:ncsrd01
```

Finally, the adapter attaches a TOSCA interface block describing how the service should be created. aerOS expects lifecycle controls under the Standard interface, including inputs, such as cliArgs and envVars. These fields do not originate from the CAMARA manifest; they are provided because aerOS supports them.

```
interfaces:
  Standard:
    create:
      implementation: application_image
      inputs:
        cliArgs: []
        envVars: []
```

The complete result is a TOSCA document resembling the following:

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3
description: TOSCA for NginxApplication
serviceOverlay: false
node_templates:
  Nginx_VDU:
    type: tosca.nodes.Container.Application
    requirements:
    - network:
        properties:
          ports:
            nginx_port:
              properties:
                protocol:
                - tcp
                source: 80
          exposePorts: false
    - host:
        node_filter:
          capabilities:
          - host:
              properties:
                cpu_usage:
                  less_or_equal: 0.4
                cpu_arch:
                  equal: x64
                mem_size:
                  greater_or_equal: '1024'
                realtime:
                  equal: false
                energy_efficiency:
                  greater_or_equal: '0'
                green:
                  greater_or_equal: '0'
                domain_id:
                  equal: urn:ngsi-ld:Domain:ncsrd01
    artifacts:
      application_image:
        file: nginx:latest
        type: tosca.artifacts.Deployment.Image.Container.Docker
        repository: docker.io/library
        is_private: false
    interfaces:
      Standard:
        create:
          implementation: application_image
          inputs:
            cliArgs: []
            envVars: []
    isJob: false
```

This example demonstrates the essential logic of the transformation: CAMARA's application-centric and compact representation is expanded into a deployment-ready TOSCA template suitable for aerOS's orchestrator, while key semantics from the CAMARA manifest, component identity, image source, network ports, and resource requirements, are preserved.

The operational flow underlying onboarding and instantiation can be illustrated through the following sequence diagram depicted in Figure 18:

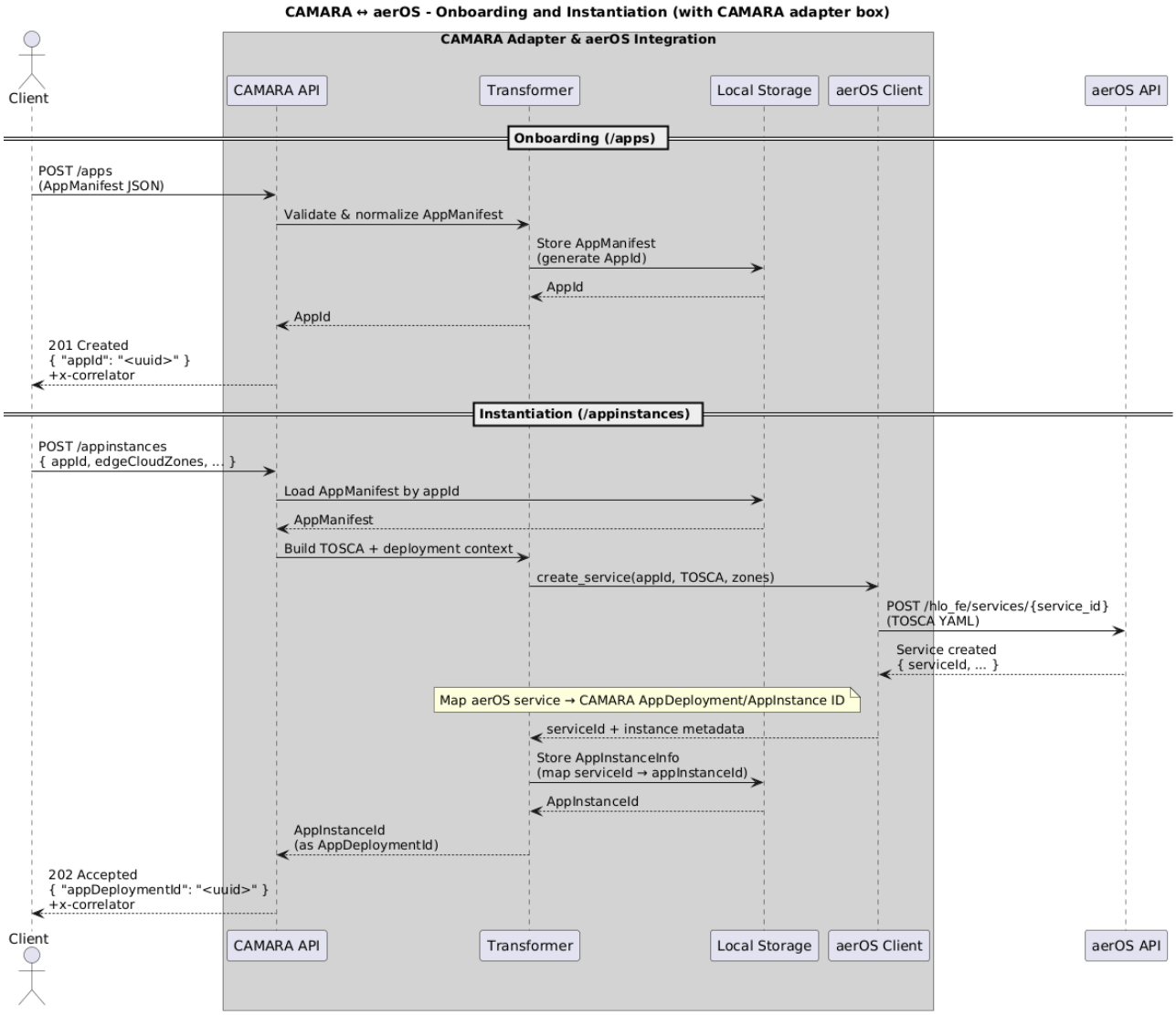

Figure 18 – CAMARA-aerOS Integration: App Onboarding and Instantiation Sequence Diagram.

This representation makes visible the role of the adapter at every stage of the process, preserving CAMARA's abstractions while converting and relaying the underlying operations to aerOS. Through this systematic translation of data structures, workflow steps, and orchestration artifacts, the adapter ensures that CAMARA's API semantics remain fully intact, while seamlessly binding them to the NGSI-LD-based and TOSCA-driven execution model that aerOS provides. The result is a consistent and operationally accurate bridge between two distinct frameworks, supporting application providers while conforming to aerOS orchestration mechanisms.

5.2.3.2.4 Endpoints

The aerOS enabler builds on the Meta-OS approach to intelligently manage and integrate a distributed set of network and computing resources into a seamless computing continuum. It enables the orchestration of hyper-distributed applications across the Edge–Cloud spectrum. Heterogeneous computing resources, potentially geographically dispersed or operating under different administrative domains, are federated and abstracted through aerOS, which uniformly exposes their capabilities to allow dynamic and on-demand service allocation. Developed as a cloud-native ecosystem, aerOS provides a comprehensive REST-based API layer that exposes its orchestration and federation functionalities. These APIs enable external systems to interact with aerOS domains for resource discovery, service deployment, lifecycle management, and monitoring across the continuum. The aerOS APIs are organized into two main categories, designed for seamless and standardized interaction:

1. Orchestration and Lifecycle Management APIs: This category comprises RESTful endpoints responsible for service orchestration and lifecycle control. Deployment requests are formatted as TOSCA (Topology and Orchestration Specification for Cloud Applications) descriptors, defining the application or service to be instantiated, potentially composed of multiple components, along with its networking requirements and infrastructure constraints. Complementary REST methods enable full lifecycle management, including service de-allocation, status retrieval, state transitions, and migration operations. The API supports querying the status of deployed components, following a predefined state model (e.g., *Running*, *Finished*, *Failed*). All services deployed via aerOS are containerized and executed over the underlying Kubernetes-based infrastructure.
2. Discovery and Query APIs: The second category includes interfaces used to discover and query registered entities such as Infrastructure Elements (IEs), zones, services, and service components. These interfaces are implemented according to the ETSI NGSI-LD standard, ensuring interoperability and semantic consistency. Responses are produced as JSON-LD objects, providing either complete entity descriptions or filtered subsets based on specified attributes. The NGSI-LD model also supports advanced capabilities, including attribute filtering, linked queries, and cross-entity relationships, thereby offering consistent visibility into the state and attributes of continuum entities.

Together, these two categories of APIs provide the foundation for unified orchestration, resource abstraction and real-time monitoring across federated aerOS domains. They enable external enablers and management platforms to seamlessly interact with aerOS for dynamic service provisioning, continuum awareness, and context-driven orchestration. Table 29 lists the most relevant aerOS API endpoints.

Table 29 – Most important aerOS API endpoints.

REST Method	Resource URI	Body/Parameter	Description
POST	/hlo_fe/services/{service_id}	Path: service_id Body: TOSCA	Orchestrate a new service based on the topology defined in a TOSCA file.

DELETE	/hlo_fe/services/{service_id}	Path: service_id	Stop a service instance
PUT	/hlo_fe/services/{service_id}	Path: service_id	(Re)Deploy an already boarded (via POST) service
GET	/hlo_fe/services/{service_id}	Path: service_id	Get status of all service components
DELETE	/hlo_fe/services/{service_id}/purge	Path: service_id	Purge all service data from the continuum registry
All Endpoints Below Follow the NGSI-LD Standard			
GET	/entities?type=InfrastructureElement	Path: type of entity (InfrastructureElement)	Get all integrated processing units (Infrastructure Elements)
GET	/entities?type=Service	Path: type of entity (Service)	Get all services
GET	/entities?type=ServiceComponent	Path: type of entity (ServiceComponent)	Get all service components

The above aerOS API table summarizes the REST API architecture which enables both service orchestration and entity discovery within the aerOS continuum.

The first tier, the Orchestration Layer (Service Management), exposes lifecycle management operations under the */hlo_fe/services* endpoint family. These endpoints enable dynamic and programmable control of deployed services:

- POST accepts a TOSCA-based descriptor to instantiate a new service according to the defined topology and resource requirements.
- DELETE stops and removes running service instances, supporting resource release and cleanup.
- PUT triggers redeployment or updates of already-boarded services, facilitating continuous delivery and service adaptation.
- GET retrieves the operational status and component inventory of a deployed service, enabling runtime monitoring.
- DELETE performs cleanup of service-related information by purging them from the continuum.

The second tier, the Entity Discovery Layer, follows the ETSI NGSI-LD specification and is accessed via the */entities* endpoint family. These APIs provide standardized and semantically rich queries across all aerOS entity categories, among which the following are of most importance and relevance to ENVELOPE use case:

- InfrastructureElement: retrieves information about available compute and network resources across federated domains.
- Service: lists information and metadata concerning services orchestrated across the continuum.
- ServiceComponent: provides detailed information regarding the individual components which are part of each service.

All NGSI-LD endpoints support **type-based queries** (e.g., */entities?type={{entity_type}}*) to retrieve resources of a given type, as well as **filter-based queries** (e.g., */entities?type=ServiceComponent&q=service=="{{service_id}}"*) to obtain entities matching specific criteria. They also allow **entity-specific queries** (e.g., */entities/{{entity_id}}*) to access information about a particular resource in the continuum. The syntax and semantics of these queries are defined in the *ETSI GS CIM 009* specification. All query responses are returned as **JSON-LD** objects, enabling external applications and higher-level orchestration systems to dynamically discover, correlate, and monitor continuum resources without requiring direct access to the underlying domains.

5.2.3.2.5 Request and response examples

This sub-section presents concrete examples of service deployment requests in TOSCA format and their corresponding orchestrator responses. The request examples highlight key configuration elements controlling placement, resource allocation and container startup behavior. The response examples demonstrate the orchestrator's handling of successful and failed requests across various HTTP status codes.

Endpoint: POST /hlo_fe/services/{service_id}
Content-Type: application/x-yaml
Purpose: This operation initiates the deployment of a service over the continuum infrastructure. The request body contains a TOSCA-based service descriptor that specifies the workload to be instantiated (e.g., container image, environment variables) along with its placement and resource constraints (e.g., CPU, memory, domain). Upon validation, the orchestrator acknowledges the request and starts the allocation process.

Request Body (TOSCA Format)

The payload uses the TOSCA Simple YAML 1.3 syntax to describe the deployment requirements. In the examples below we have in red boxes we highlight configuration and constraints requested for the (containerized) workload deployment:

```
tosca_definitions_version: tosca_simple_yaml_1_3
description: NEF Deployment (mongodb, amf-crawler, nef)
serviceOverlay: true

node_templates:
  mongodb:
    type: tosca.nodes.Container.Application
    requirements:
      - network:
          properties:
            ports:
              api:
                properties:
                  protocol: [tcp]
                  source: 27017
            exposePorts: false
      - host:
          node_filter:
            capabilities:
              - host:
                  properties:
                    cpu_arch:
                      equal: x64
                    realtime:
                      equal: false
                    cpu_usage:
                      less_or_equal: '0.4'
                    mem_size:
                      greater_or_equal: '1'
                    domain_id:
                      equal: urn:ngsi-ld:Domain:NCSRD
    artifacts:
      mongodb-image:
        file: mongo:latest
        type: tosca.artifacts.Deployment.Image.Container.Docker
        repository: docker_hub
    interfaces:
      Standard:
        create:
          implementation: mongodb-image
          inputs:
            envVars:
              - ENV1: "value"
              - ENV2: "value2"
```

Figure 19 – aerOS deployment descriptor (TOSCA based) with host capabilities requirements.

```
tosca_definitions_version: tosca_simple_yaml_1_3
description: TOSCA for network performance
serviceOverlay: true

node_templates:
  influxdb:
    type: tosca.nodes.Container.Application
    requirements:
      - network:
          properties:
            ports:
              fastapi:
                properties:
                  protocol: [tcp]
                  source: 8086
            exposePorts: true
      - host:
          node_filter:
            properties:
              id: "urn:ngsi-ld:InfrastructureElement:CloudFerro:fa163e5e25ef"
    artifacts:
      influxdb-image:
        file: p4lik4ri/influxdb
        type: tosca.artifacts.Deployment.Image.Container.Docker
        repository: docker_hub
    interfaces:
      Standard:
        create:
          implementation: influxdb-image
          inputs:
            envVars:
              - ENV1: void
```

Figure 20 – aerOS deployment descriptor (TOSCA based) with host specification.

Key elements of interest are:

- *ports*: Defines the networking configuration of the service, including exposed ports and supported protocols. An example is protocol: [tcp], source: 8086.
- *exposePorts*: Indicates whether the container port should be externally accessible. An example is true.
- *host.node_filter.capabilities.host.properties*: Declares resource constraints for auto-matic placement. Specifies requirements such as CPU architecture, memory size, or domain. Used when the orchestrator selects the target node dynamically based on avail-able resources. An example is CPU architecture (x64), memory (≥1 GiB), CPU usage (≤0.4), real-time support (false), domain (urn:ngsi-ld:Domain:NCSRD).
- *host.node_filter.properties.id*: Provides a list of one or more preferred infrastructure elements (InfrastructureElement URNs). Used for direct or guided placement, where deploy-

ment must occur on specific target nodes or within a defined subset of resources. An example is id: ["urn:ngsi-ld:InfrastructureElement:CloudFerro:fa163e5e25ef", "urn:ngsi-ld:InfrastructureElement:NCSRD:1234abcd"].

- *artifacts*: Defines the container image, its deployment type, and the source repository. An example is file: p4li4k4ri/influxdb, type: tosca.artifacts.Deployment.Image.Container.Docker, repository: docker_hub.
- *interfaces.Standard.create.inputs.envVars*: Specifies environment variables passed to the container at instantiation time to configure runtime parameters. An example is ENV1=value, ENV2=value2.
- *interfaces.Standard.create.inputs.cli*: Declares command-line arguments that are appended when starting the container. Used to provide startup flags or configuration arguments that complement environment variables. An example is --config /etc/influxdb/config.yml, --auth-enabled true.

Some notes:

- The **host.node_filter** section can define **either**
  - *capabilities.host.properties* (for *requirement-based placement*), **or**
  - *properties.id* (for *explicit resource selection*).

    Both structures are valid but **mutually exclusive** within the same template.

When multiple infrastructure elements are listed under properties.id, the orchestrator selects the most suitable one according to internal policies.

envVars and *cli* provide complementary runtime configuration mechanisms — the former through environment variables, the latter through direct container start-up parameters.

When the TOSCA request is valid, the orchestrator acknowledges that the service allocation has started and status code is 202 (Accepted). Headers include a *Location* field with value */hlo_fe/services/{service_id}.* So, the request does not immediately deploy the service but initiates its orchestration workflow. The 202 response aligns with REST best practices for asynchronous operations. Subsequent GET calls to /hlo_fe/services/{service_id} allow monitoring of the service status. The actual response body has the following format:


```
{
    "serviceId": "urn:ngsi-ld:Service:envelope-app",
    "status": "starting",
    "message": "Service allocation initiated. Check service components status at the URL provided.",
    "url": "/hlo_fe/services/urn:ngsi-ld:Service:envelope-app"
}
```


Figure 21 – aerOS response or service allocation request.

In Table 30, we provide the full set of responses for the deployment request (POST) endpoint.

Table 30 – aerOS deployment endpoint (POST, /hlo_fe/services) responses.

HTTP Code	Meaning	Typical Cause	Example Response Body
202 Accepted	Service allocation successfully initiated.	Valid TOSCA descriptor received; orchestration workflow started.	json { "serviceId": "nef-mongodb", "status": "starting", "message": "Service

			allocation initiated. Check service components status at the URL provided.", "url": "/hlo_fe/services/nef-mongodb" }
400 Bad Request	Payload validation error generated by FastAPI/Pydantic.	YAML body does not match expected schema (e.g., wrong data types, missing mandatory fields, syntax issues). This response is automatically generated by FastAPI/Pydantic validation	json { "detail": "Invalid Service Parameters" }
422 Unprocessable Entity	Missing post body.	Missing request body	json { "detail": [{ "type": "missing", "loc": ["body"], "msg": "Field required", "input": null, "url": "https://errors.pydantic.dev/2.6/v/missing" }] }
500 Internal Server Error	Orchestrator could not publish the deallocation event.	Internal HLO (pipeline broken) error.	{ "detail": "Redpanda failure,HLO pipeline broken: <error details>" }

5.2.3.2.6 Error codes and error handling

All error codes, related to resources queries over the continuum and their handling are implemented as specified in NGSI-LD API v1.6.1 clause 6.3.2. As a quick reference we put the basic response codes in Figure 22.

Error Type	HTTP status
https://uri.etsi.org/ngsi-ld/errors/InvalidRequest	400
https://uri.etsi.org/ngsi-ld/errors/BadRequestData	400
https://uri.etsi.org/ngsi-ld/errors/AlreadyExists	409
https://uri.etsi.org/ngsi-ld/errors/OperationNotSupported	422
https://uri.etsi.org/ngsi-ld/errors/ResourceNotFound	404
https://uri.etsi.org/ngsi-ld/errors/InternalError	500
https://uri.etsi.org/ngsi-ld/errors/TooComplexQuery	403
https://uri.etsi.org/ngsi-ld/errors/TooManyResults	403
https://uri.etsi.org/ngsi-ld/errors/LdContextNotAvailable	503
https://uri.etsi.org/ngsi-ld/errors/NoMultiTenantSupport	501
https://uri.etsi.org/ngsi-ld/errors/NonexistentTenant	404

Figure 22 – List of NGSI-LD error codes.

5.2.3.2.7 Implementation details

AerOS is built using a set of cloud-native and identity-management technologies that together enable secure, scalable orchestration across multiple edge–cloud domains.

Each aerOS domain runs on Kubernetes, exposing its APIs through Nginx Ingress and KrakenD acting as the API Gateway and Policy Enforcement Point. User authentication and authorization

rules are centrally registered using Keycloak, backed by LDAP, which issues OAuth2 tokens which are used to authorize APIs access within each aerOS domain. The HLO is implemented in Python (FastAPI when APIs are used), interprets TOSCA descriptors, queries local resources, performs AI-based placement and coordinates deployments. Internal communication among HLO components is event-driven, using Redpanda as a message bus with protobuf serialization. Actual workload placement is performed by the LLO, which is implemented as K8s Operators developed in Go. All context and resource information within each domain is maintained in Orion-LD, the NGSI-LD context broker, which allows both domains to expose and consume structured infrastructure and service metadata.

The deployed continuum, used for ENVELOPE activities, consists of two interconnected aerOS domains: one hosted at NCSRD and one at OTE. The NCSRD site acts as the entry point of the continuum and hosts the central IdM services, including Keycloak, LDAP (also the aerOS GUI). This site is deployed on three virtual machines running in an OpenNebula-based VIM environment. The NCSRD domain also exposes a public IP address and DNS name used by external clients to access the GUI and aerOS APIs. The OTE site hosts a second aerOS domain integrating two virtual machines. Both the NCSRD and OTE domains operate their own Kubernetes clusters, Ingress controllers, PEP/API Gateways, HLO/LLO components and Orion-LD context brokers.

The two sites are interconnected via a dark-fiber link, and the LANs hosting the aerOS domains are fully routable across this connection. This enables seamless inter-domain communication and makes it possible for components such as HLO and Orion-LD to exchange information transparently. A set of NGSI-LD subscriptions has been established between the two Orion-LD brokers hosted in the aerOS domains. These subscriptions allow each domain to receive real-time updates about all aerOS resources, including computing resource capabilities and availabilities and deployed services and their status. This creates a federated NGSI-LD knowledge base spanning both domains, enabling placement decisions, monitoring and resource discovery across the entire continuum, not just within individual domains.

5.2.3.3 Liqo-based K8s multi-cluster enabler

Table 31 – API implemented and modules used for the integration with the Liqo-based K8s multi-cluster enabler.

API name	Description	Modules Involved	Use Cases
Edge Application Management	Lifecycle of applications and discovery of edge zones/clusters: submit apps (registration), instantiation, deployment, termination; and listing of edge clusters and zones.	AF, Liqo-based K8s multi-cluster enabler, vanilla K8s clusters	Dt-UC-5

This sub-section describes the implementation details of the CAMARA Edge Application Management API when integrated with the Liqo-based K8s multi-cluster enabler. Details of such integration is given for the different endpoints categories as defined by the CAMARA specification:

Application (**/apps**), *Application Instances* (**/appinstances**), Edge Cloud Zones (**/edge-cloud-zones**), and *Clusters* (**/clusters**).

5.2.3.3.1 Application onboarding: *Application (/apps)*

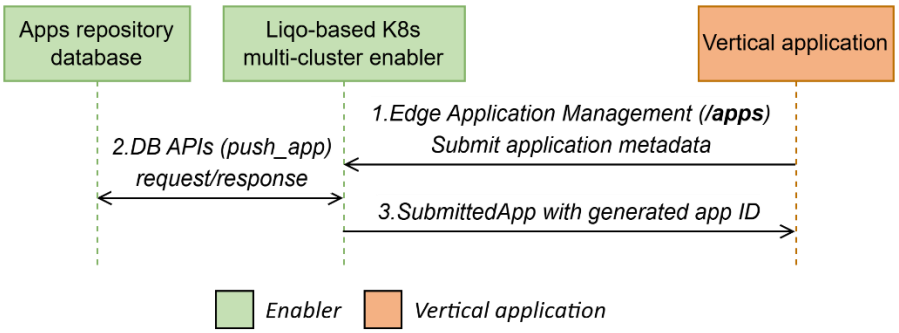

Figure 23 – Message flow for the Edge Application Management **Application** API.

The Edge Application Management *Application* (**/apps**) API is used to onboard (register) a new application in the edge platform. The message flow procedure for its integration is shown in Figure 23:

1. The vertical application submits application metadata with description of application requirements (i.e., network and computing requirements) and how the application can be run in the edge platform. In the current implementation, this application metadata must contain a URI reference to a Helm chart.
2. The edge enabler pushes the application metadata to the application repository database.
3. The edge enabler confirms the request with the payload SubmittedApp.

5.2.3.3.2 Application instantiation and handover: *Application Instance (/appinstances)*

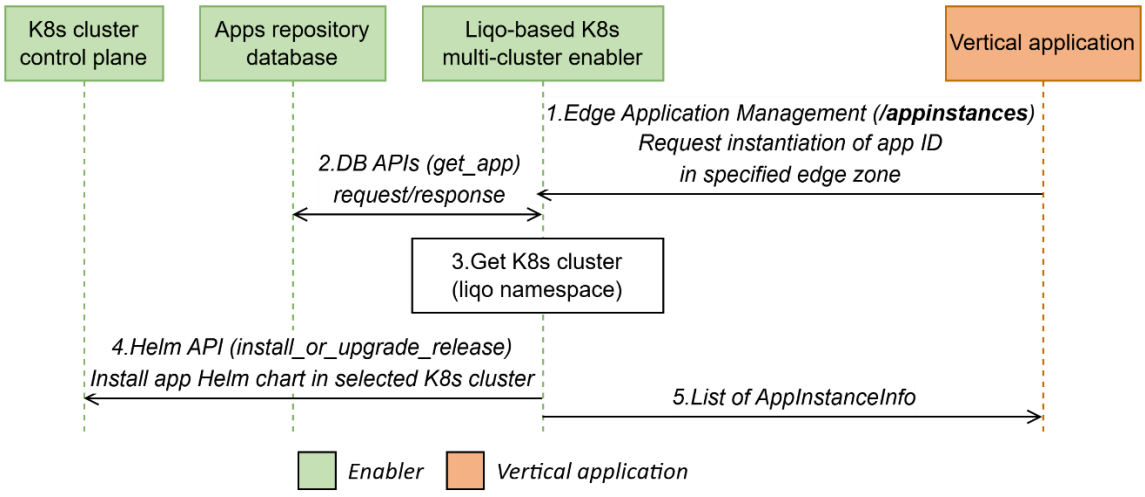

Figure 24 – Message flow for the Edge Application Management **Application Instances** API.

The Edge Application Management *Application Instance* (**/appinstances**) API is used to instantiate an application in a particular edge zone. This API allows vertical applications to perform handover (migration) of edge applications from one edge zone to another, e.g., by requesting the instantiation

in a second edge zone and requesting the deletion of the old application instance. The message flow procedure for its integration is shown in Figure 24:

1. The vertical application submits a request for the instantiation of an application of a given app ID to be deployed in a specified edge zone.
2. The edge enabler retrieves the corresponding application metadata from the application repository database as well as Helm chart associated with the application metadata.
3. The edge enabler finds the K8s cluster that maps to the requested edge zone. This is done via mapping of the edge zones and underlying Liqo K8s namespaces available in the edge platform.
4. The edge enabler installs the Helm chart in the corresponding K8s cluster.
5. The edge enabler confirms the application instantiation request to the vertical application.

5.2.3.3.3 Edge zone discovery: *Edge Cloud Zones (**/edge-cloud-zones**), Clusters (**/clusters**)*

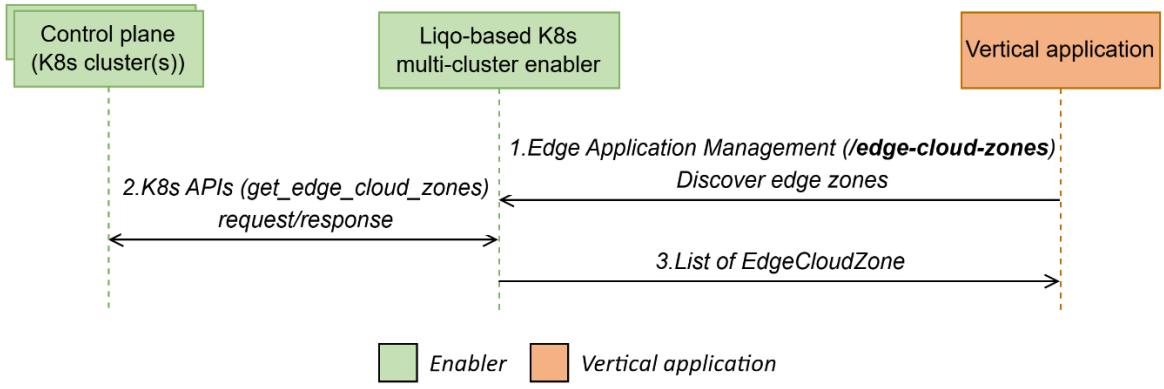

Figure 25 – Message flow for the Edge Application Management **Edge Cloud Zones** API.

The Edge Application Management *Edge Cloud Zones* (**/edge-cloud-zones**) API is used to discover available edge zones in the edge platform. The message flow procedure for its integration is shown in Figure 25:

1. The vertical application submits a request to retrieve the list of edge zones available in the edge platform
2. The edge enabler retrieves the information about the available edge zones via K8s APIs.
3. The edge enabler confirms the request and responds with the list of available edge cloud zones.

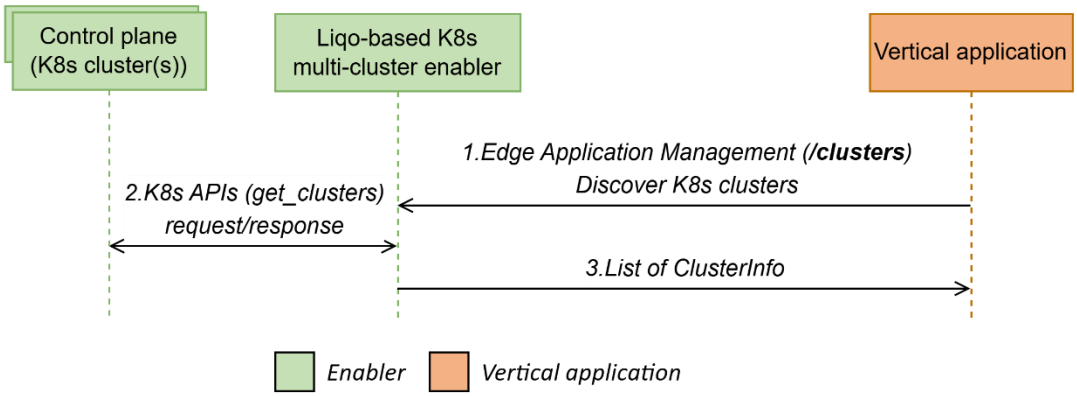

Figure 26 – Message flow for the Edge Application Management **Clusters** API.

The Edge Application Management *Clusters* (**/clusters**) API may also be used to discover edge zones. In this case, more detail information about each edge cluster corresponding to an edge zone is returned to the API consumer. The message flow procedure for its integration is shown in Figure 26:

1. The vertical application submits a request to retrieve the list of clusters (i.e., K8s clusters) available.
2. The edge enabler retrieves the list of clusters via K8s APIs.
3. The edge enabler confirms the request and responds with the list of available clusters.

5.2.3.3.4 Implementation Details

Figure 27 illustrates the implemented architecture of the Liqo-based K8s multi-cluster enabler. At the top, an application server interacts with CAMARA Edge Application Management APIs to manage application lifecycle operations. The core component is the Liqo-based multi-cluster orchestrator that provides APIs for application registration (**/apps**), instantiation (**/appinstances**), and edge-cloud zone discovery **(/edge-cloud-zones**, **/clusters**). This orchestrator communicates with an application repository via database APIs and leverages Helm/Kubernetes APIs to deploy workloads across distributed clusters. At the bottom, two zones are shown: an edge cluster and a cloud cluster, each with its own Kubernetes control plane and nodes where vertical applications are instantiated. Liqo enables resource sharing between these clusters, ensuring dynamic, elastic deployment across edge and cloud environments to support low-latency and scalable services.

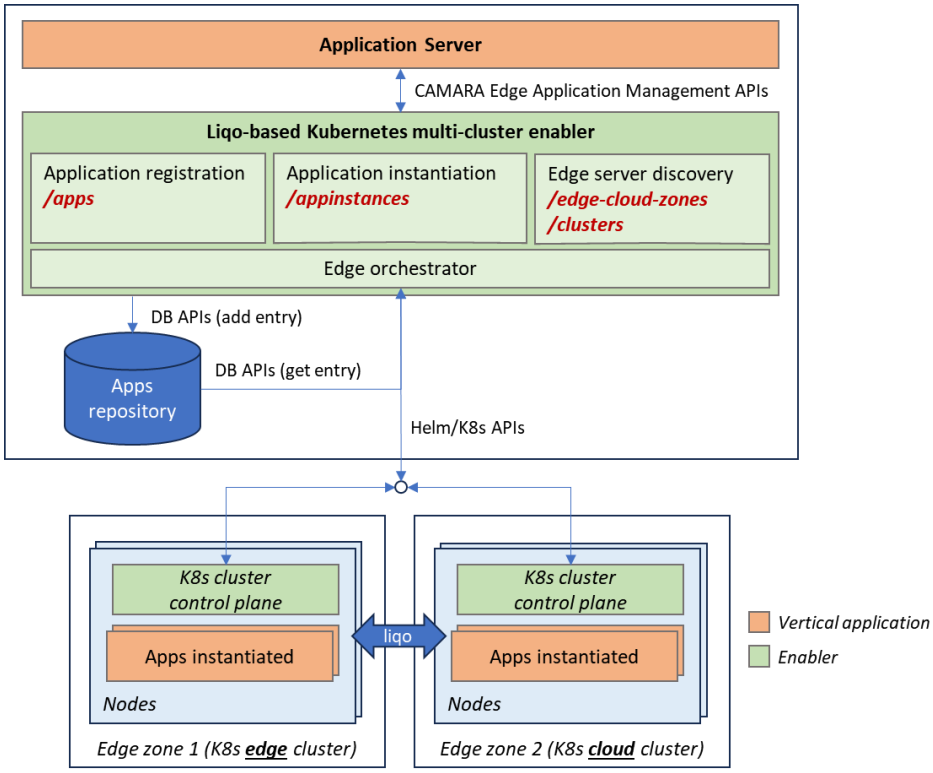

Figure 27 – implementation architecture of the Liqo-based K8s multi-cluster enabler.

The Liqo-based K8s multi-cluster enabler has been implemented in Python. In the northbound, the FastAPI library has been used as basis for developing the CAMARA Edge Application Management API endpoints. The DB APIs have been implemented with the Python library *pymongo* that is used to communicates with MongoDB using the MongoDB Wire Protocol, which is a binary protocol over TCP sockets. The Helm APIs have been implemented using the Python library *pyhelm3*, which acts as a client wrapper around the Helm CLI. Finally, the Kubernetes (K8s) APIs have been implemented using the Python library *kubernetes.client*, which is the official Python SDK for Kubernetes that provides a direct interface to the Kubernetes API server.

5.2.4 Endpoints

Table 32 – Overview of implemented CAMARA Edge Application Management endpoints.

Resource name	Resource URI	HTTP method	Description
Applications	/apps	POST	Submit application metadata
Applications	/apps	GET	List applications
Individual Application	/apps/{appId}	GET	Retrieve application
Individual Application	/apps/{appId}	DELETE	Delete application

Application Instances	/appinstances	POST	Create instance(s) for an app
Application Instances	/appinstances	GET	List instances (filter by appId, appInstanceId, region)
Individual Application Instance	/appinstances/{appInstanceId}	DELETE	Terminate instance
Clusters	/clusters	GET	List clusters (filter by region, clusterRef, edgeCloudZoneId)
Edge Cloud Zones	/edge-cloud-zones	GET	List zones (filter by region, status)

5.2.5 Request and response examples

POST /apps

Request fields (*AppManifest* data type):

- **name**: Name of the application.
- **appProvider**: Human readable name of the Application Provider.
- **version**: Application version information.
- **packageType**: Format of the application image package (e.g., CONTAINER, HELM).
- **operatingSystem**: Information about the Operating System of the application image (e.g., Ubuntu 22.04 x86_64).
- **appRepo**: Repository where Application Provider stores the application image
    - **type**: Application repository and image URI information. PUBLICREPO is used of public urls like github, helm repo etc. PRIVATEREPO is used for private repo managed by the application developer. Private repo can be accessed by using the app developer provided username and password.
    - **imagePath**: A Uniform Resource Identifier (URI) as per RFC 3986, identifies the endpoint within an Edge Cloud Zone where the user equipment may connect to the selected application instance.
    - **userName**: Username to access the Helm chart, docker-compose file or VM image repository.
    - **credentials**: Password or personal access token created by developer to access the app repository.
    - **authType**: The credentials can also be formatted as a Basic auth or Bearer auth in HTTP "Authorization" header.
    - **checksum**: MD5 checksum for VM and file-based images, sha256 digest for containers.

- **requiredResources**: Fundamental hardware requirements to be provisioned by the Application Provider. Only KubernetesResources is supported in the current implementation.
- **componentSpec**: Information defined in "appRepo" point to the application descriptor e.g. Helm chart, docker-compose yaml file etc. The descriptor may contain one or more containers and their associated meta-data. A component refers to additional details about these containers to expose the instances of the containers to external client applications. App provider can define one or more components (via the associated network port) in componentSpec corresponding to the containers in helm charts or docker-compose yaml file as part of app descriptors.
    - **componentName**: Component name must be unique with an application.
    - **networkInterfaces**: Each application component exposes some ports either for external users or for inter component communication. Application provider is required to specify which ports are to be exposed and the type of traffic that will flow through these ports.

Response fields (*SubmittedApp* data type):

- **appId**: Unique application identifier.

Table 33 – Example for POST /apps.

POST /apps

```
{
  "name": "ExampleApp",
  "version": "1.0.0",
  "appProvider": "ExampleProvider",
  "packageType": "HELM",
  "appRepo": {
    "type": "PUBLICREPO",
    "imagePath": "https://charts.example.com/helm/example-chart:1.0.0",
    "authType": "NONE"
  },
  "requiredResources": {
    "infraKind": "kubernetes",
    "applicationResources": {
      "cpuPool": {
        "numCPU": 2,
        "memory": 4096,
        "topology": {
          "minNumberOfNodes": 3,
          "minNodeCpu": 1,
          "minNodeMemory": 2048
        }
      },
      "gpuPool": {
        "numCPU": 1,
        "memory": 2048,
        "gpuMemory": 8,
        "topology": {
          "minNumberOfNodes": 1,
          "minNodeCpu": 1,
          "minNodeMemory": 2048,
          "minNodeGpuMemory": 4
        }
      }
    },
    "isStandalone": false,
```

```
    "version": "1.29",
    "additionalStorage": "50GB",
    "networking": {
      "primaryNetwork": {
        "provider": "cilium",
        "version": "1.13"
      },
      "additionalNetworks": [
        {
          "name": "net1",
          "interfaceType": "netdevice"
        }
      ]
    },
    "addons": {
      "monitoring": true,
      "ingress": true
    }
  },
  "componentSpec": [
    {
      "componentName": "frontend",
      "networkInterfaces": [
        {
          "interfaceId": "frontendPort",
          "protocol": "TCP",
          "port": 8080,
          "visibilityType": "VISIBILITY_EXTERNAL"
        }
      ]
    },
    {
      "componentName": "backend",
      "networkInterfaces": [
        {
          "interfaceId": "backendPort",
          "protocol": "TCP",
          "port": 9090,
          "visibilityType": "VISIBILITY_INTERNAL"
        }
      ]
    }
  ]
}
```

Response

```
{
  "appId": "123e4567-e89b-12d3-a456-426614174000"
}
```

POST /appinstances

Request fields (custom data type):

- **appId**: A globally unique identifier associated with the application. The edge platform generates this identifier when the application is submitted.
- **appZones**: List of edge cloud zones and/or Kubernetes cluster reference where the Application Provider wants to instantiate the application.

Response fields (List of *AppInstanceInfo* data type objects):

- **appInstanceId**: A globally unique identifier associated with a running instance of an application. The edge platform generates this identifier when the instantiation in the edge cloud zone is successful.
- **status**: Status of the application instance (default is 'unknown'). Possible values: ‘ready’, ‘instantiating’, ‘failed’, ‘terminating’, ‘unknown’.
- **componentEndpointInfo**: Information about the IP and Port exposed by the edge platform. The application client shall use these access points to reach this application instance.
    - **interfaceId**: This is the interface Identifier that application provider defines when application is being submitted.
    - **accessPoints**: Application endpoint (e.g., port, IP address) for an specific instance that is running in an specific edge cloud zone.
- **kubernetesClusterRef**: A global unique identifier associated with a Kubernetes cluster infrastructure.
- **edgeCloudZone**: An edge cloud zone, uniquely identified by a combination of the value of the edge cloud zone ID object and the value of the edge cloud provider object.
    - **edgeCloudZoneId**: Unique identifier created by the edge platform to identify an edge cloud zone.
    - **edgeCloudZoneName**: Human readable name of the geographical zone of the edge cloud.
    - **edgeCloudZoneStatus**: Status of the Edge Cloud Zone (default is 'unknown'). Possible values: ‘active’, ‘inactive’, ‘unknown’.
    - **edgeCloudProvider**: Human readable name of the edge cloud provider.
    - **edgeCloudRegion**: Human readable name of the geographical edge cloud region of the edge cloud.

Table 34 – Example for POST /appinstances.

POST /appinstances

```
{
  "appId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "appZones": [
    {
      "kubernetesClusterRef": "642f6105-7015-4af1-a4d1-e1ecb8437abc",
      "EdgeCloudZone": {
        "edgeCloudZoneId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
        "edgeCloudZoneName": " edgeCloudZone1",
        "edgeCloudZoneStatus": "unknown",
        "edgeCloudProvider": " edgeCloudProvider1",
        "edgeCloudRegion": " edgeCloudRegion1"
      }
    }
  ]
}
```

Response

```
{
  "appInstances": [
    {
      "appInstanceId": "3fa85f64-5717-4562-b3fc-2c963f66afa7",
      "status": "unknown",
      "componentEndpointInfo": [
        {
          "interfaceId": "6_2WLl2Sq0x0ou0",
          "accessPoints": {
            "port": 8080,
```

```
            "fqdn": "accesspoint1@hotsname.com",
            "ipv4Addresses": [
              "192.168.0.1"
            ],
            "ipv6Addresses": [
              "2001:db8:85a3:8d3:1319:8a2e:370:7344"
            ]
          }
        }
      ],
      "kubernetesClusterRef": "642f6105-7015-4af1-a4d1-e1ecb8437abc",
      "edgeCloudZone": {
        "edgeCloudZoneId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
        "edgeCloudZoneName": "edgeCloudZone1",
        "edgeCloudZoneStatus": "unknown",
        "edgeCloudProvider": "edgeCloudProvider1",
        "edgeCloudRegion": "edgeCloudRegion1"
      }
    }
  ]
}
```


GET /edge-cloud-zones

Optional request parameters (no request body required):

- **x-correlator**: Correlation id for the different services.
- **region**: Human readable name of the geographical edge cloud region of the edge cloud.
- **status**: Human readable status of the edge cloud zone. Available values: ‘active’, ‘inactive’, ‘unknown’.

Response fields (List of *EdgeCloudZones* data type objects):

- **edgeCloudZoneId**: Unique identifier created by the edge platform to identify an edge cloud zone.
- **edgeCloudZoneName**: Human readable name of the geographical zone of the edge cloud.
- **edgeCloudZoneStatus**: Status of the Edge Cloud Zone (default is 'unknown'). Possible values: ‘active’, ‘inactive’, ‘unknown’.
- **edgeCloudProvider**: Human readable name of the edge cloud provider.
- **edgeCloudRegion**: Human readable name of the geographical edge cloud region of the edge cloud.

Table 35 – Example for GET /edge-cloud-zones.

GET /edge-cloud-zones
• **correlator**: "100" • **region**: "edgeCloudRegion1" • **status**: "unknown".
Response

[
  {
  "edgeCloudZoneId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "edgeCloudZoneName": "edgeCloudZone1",
  "edgeCloudZoneStatus": "unknown",
  "edgeCloudProvider": "edgeCloudProvider1",
  "edgeCloudRegion": "edgeCloudRegion1"
  }

```
]
```

GET /clusters

Optional request parameters (no request body required):

- **x-correlator**: Correlation id for the different services.
- **region**: Human readable name of the geographical edge cloud region of the edge cloud.
- **clusterRef**: A globally unique identifier for the cluster.
- **edgeCloudZoneId**: Edge cloud zone identifier.

Response fields (List of *ClusterInfo* data type objects):

- **name**: Name of the Cluster.
- **provider**: Human readable name of the application provider.
- **clusterRef**: A global unique identifier associated with a K8s cluster infrastructure.
- **edgeCloudZoneId**: Unique identifier created by the edge platform to identify an edge cloud zone.
- **edgeCloudRegion**: Human readable name of the geographical edge cloud region of the edge cloud.
- **version**: Kubernetes version of the cluster.
- **nodePools**: A K8s node pool is a set of Kubernetes nodes that have the same configuration (vCPU, memory, networking, OS, etc) on each node.
    - **name**: Human readable name of the Kubernetes Node Pool.
    - **numNodes**: Number of nodes in the Node Pool.
    - **scalable**: Indicates if the node pool can be dynamically scaled up by the system to accommodate more applications and dynamically scaled down by the system when there are unused resources.
    - **nodeResources**: number of CPUs and memory information.

Table 36 – Example for GET /clusters.

GET /clusters

- **correlator:** "100"
- **region:** "edgeCloudRegion1"
- **clusterRef:** "642f6105-7015-4af1-a4d1-e1ecb8437abc"
- **edgeCloudZoneId:** "842f6105-7015-4af1-a4d1-e1ecb8437abc"

Response

```
[
  {
    "name": " edgeCluster1",
    "provider": "edgeCloudProvider1",
    "clusterRef": "842f6105-7015-4af1-a4d1-e1ecb8437abc",
    "edgeCloudZoneId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "edgeCloudRegion": "edgeCloudRegion1",
    "version": "1.34",
    "nodePools": [
      {
        "name": "pool1",
        "numNodes": 2,
        "scalable": false,
        "nodeResources": {
          "numCPU": 2,
```

```
            "memory": 4096
          }
        }
      ]
    }
]
```

5.2.6 Error codes and error handling

The status codes and their handling are implemented as specified in CAMARA Edge Application Management specification version v0.9.3-wip (April 2025), see exact git commit for the YAML specification implemented here.

5.3 Traffic Influence API

5.3.1 Short description

The CAMARA Traffic Influence API enables application providers and network operators to optimize routing between user devices (UEs) and the most suitable edge server. Its primary function is to influence traffic flow so that data is directed to the nearest application server instance that is running at the optimal edge server with respect to a given user device (UE), thereby reducing latency and improving performance for applications in the automotive (e.g., tele-operation) and other domains (e.g., VR/XR). The Traffic Influence API is meant to be used in coordination with the CAMARA Edge Application Management APIs which allow for the deployment of an application in a specified edge zone.

5.3.2 Relation with standards

This specification complies with the Traffic Influence API following CAMARA specification version 0.9.0-rc.1 (April 2025), see exact git commit for the YAML specification implemented here. In addition, the current implementation complies with the 3GPP specification TS 29.522 v17.9.0 for the NEF ServiceParameter exposure and TS 23.502 v17.12.0 section 4.15.6.10 describing the *Application guidance for URSP determination* procedure.

5.3.3 Functional overview

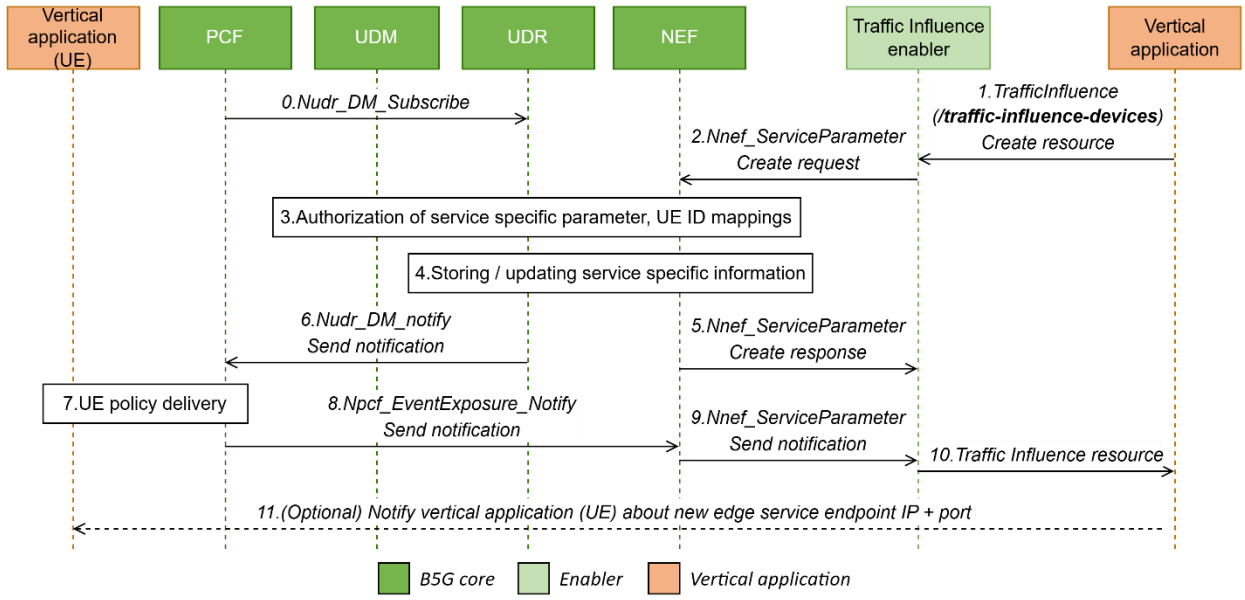

Figure 28 - Message flow for the **Traffic Influence Devices** API

The message flow procedure for the *Traffic Influence Devices* API is shown in Figure 28. Steps 2-9 follow the 3GPP specification TS 23.502[8] v17.12.0 section 4.15.6.7 (*Service specific parameter provisioning*) by following the guidelines described in section 4.15.6.10 (*Application guidance for URSP determination*) in the same document.

0. Prior to the beginning of this message flow, the PCF subscribes to changes in the user subscription database in the UDR.
1. The vertical application initiates a TrafficInfluence request to create a resource for traffic-influence-devices. In this resource, the target UE (device), application ID and application instance ID, and edge zone are included.
2. The Traffic Influence enabler sends a Nnef_ServiceParameter create request to the NEF containing information about the service description, service parameters, target UE, and subscription to notification about the outcome of the UE policy delivery.
3. The NEF may need to authorize the service specific parameter provisioning request with the UDM. In addition, The NEF may perform mapping of the GPSI in Target UE Identifier into SUPI, according to information received from UDM.
4. The new service-specific information is stored or updated in the UDR.
5. The NEF responds to the Traffic Influence enabler with a Nnef_ServiceParameter create response.
6. The PCF receives a Nudr_DM_Notify notification of data change from the UDR.
7. The PCF initiates the UE Policy delivery to vertical application's UE.
8. The PCF sends an Npcf_EventExposure_Notify notification to the NEF.
9. The NEF sends a Nnef_ServiceParameter notification to the Traffic Influence enabler.
10. The Traffic Influence enabler returns the created Traffic Influence resource to the vertical application.
11. Optionally, the vertical application may notify the vertical application's UE about the new application server IP and port.

[8] https://www.3gpp.org/ftp/Specs/archive/23_series/23.502/23502-hc0.zip

5.3.4 Endpoints

Table 37 – Overview of implemented CAMARA Traffic Influence endpoints.

Resource name	Resource URI	HTTP method	Description
Traffic Influence Devices	/traffic-influence-devices	POST	Creates a new Traffic Influence resource influencing the traffic toward local instances of the Application for a specific user.
Individual Traffic Influence resources	/traffic-influences/{trafficInfluenceID}	DELETE	Delete an existing Traffic Influence resource

5.3.5 Request and response examples

POST /traffic-influence-devices

Request fields (*PostTrafficInfluenceDevice* data type):

- **apiConsumerId**: Unique Identifier of the API consumer.
- **appId**: A globally unique identifier associated with the application. The edge platform generates this identifier when the application metadata is submitted (see Edge Application Management */apps* endpoint).
- **appInstanceId**: A globally unique identifier associated with a running instance of an application. The edge platform generates this identifier when the application is instantiated (see Edge Application Management */appinstances* endpoint).
- **edgeCloudRegion**: Human readable name of the geographical edge cloud region of the edge cloud.
- **edgeCloudZoneId**: Unique identifier created by the edge cloud platform to identify an edge cloud zone within an edge cloud.
- **sourceTrafficFilters**: Public source port used by the device for flows to which the requested traffic influence should apply. Traffic influence will be applied to the flow between "sourcePort" and the application server address and port specified in "destinationTrafficFilters".
    - **sourcePort**: TCP or UDP port number
- **destinationTrafficFilters**: Identifies the destination IP packet filters. To be used when it is needed a traffic flow towards a specific edge service interface identified by a protocol and a port. Also the Protocol (e.g. TCP or UDP) can be specified.
    - **destinationPort**: TCP or UDP port number.

    - **destinationProtocol**: The protocol for the influenced flow. It can be specified, and it is identified by a string according to the column “Keyword” as defined by IANA, e.g. UDP or TCP.
- **notificationUri**: Defines the callback URI which should be notified in asynchronous way when the state for the requested resources changes (i.e. ordered to activated).
- **notificationAuthToken**: Authentication token for callback API.
- **device**: End-user equipment able to connect to a mobile network. In the current implementation, only IPv4 is accepted as identifier.

Response fields (*TrafficInfluence* data type):

- **trafficInfluenceID**: Identifier for the Traffic Influence resource. This parameter is returned by the API and it is used to update it (e.g., modify or delete the TrafficInfluence resource).
- **apiConsumerId**: Unique Identifier of the API consumer.
- **appId**: A globally unique identifier associated with the application. The edge platform generates this identifier when the application metadata is submitted (see Edge Application Management */apps* endpoint).
- **appInstanceId**: A globally unique identifier associated with a running instance of an application. The edge platform generates this identifier when the application is instantiated (see Edge Application Management */appinstances* endpoint).
- **edgeCloudRegion**: Human readable name of the geographical edge cloud region of the edge cloud.
- **edgeCloudZoneId**: Unique identifier created by the edge cloud platform to identify an edge cloud zone within an edge cloud
- **state**: It reports the state of the TrafficInfluence resource. When first invoked, the resource is 'ordered'. When the platforms prepares the resource, it is 'created'. When the new routing is enabled in the network, the state is 'active'. Possible values: ‘ordered’, ‘created’, ‘active’, ‘error’, ‘deletion in progress’, ‘deleted’.
- **sourceTrafficFilters**: Public source port used by the device for flows to which the requested traffic influence should apply. Traffic influence will be applied to the flow between "sourcePort" and the application server address and port specified in "destinationTrafficFilters".
    - **sourcePort**: TCP or UDP port number
- **destinationTrafficFilters**: Identifies the destination IP packet filters. To be used when it is needed a traffic flow towards a specific edge service interface identified by a protocol and a port. Also the Protocol (e.g. TCP or UDP) can be specified.
    - **destinationPort**: TCP or UDP port number.
    - **destinationProtocol**: The protocol for the influenced flow. It can be specified and it is identified by a string according to the column “Keyword” as defined by IANA, e.g. UDP or TCP.
- **notificationUri**: Defines the callback URI which should be notified in asynchronous way when the state for the requested resources changes (i.e. ordered to activated).
- **notificationAuthToken**: Authentication token for callback API.

Table 38 – Example for POST /traffic-influence-devices.

POST /traffic-influence-devices

```
{
  "apiConsumerId": "apiConsumer1",
  "appId": "6B29FC40-CA47-1067-B31D-00DD010662DA",
  "appInstanceId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "edgeCloudRegion": "edgeCloudRegion1",
  "edgeCloudZoneId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "sourceTrafficFilters": {
    "sourcePort": 65535
  },
  "destinationTrafficFilters": {
    "destinationPort": 65535,
    "destinationProtocol": "TCP"
  },
  "notificationUri": "https://www.example.com/notificationserver",
  "notificationAuthToken": "token1",
  "device": {
    "ipv4Address": {
      "publicAddress": "84.125.93.10"
    }
  },
}
```

Response

```
{
  "trafficInfluenceID": "trafficInfluenceID1",
  "apiConsumerId": "apiConsumer1",
  "appId": "6B29FC40-CA47-1067-B31D-00DD010662DA",
  "appInstanceId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "edgeCloudRegion": "edgeCloudRegion1",
  "edgeCloudZoneId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "state": "ordered",
  "sourceTrafficFilters": {
    "sourcePort": 65535
  },
  "destinationTrafficFilters": {
    "destinationPort": 65535,
    "destinationProtocol": "TCP"
  },
  "notificationUri": "https://www.example.com/notificationserver",
  "notificationAuthToken": "token1"
}
```

5.3.6 Error codes and error handling

The status codes and their handling are implemented as specified in CAMARA Traffic Influence specification version 0.9.0-rc.1 (April 2025), see exact git commit for the YAML specification implemented here.

5.3.7 Implementation Details

Figure 29 illustrates the implemented architecture of the Traffic Influence enabler. A vertical application consumes the CAMARA Traffic Influence APIs to trigger to request traffic re-routing for a particular UE. The Traffic Influence enabler translates the request from the **/traffic-influence-devices** endpoint to *Nnef_ServiceParameter* API calls at the B5G core. The B5G core supports URSP-based traffic steering to direct specific UE data traffic to a particular edge UPF. Edge UPFs

are present in each ‘edge zone’, thereby allowing for optimal traffic routing to a given edge application instance located in the closest edge zone.

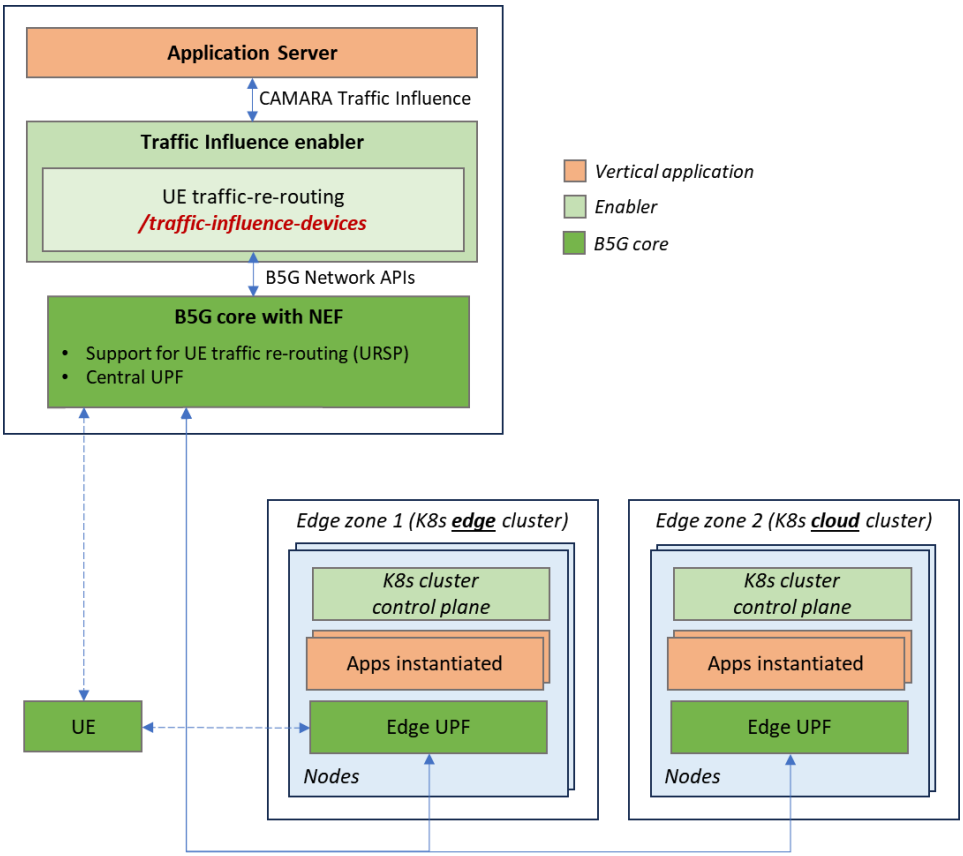

Figure 29 – Implementation architecture of the Traffic Influence enabler.

The Traffic Influence enabler has been implemented in Python. In the northbound, the FastAPI library has been used as basis for developing the CAMARA Traffic Influence API endpoints. Extensions to the B5G core (based on Open5GS) have been added to support URSP functionality. This involved adding extensions to the PCF, UDR, UDM, and NEF components. For URSP determination, the service parameters at the NEF component include a list of URSP rules that associate an application traffic descriptor and a desired route selection. In particular, the following service parameters have been implemented:

- Traffic descriptor: This parameter is used to define to which traffic the new UE route selection policy applies to. In the current implementation, only **server IP** and **port** of the application server are possible. This information is retrieved by the Traffic Influence enabler by using the received application ID and application instance ID.
- Route selection parameter: This parameter defines the route that must be selected for the traffic specified in traffic descriptor. Currently, **DNN** is the only parameter supported. The DNN is derived based on the edge cloud zone data received in the TrafficInfluence request.

5.4 Lessons Learned

The design and development stages of various edge computing enablers have underscored the benefits of implementing a unified set of high-level northbound edge computing APIs for vertical applications. This approach aims to deliver a consistent experience for API consumers and minimize integration efforts. The CAMARA Edge Application Management APIs facilitate this by enabling vertical applications to perform edge zone discovery, application orchestration (including onboarding and instantiation), and application handover (migration), regardless of the different underlying edge platforms deployed at each trial site. Additionally, all edge enablers utilize a modular, cloud-native architecture employing containerized orchestration, which supports flexible application deployment across multiple edge zones. The CAMARA Traffic Influence API can further be leveraged alongside the Edge Application Management APIs to optimize routing between user devices (UEs) and the most appropriate edge application instance for both application providers and network operators.

A notable limitation identified is the lack of widespread market adoption of standardized east-westbound interfaces, which may present significant interoperability challenges among edge domains during real-world deployments. While initiatives such as the ETSI MEC standard address edge federation, they have yet to achieve broad adoption within the industry. Consequently, this current lack of uniformity can increase integration complexity and prevent seamless application portability across multi-operator or multi-site environments.

6 Zero-Touch Management

6.1 Overview and Innovations

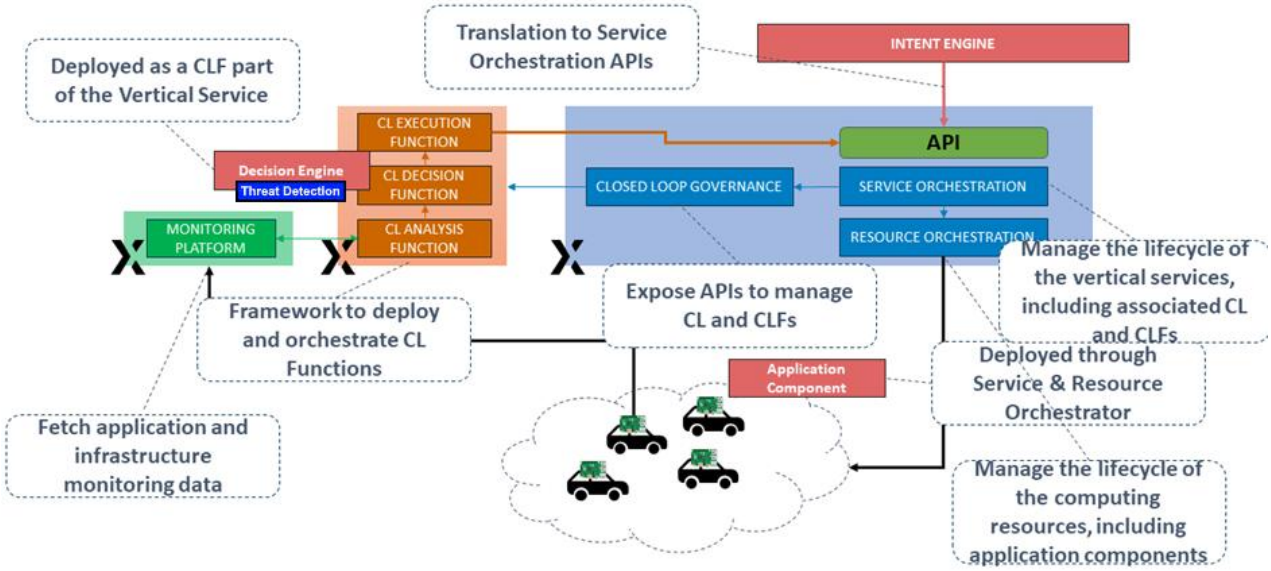

Figure 30 – Integration of the Zero Touch Management modules with the Intent and Decision Engines.

Zero-touch management enables dynamic onboarding and deployment of new CAM applications within converged telco mobile and edge/cloud infrastructures, ensuring seamless adaptability to changes in the underlying 5G/6G mobile network and infrastructure. It is an integral part of 5G/6G networks management architecture, offering combined network and service automation through the integration of cognitive and AI/ML based closed loops. The ENVELOPE zero-touch management solution bridges the management and orchestration of CAM services and applications (specifically in support of user experiments) with the management of mobile networks, ensuring that network performance requirements from the application layer are satisfied through adaptive network slice configurations. While embedded in the ENVELOPE overall solution, fully deployed in the Italian site infrastructure, and not directly exposed to experimenters for most of its primitives and service, network and edge/cloud resource management APIs, the zero-touch management framework is tightly coupled with the EaaS to enable programmability and full automation in experiments execution. The ENVELOPE Decision Engine is a closed loop (integrated with the zero-touch management solution in the project), offering malfunction/accident detection (as part of UC1) based on OBD-II data that are transmitted periodically. The Intent Engine offers innovative, intent-driven service deployment, offering the experimenters the possibility to execute an experiment using high-level natural language, which will be then translated into actionable tasks (to adjust an application or the network). The Threat Detection Module receives the same data the Decision Engine receives and it Identifies Frei and Stau attacks and data poisoning attempts. It can be viewed as a part of the Decision Engine. Finally, the Forecasting Component for Capacity Planning relates to Pre-scaling containers or allocation of vCPUs/RAM at MEC. It is needed in both UCs in the Italian site for heavy edge applications. In ENVELOPE, the workflow of the integrated technologies (see Figure 30) is the following (see Figure 31 for the sequence diagram):

- The experimenter inputs a high-level intent (Intent Engine) related to malfunction detection sensitivity
- OBD data is transmitted to the Decision Engine periodically using MQTT
- The Decision Engine (trained with data indicating normal operation), based on the Intent and the OBD data, performs malfunction detection
- If an accident has occurred, it is reported
- The Decision Engine readjusts itself based on the outcome (the malfunction detection sensitivity is readjusted)

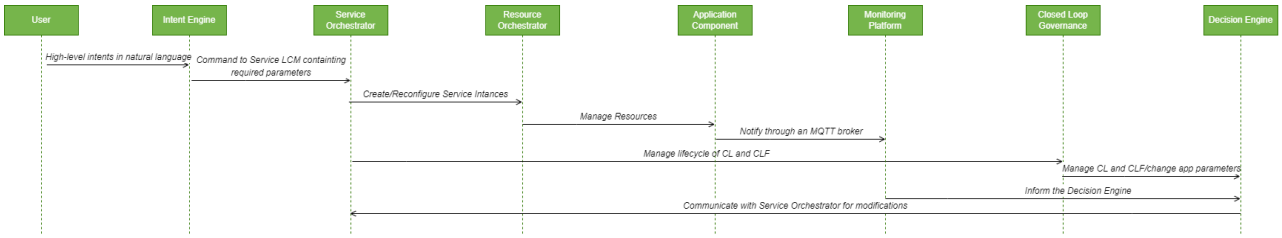

Figure 31 – Sequence diagram for zero-Touch Management solutions.

6.2 Management and Orchestration Layer

The Management and Orchestration Layer enable flexible, efficient deployment of services. It brings together various components to build an integrated framework capable of coordinating computing and network resources, along with the services deployed on virtualized environments. This enables the automated deployment, scaling, and optimization of applications.

The ENVELOPE Management and Orchestration Layer acts as a modular system designed to guarantee that both network and computing infrastructures can satisfy application needs when a user requests the instantiation of an application to run an experiment through the ENVELOPE Experimentation-as-a-Service Module.

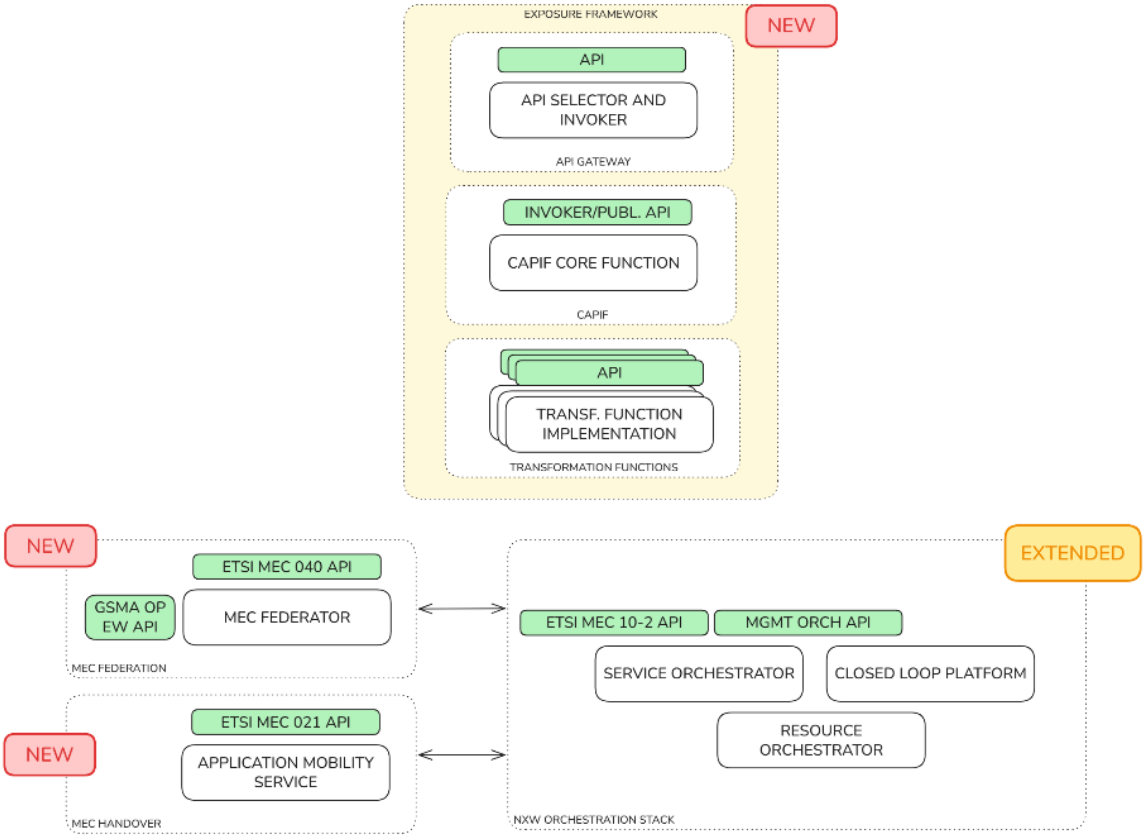

Figure 32 – Management and Orchestration Layer architecture.

Figure 32 depicts the high-level architecture of the Management and Orchestration layer, which consists of integrated components that automate 5G network and service operations. Part of these components, as described below, have been inherited (and in some case enhanced and extended in ENVELOPE) from existing software prototypes. Others, like Exposure Gateway, the MEC Federator and the Application Mobility Service have been developed from scratch in ENVELOPE.

Service Orchestrator. The component that manages the end-to-end orchestration of services, coordinating resource allocation for application functions (through the Resource Orchestrator), creating and configuring network slices, setting up service components, and activating monitoring tasks. Through its northbound interface, it provides various APIs that define vertical service blueprints and descriptors (VSBs/VSDs) using proprietary models. It also supports runtime service management, enabling both manual and automated reconfiguration and scaling. Automation workflows are executed by Closed Loops (CLs), which are dynamically instantiated and governed by the CL Governance component. The Service Orchestrator has been inherited from the solution developed in the SNS JU ADROIT-6G project, and extended to introduce:

- A new ETSI MEC-based REST interface has been developed, embedding the existing logic of the component to support Application Package/Instance lifecycle management (ETSI MEC 010-2).
- The required mechanisms to interact with the MEC Federator (ETSI MEC 040 REST client) have been implemented to support federation scenarios.
- The required mechanisms to interact with the Application Mobility service (ETSI MEC 021 REST client) have been implemented to support the mobility of applications between edge servers.

The Service Orchestrator supports the API operation described in Table 39

Table 39 – Service LCM API Endpoints

Resource name	Resource URI	HTTP method	Description
Service Instances	/api/v1/service-instance-lcm/	POST	Create a new Service Instance.
Service Instance instantiation	/api/v1/service-instance-lcm/{serviceInstanceId}/instantiate	POST	Instantiate the specified Service Instance.
Service Instance scaling	/api/v1/service-instance-lcm/{serviceInstanceId}/scale	POST	Scale the specified Service Instance.
Service Instance reconfiguration	/api/v1/service-instance-lcm/{serviceInstanceId}/reconfigure	POST	Reconfigure components of the specified Service Instance.
Service Instance modification	/api/v1/service-instance-lcm/{serviceInstanceId}/modify	POST	Modify properties of the specified Service Instance.
Service Instance migration	/api/v1/service-instance-lcm/{serviceInstanceId}/migrate	POST	Migrate components of the specified Service Instance.
Service Instance termination	/api/v1/service-instance-lcm/{serviceInstanceId}/terminate	POST	Terminate the specified Service Instance.
Service Instance details	/api/v1/service-instance-catalogue/{serviceInstanceId}	GET	Get information about the specified Service Instance.
Service Instance deletion	/api/v1/service-instance-catalogue/{serviceInstanceId}	DELETE	Delete the specified Service Instance.
Service Instance list	/api/v1/service-instance-catalogue/	GET	Get a list of Service Instances.

Resource Orchestrator. It manages resource orchestration across multi-cluster infrastructures spanning the entire edge to cloud continuum, extending to extreme edge devices. It integrates with IoT platforms and supports heterogeneous nodes with varying characteristics, such as mobility, sensing, and actuation capabilities, as well as diverse operational constraints including compute capacity and energy consumption. These capabilities and constraints are semantically described to enable intelligent resource management. The Resource Orchestrator interfaces with platforms like OpenStack and Kubernetes distributions (e.g., K8s, K3s), while remaining extensible through additional southbound drivers to support other platform types. The Resource Orchestrator has been inherited from the solution developed in the SNS JU Hexa-x-ii project.

The Resource Orchestrator is internally organized in several modules, which interact within each other's and exposes different functionalities:

- **Platform Manager:** it handles registration of edge and cloud platforms, deployment of applications on them, and retrieval of platform and application status and metadata. It exposes the API endpoints described in Table 40 – Platform Manager API Endpoints

Table 40 – Platform Manager API Endpoints

Resource name	Resource URI	HTTP method	Description
Application update	/api/v1/platform-manager/update	POST	Update an Application on a managed platform.
Application status	/api/v1/platform-manager/status	POST	Get the status of an Application on a managed platform.
Platforms list	/api/v1/platform-manager/platforms	GET	Get information about all managed platforms.
Platform creation	/api/v1/platform-manager/platforms	POST	Add a new platform.
Application deletion	/api/v1/platform-manager/delete	POST	Delete an Application from a managed platform.
Platform info	/api/v1/platform-manager/platforms/{platformInfoId}	GET	Get information about a specific platform.
Platform removal	/api/v1/platform-manager/platforms/{platformInfoId}	DELETE	Remove a specific platform.

- **Service Deployer:** it manages the deployment lifecycle of services, including creating, updating, deleting, and monitoring service deployments and their associated runtime information. It exposes the API endpoints described in Table 41

Table 41 – Service Deployer API Endpoints

Resource name	Resource URI	HTTP method	Description
Service update	/api/v1/service-deployer/update/{deploymentInfoId}	POST	Update a deployed Service.
Service deployment	/api/v1/service-deployer/deploy	POST	Deploy a new Service.
Service deletion	/api/v1/service-deployer/delete/{deploymentInfoId}	POST	Delete a deployed Service.
Service status	/api/v1/service-deployer/status/{deploymentInfoId}	GET	Get the status of a deployed Service.
Deployment info search	/api/v1/deployment-info	GET	Find deployment information.

Deployment info details	/api/v1/deployment-info/{deploymentInfoId}	GET	Get deployment information by ID.
Deployment info deletion	/api/v1/deployment-info/{deploymentInfoId}	DELETE	Delete deployment information by ID.

- **Service Template catalogue:** it stores and manages reusable service templates, offering upload, retrieval, download, and removal of templates by name, version, or identifier. It exposes the API endpoints described in Table 42

Table 42 – Service Template catalogue API Endpoints

Resource name	Resource URI	HTTP method	Description
Service Template list	/api/v1/service-template-catalogue/templates	GET	Get all Service Template info entries.
Service Template upload	/api/v1/service-template-catalogue/templates	POST	Upload a Service Template.
Service Template info (by name and version)	/api/v1/service-template-catalogue/templates/{templateName}/{templateVersion}	GET	Get Service Template info by name and version.
Service Template download (by name and version)	/api/v1/service-template-catalogue/templates/{templateName}/{templateVersion}/template	GET	Download a Service Template by name and version.
Service Template info (by ID)	/api/v1/service-template-catalogue/templates/{serviceTemplateId}	GET	Get Service Template info by ID.
Service Template removal (by ID)	/api/v1/service-template-catalogue/templates/{serviceTemplateId}	DELETE	Remove a Service Template by ID.
Service Template download (by ID)	/api/v1/service-template-catalogue/templates/{serviceTemplateId}/template	GET	Download a Service Template by ID.

- **Resource Manager:** it maintains an inventory of clusters and nodes, enabling creation, querying, deletion, and bulk cleanup of cluster and node records across the infrastructure. It exposes the API endpoints described in Table 43

Table 43 – Resource Manager API Endpoints

Resource name	Resource URI	HTTP method	Description
Cluster list	/api/v1/resource-inventory/clusters	GET	Get all clusters.
Cluster creation	/api/v1/resource-inventory/clusters	POST	Add a new cluster.
Node list	/api/v1/resource-inventory/nodes	GET	Get all nodes.
Node info	/api/v1/resource-inventory/nodes/{nodeId}	GET	Get information about a specific node.
Cluster info	/api/v1/resource-inventory/clusters/{clusterId}	GET	Get information about a specific cluster.
Cluster deletion	/api/v1/resource-inventory/clusters/{clusterId}	DELETE	Delete a specific cluster by ID.
Cluster nodes	/api/v1/resource-inventory/clusters/{clusterId}/nodes	GET	Get nodes of a specific cluster.
Cluster node info	/api/v1/resource-inventory/clusters/{clusterId}/nodes/{nodeId}	GET	Get information about a specific node in a cluster.
Cluster truncate	/api/v1/resource-inventory/truncate	DELETE	Truncate all clusters

Closed-Loop Platform. It enables full automation of network and service management through the dynamic provisioning, configuration, runtime updating, and termination of closed-loop (CL) functions. Following the ETSI ZSM framework, it provides an end-to-end, AI-driven environment for zero-touch automation across future network and service domains. Closed-loops consist of functional components that continuously exchange data to analyze system behavior, make decisions, and execute actions toward a desired state.

The framework governs the instantiation and lifecycle of cloud-native CL functions, monitoring, analysis, decision, and execution, deployed as containers across the edge to cloud continuum under the control of the Resource Orchestrator. CLs may employ AI/ML methods for analysis and decision-making. Their deployment and configuration follow Closed-Loop Descriptors (CLDs), extending 3GPP data models with additional parameters for dynamic lifecycle management.

Overall, it provides a governance framework for customizable, cross-layer, zero-touch CL functions aligned with ETSI ZSM and 3GPP specifications. The Closed Loop platform has been inherited from the solution developed in the SNS JU Hexa-x-ii project.

The Closed loop platform supports the API operation described in Table 44.

Table 44 – Closed loop platform API Endpoints

Resource name	Resource URI	HTTP method	Description
Closed-loop update	/api/v1/closed-loop/update/{closedLoopInstanceId}	POST	Update the specified closed-loop instance.

Closed-loop termination	/api/v1/closed-loop/terminate/{closedLoopInstanceId}	POST	Terminate the specified closed-loop instance.
Closed-loop instantiation	/api/v1/closed-loop/instantiate/{closedLoopInstanceId}	POST	Instantiate the specified closed-loop instance.
Closed-loop instances list	/api/v1/closed-loop/instances	GET	Get all closed-loop instances.
Closed-loop instance creation	/api/v1/closed-loop/instances	POST	Create a new closed-loop instance.
Closed-loop instance details	/api/v1/closed-loop/instances/{closedLoopInstanceId}	GET	Get information about the specified closed-loop instance.

Application Mobility Service (AMS). It supports the handover of application between edge servers. This service supports the migration or handover of running application instances or related session contexts from one edge server to another. This is crucial for maintaining low latency, service continuity, and quality of experience in scenarios where users or devices move geographically between edge coverage areas.

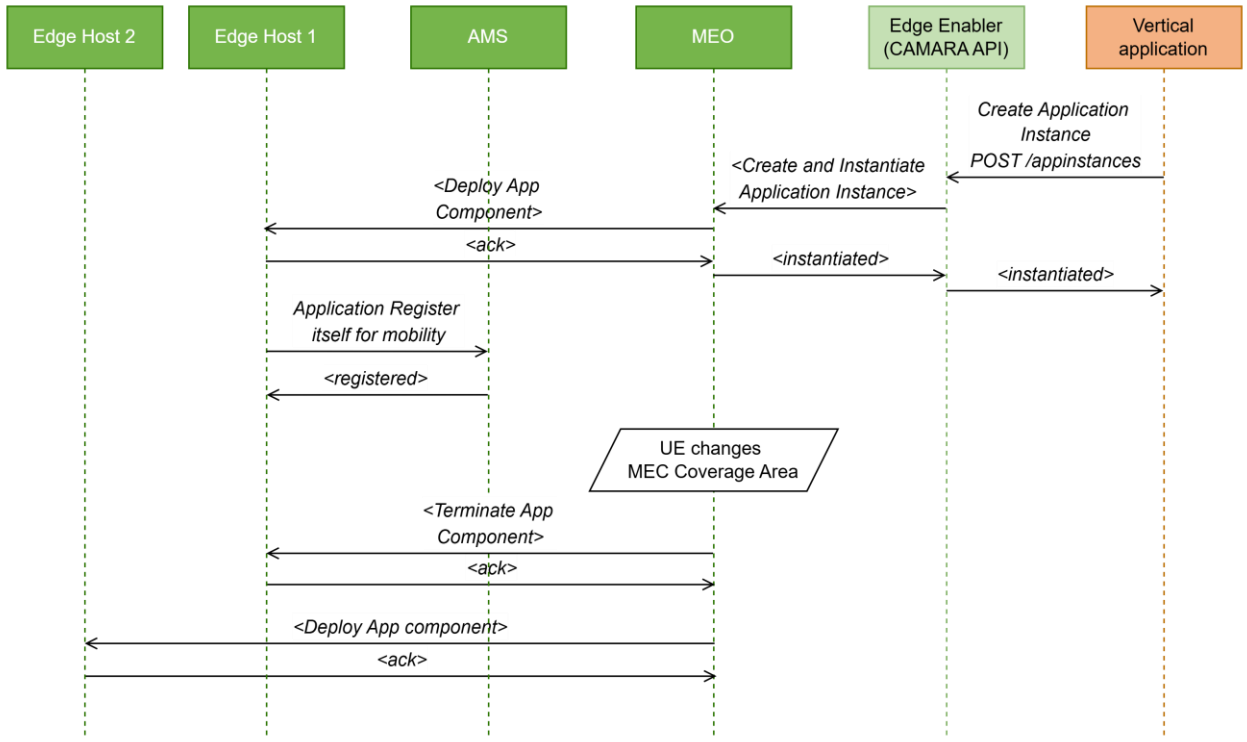

Figure 33 – Interaction between the Vertical Application and the Edge Enabler to create an application instance and orchestrate its relocation between MEC hosts in response to UE mobility.

The AMS exposes the following API endpoint (in Table 45).

Table 45 – AMS API Endpoints

Resource name	Resource URI	HTTP method	Description
Application Mobility Services	/app_mobility_services	POST	Creates a new Application Mobility Service resource indicating that a given application should be migrated if needed.
Application Mobility Services list	/app_mobility_services	GET	Queries existing Application Mobility Service resources.

Figure 33 illustrates the dynamic end-to-end flow for MEC application handover in response to user equipment (UE) mobility across MEC coverage areas. The process is initiated by a Vertical Application requesting the creation and instantiation of an app instance via the Edge Enabler using the CAMARA API. The Edge Enabler coordinates with the MEC Orchestrator (MEO), which triggers deployment of the app component on the designated Edge Host (e.g., Edge Host 1). After confirmation of successful deployment ("ack"), the application is live at the initial edge location. When the UE moves to a new coverage area, a mobility event to the MEO orchestrates the termination of the app component on the source edge host and coordinates its redeployment on the target edge host (Edge Host 2), with acknowledgments for each operation.

MEC Federator. The component to enable MEC Federation (whose service is described in section 5.2). It operates as the interoperability layer between multiple heterogeneous MEC domains, allowing discovery, registration, and consumption of edge services across administrative boundaries. It exposes standardized northbound and southbound APIs aligned with ETSI MEC specifications to support inter-MEC communication, service advertisement, and resource sharing.

Exposure Framework. The Exposure Framework provides secure and standardized access to network and computational resources through open APIs. It serves as the interface between application developers and the underlying 5G/6G infrastructure, enabling application deployment, orchestration, and QoS management across diverse domains.

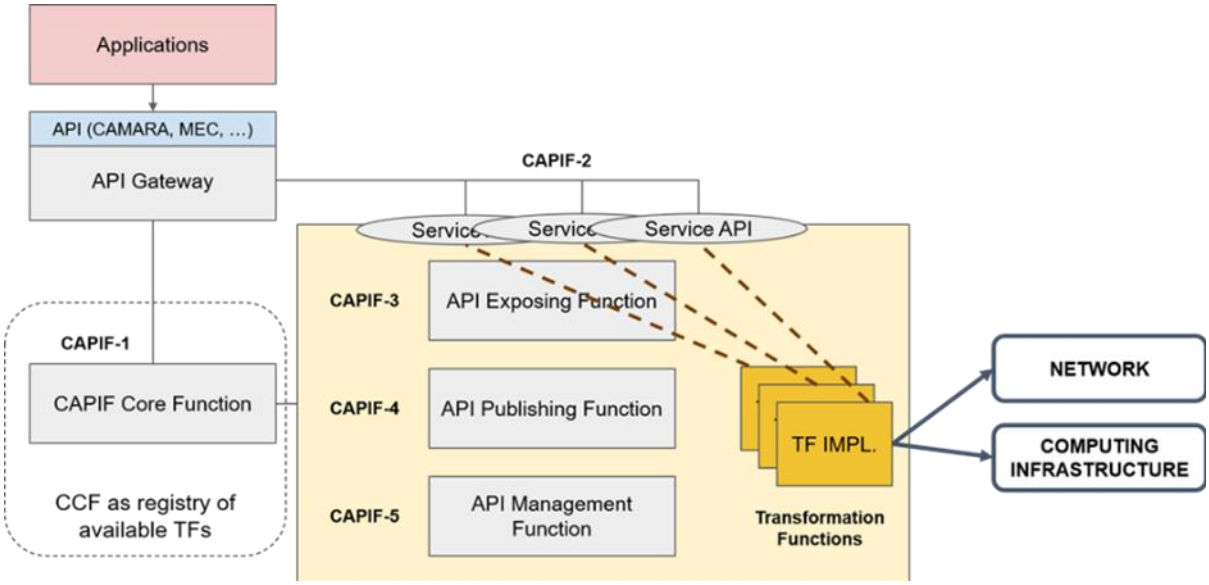

Figure 34 – Exposure Framework architecture.

At the architectural level (Figure 34), the framework builds upon the CAPIF (Common API Framework) specification to manage a registry of available transformation functions and exposed APIs. Its main components include:

- **Applications:** Consumer-side entities interacting with the network via open APIs (e.g., CAMARA, MEC).
- **API Gateway**: The unified entry point for all application requests, responsible for routing them to the appropriate backend service through standardized API interfaces.
- **CAPIF Core Function:** Provides core capabilities such as API discovery, authentication, authorization, and usage control.
- **API Exposing Function (CAPIF-3):** Publishes service APIs offered by backend Transformation Functions (TFs).
- **API Publishing Function (CAPIF-4) and Management Function (CAPIF-5)**: Handle service registration, cataloging, and lifecycle management.
- **Transformation Functions (TFs):** Backend services that abstract and simplify access to network and computing infrastructure.

When an API request is received from an application, the API Gateway determines the most suitable Transformation Function to process it. This selection is context-aware and based on several factors, including user identity or role, request context (such as location or network conditions), the specific API being invoked, and the capabilities or constraints of the underlying infrastructure. To make this determination, the API Gateway interacts with the CAPIF Core Function, which maintains an up-to-date registry of all available TFs, including their metadata, policies, and service-level attributes. This dynamic orchestration ensures that each request is routed to the TF instance that best satisfies the application's requirements and the current network environment.

The Exposure framework is also composed of two SDKs that allows developers to easily deliver CAPIF enabled transformation functions and API Gateways.

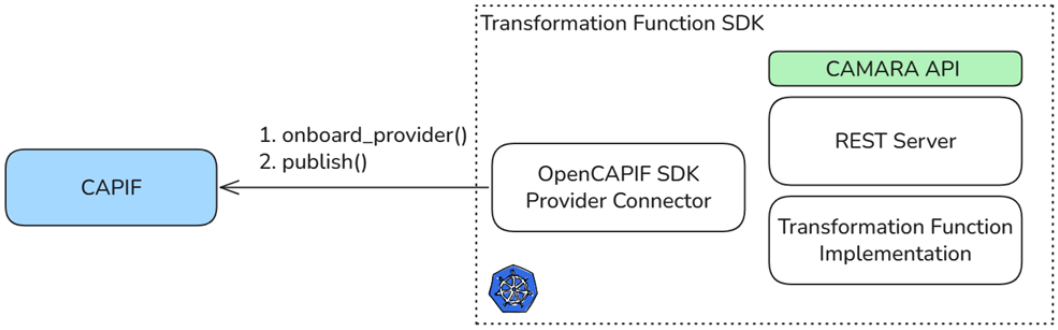

Figure 35 – Transformation functions SDK.

The Transformation Function SDK (depicted in Figure 35) provides a ready-to-use toolkit for implementing CAMARA-compliant Transformation Functions and integrating them with a CAPIF-based exposure environment. It bundles an OpenCAPIF SDK Provider Connector that handles provider onboarding and API publishing towards the CAPIF core, abstracting security, registration, and lifecycle procedures from the developer. On the application side, the SDK offers a REST server where APIs are exposed and bound to the underlying Transformation Function implementation, which encapsulates the business logic and southbound interactions with network and computing resources. This packaging enables developers to focus on service logic while ensuring their functions are automatically discoverable, manageable, and securely exposed through CAPIF in cloud-native (e.g., Kubernetes-based) deployments.

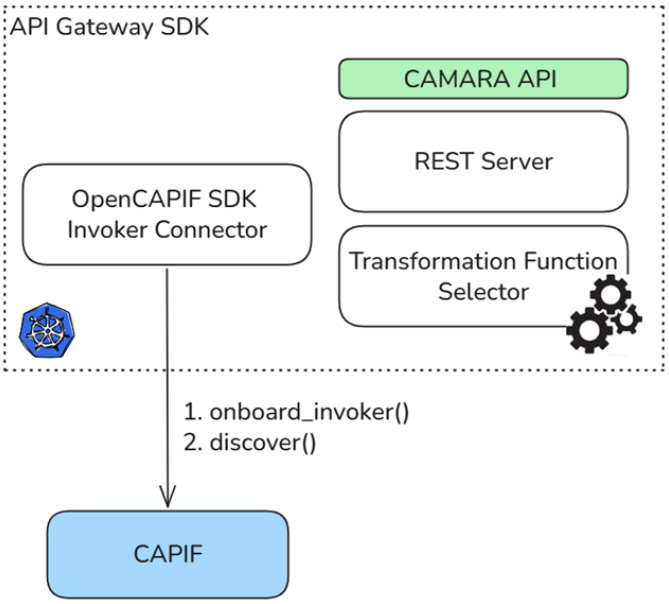

Figure 36 – API Gateway SDK.

The API Gateway SDK (shown in Figure 36) offers a complete toolkit for building CAPIF-aware API gateways that can securely consume services exposed through the Common API Framework. It

embeds an OpenCAPIF SDK Invoker Connector that automates the onboarding of the gateway as an API invoker and manages the discovery of published service APIs, including handling certificates, tokens, and security context required by CAPIF. On top of this foundation, the SDK provides a REST server where APIs are exposed and a Transformation Function Selector that resolves, at request time, which backend Transformation Function instance should be invoked based on service metadata and context. This design allows developers to focus on request routing and policy logic while ensuring that API invocation, discovery, and access control remain compliant with CAMARA and CAPIF specifications in cloud-native deployments.

6.3 Intent Engine

The ENVELOPE Intent Engine is responsible for processing, reasoning, and fulfilling intents related to zero-touch decisions. An intent engine is a critical component of Intent-Based Networking (IBN), i.e., an autonomous paradigm evolution in network and service management. The Intent Engine module enables users to express service requests in high-level intent representations (following TMF 921 standard – part of the TMF IDAN Catalyst project[9]) streamlining network operations and translating user intent into actionable tasks. The actions based on the Intents are handled by the respective Intent Handler (see D3.1, section 2.3). Each Intent Handler is part of a respective Decision Engine. The latter will perform some action related to a specific CAM application.

In its initial version[10] the Intent Engine translated high-level natural language in TMF representation using LLM. Then, the TMF document is queried to extract specific values and to create k8s manifests. The entire pipeline was implemented using Python. In ENVELOPE, instead of creating k8s manifests, specific NXW (Management and Orchestration layer) APIs are called to inform the Decision Engine about the malfunction sensitivity threshold (see next subsection). This value overrides the value set internally in the Decision Engine as the default value. Since, the aim is to extract just the threshold value from the intents, the TMF representation is trivialized and hence it is bypassed in the specific experiments involving the Decision Engine. However, this functionality remains at the disposal of interested experimenters (Open Callers) for more complex actions to take place.

[9] T. Forum, “TMF921A Intent Management API profile,” 2022.
[10] J. Fonseca, et al. "Realizing Intent-Driven Network Management with TM Forum Standards." 2024 IEEE 29th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD). IEEE, 2024.

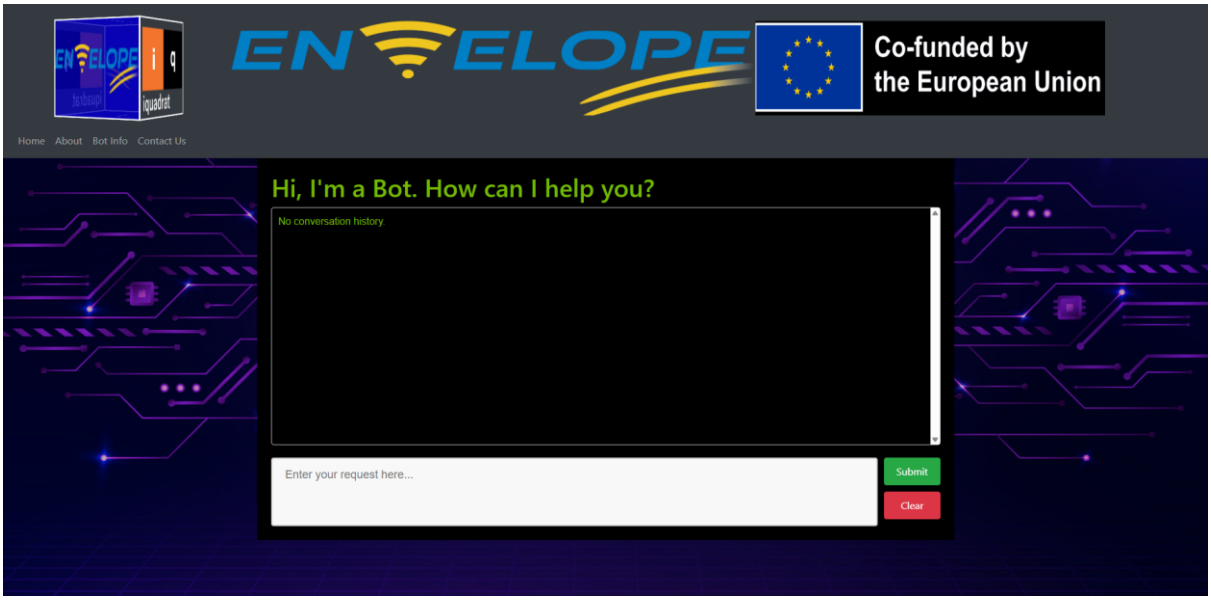

Figure 37 – The Intent Engine graphical interface.

Figure 37 depicts the front-end of the Intent Engine. As mentioned above, the intents are related to the malfunction/anomaly detection sensitivity threshold (THRESHOLD_PERCENTILE environment variable in the Decision Engine – see next subsection). Examples of such intents are presented below:

"I want to set the sensitivity of the malfunction detection to a medium value".

"I'd like you to check for malfunctions using high sensitivity".

"I'd like you to look for accidents using low sensitivity".

6.4 Decision Engine

The Decision Engine is designed to analyze the time series (OBD-II data) that are provided and is particularly effective in recognizing patterns and predicting possible issues based on historical and real-time data. It processes the input data to predict faults and potential failures in the vehicle's subsystems, which is essential for predictive maintenance and early detection of errors.

First, the autoencoder is trained using a data set[11] of a Toyota Etios (year 2014, Engine Displacement1496 CC) related to the vehicle in motion mode in high-speed roads. The data are standardized using StandardScaler (mean=0, variance=1) and the fitted scaler is saved (scaler.pkl) for later use. A feedforward encoder is used; it compiles with Adam optimizer and MSE loss and the model is saved (autoencoder_model.h5).

The Decision Engine runs continuously in a loop to monitor incoming data and assess accidents using the saved model. Then, OBD-II data (from the vehicle in motion mode where some values have been altered) is periodically sent to the decision engine loop via an MQTT broker (the topic used is called "obd/data"). A separate script is responsible for transforming the data into a csv file (called "received_data.csv" which is overwritten periodically). 300 lines of data is sent every 30

[11] https://github.com/eron93br/carOBD

seconds (we assume a high-end approach where 10 lines of data are transmitted every 10 seconds).

The Decision Engine loop is executed and performs malfunction detection every 30 seconds (high-level pseudocode is presented below). In summary, it is a real-time malfunction detection and accident risk monitoring loop for vehicle engine data, using a trained autoencoder and adaptive thresholds. The Decision Engine is a closed loop integrated with the NXW orchestration solution (see Figure 30). The THRESHOLD_PERCENTILE is an environment variable the value of which can be set using values coming from intents (the Intent Engine) and override the current value of the variable. The Decision Engine's function is related to UC-1 and functional requirement It-UC1_FN_4 (Detection Accident/Near miss). The Decision Engine was developed in Python, it was containerized, and a helm chart was created in order to be integrated with the zero-touch management framework (section 6.2).

Decision Engine high-level pseudocode

```
BEGIN PROGRAM

DEFINE CONSTANTS:
    DATA_FILE = "received_data.csv"
    THRESHOLD_PERCENTILE = environment variable "THRESHOLD_PERCENTILE" OR default
"80"
    ACCIDENT_DANGER = 0
    FEATURES = [list of sensor feature names]

LOAD pretrained scaler from "scaler.pkl"
LOAD pretrained autoencoder model from "autoencoder_model.h5"

FUNCTION create_seq(X, window_size):
    RETURN sequences of length window_size from dataset X

FUNCTION detect_anomalies():
    IF DATA_FILE does not exist:
        PRINT "Waiting for file..."
        RETURN

    TRY:
        LOAD data from DATA_FILE

        IF required FEATURES are missing in data:
            PRINT "Missing features"
            RETURN

        SCALE data using loaded scaler
        STORE as X_seq using create_seq

        PREDICT reconstruction using autoencoder
        CALCULATE mean squared error (mse) between prediction and input

        CALCULATE threshold = percentile(mse, THRESHOLD_PERCENTILE)
        FLAG anomalies where mse > threshold
        CALCULATE anomalies_number and anomaly_percentage

        #danger variable value will be used to decide if an accident took
        #place
        IF anomaly percentage >= 15%:
            danger = 1
        ELSE:
            danger = 0

        Decide if accident took place (ACCIDENT_DANGER variable) based on
```

```
        danger variable value and THRESHOLD_PERCENTILE.
        Readjust THRESHOLD_PERCENTILE if needed

    CATCH Exception e:
        PRINT "Error during malfunction detection"

MAIN LOOP:
    PRINT "Starting malfunction detection loop (every 30 seconds)..."
    WHILE True:
        CALL detect_anomalies()
        WAIT 30 seconds

END PROGRAM
```

6.5 Threat Detection Component

A threat-detection engine (which can be viewed as part of the Decision Engine) was designed to assess the monitoring capabilities of the ENVELOPE platform, for any malicious data originating from vehicles directed toward the Intent Decision Engine. The primary objective of this effort is to safeguard the Intent Decision Engine from malicious activities, with a particular focus on identifying Frei, Stau attacks and data poisoning attempts.

Attacks can render the Intent Decision Engine unavailable to the network, disrupting critical functionalities and the overall system's reliability. Meanwhile, data poisoning attacks can corrupt the decision-making process of the Intent Decision Engine by introducing manipulated or inaccurate data, thereby degrading its performance and eroding confidence in its outputs.

The implementation of the threat-detection engine aims to fortify the IDE against these vulnerabilities. The system maintained its availability, integrity, and resilience under adverse conditions by detecting and neutralizing malicious activities in near real-time.

This enhancement significantly increases the trustworthiness and reliability of the Intent Decision Engine by ensuring that its outputs remain accurate and its operations uninterrupted. Additionally, the trust in the overall ENVELOPE ecosystem will be bolstered, as stakeholders can be assured of the system’s ability to withstand and counteract cyber threats effectively.

6.5.1 Methodology and Dataset

In order to train and test the ML model, a dataset consisted of OBD-II vehicle sensor readings in CSV format, covering coolant temperature, manifold pressure, RPM, speed, intake air temperature, mass airflow, throttle position, ambient temperature, and accelerator pedal positions D and E, was used. The dataset was described by the variables presented in Table 46.

Table 46 – Dataset of OBD-II vehicle sensor readings.

Feature	Description
Time	Timestamp of the recorded data.
Engine Coolant Temperature [°C]	Temperature of the engine's coolant in degrees Celsius.
Intake Manifold Absolute Pressure [kPa]	Pressure inside the intake manifold in kilopascals.

Engine RPM [RPM]	Engine speed measured in revolutions per minute.
Vehicle Speed Sensor [km/h]	Speed of the vehicle in kilometres per hour.
Intake Air Temperature [°C]	Temperature of the air entering the engine in degrees Celsius.
Air Flow Rate from Mass Flow Sensor [g/s]	Airflow entering the engine, measured in grams per second.
Absolute Throttle Position [%]	Position of the throttle as a percentage of its full range.
Ambient Air Temperature [°C]	Temperature of the air outside the vehicle.
Accelerator Pedal Position D [%]	Percentage position of the accelerator pedal (sensor D).
Accelerator Pedal Position E [%]	Percentage position of the accelerator pedal (sensor E).

As part of the the database features, were various attack scenarios for the vehicles. More specifically, the two scenarios were Frei and Stau attacks. Specifically:

1. **Frei attacks:** Misguiding navigation through spoofed or manipulated signals
   - **Nature**: Aims to confuse or mislead an autonomous vehicle's navigation system.
   - **Mechanism:** The mechanism is based on sensor inputs manipulation by creating false obstacles or alter the vehicle's route. GPS signals spoofing can be involved, injecting fake road signs, or interfering with object detection algorithms.
   - **Impact:** Can cause AVs to deviate from their path or stop unexpectedly, leading to potential accidents or traffic disruptions.
2. **Stau attacks**: Simulating artificial congestion by falsifying V2V or roadside sensor information.
   - **Nature:** Aims to create **artificial traffic congestion** by **tricking AVs** into behaving abnormally.
   - **Mechanism:**
     - Uses **malicious signals or spoofed sensor data** to **simulate heavy traffic**.
     - May involve hacking **vehicle-to-vehicle (V2V) communication** or **roadside sensors** to make AVs **slow down unnecessarily**.
   - **Impact:**
     - Can **paralyse traffic flow**, affecting transportation efficiency.
     - May be exploited in **ransom-based attacks** or **denial-of-service (DoS) attacks** on smart cities.

6.5.1.1 Data Processing Pipeline

Before we were able to use the dataset, we had to take some preprocessing steps. The aim of the preprocessing is to clean and prepare the data for handling in our ML models. The data processing workflow includes the following steps:

- Removal of Year, Month, Day, Time columns

- Removal of rows with missing values and duplicates.
- Encoding (string to int) - Label Encoding to the specified target column
- Identifies categorical and numerical columns in the DataFrame.
- Standardization of the numerical columns (scales them to have mean = 0 and variance = 1).
- Applies One-Hot Encoding to the categorical columns.
- Shuffles the dataset randomly to ensure the data is not ordered.
- Returns the cleaned and transformed DataFrame.

Following these steps, the pre-processing pipeline ensures methodological consistency across downstream ML tasks.

6.5.1.2 Feature Analysis

Furthermore, due to the large volume of data a representative subset was selected containing normal flows as well as STAU and FREI attacks and validated through statistical comparison (see Figure 38), confirming negligible difference from the full dataset.

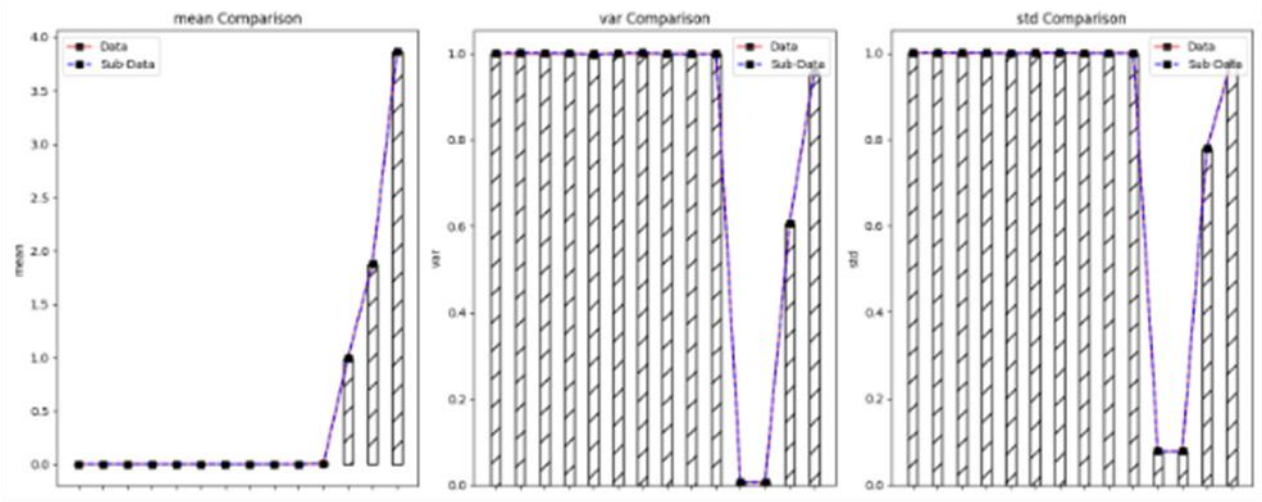

Figure 38 – Statistical metrics for subset validation (left to right: mean, standard deviation, and variation).

Information Gain analysis revealed that Ambient Air Temperature is the most informative feature, with a value of 0.197, indicating it plays a crucial role in distinguishing between classes. It is followed by End (0.086) and Model (0.071), both of which also contribute significantly to the predictive power of the dataset. Moderate information gain is observed in features like Intake Air Temperature (0.060), Start (0.044), Engine Coolant Temperature (0.026), Engine RPM, Absolute Throttle Position, Air Flow Rate, and the Accelerator Pedal Positions D and E, all ranging between 0.017–0.020. Less informative features include Vehicle Speed Sensor and Intake Manifold Absolute Pressure, while Brand has the lowest information gain (0.002), suggesting it contributes very little to the prediction and might be considered for removal or further analysis. Overall, the dataset is influenced more by environmental and trip-specific features than by vehicle identification attributes.

Air Temperature showed the highest mutual information score, close to 0.20, indicating the strongest relationship with the target variable (Figure 39). Following that, End and Model also contribute significantly, with values around 0.08–0.09. Other features, such as Air Temperature,

Start, Coolant Temperature, and Engine RPM, show moderate importance. Features like Throttle Position, Mass Flow Sensor, Pedal Positions D and E, and Vehicle Speed Sensor exhibit lower mutual information, indicating a weaker influence on the target. The least informative features are Absolute Pressure and Brand, both with minimal contribution. Overall, the chart highlights that environmental and trip timing features provide the most predictive value, while vehicle identity and some sensor readings offer comparatively little.

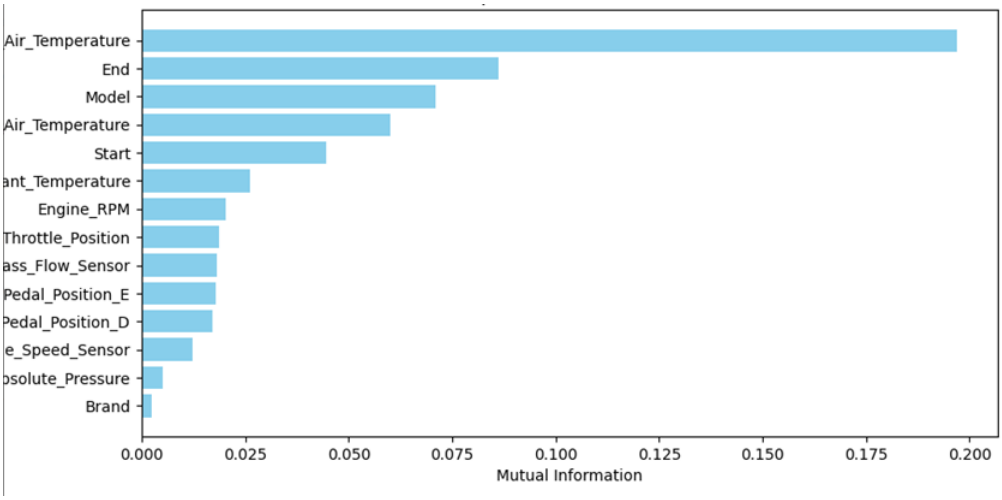

Figure 39 – Feature importance scoring bar chart.

The feature variance analysis shows that most sensor-based features, such as Throttle Position, Engine RPM, and Mass Flow Sensor, exhibit high variance of approximately 1.0, indicating significant variability across the dataset—an essential trait for effective model training. While Ambient Air Temperature has slightly lower variance, it remains highly informative. In contrast, features like Start and End show moderate variance, reflecting some temporal diversity. However, Brand and Model demonstrate extremely low variance (~0.006), suggesting they are nearly constant and contribute little to the model's ability to differentiate between samples, confirming their low mutual information and overall importance.

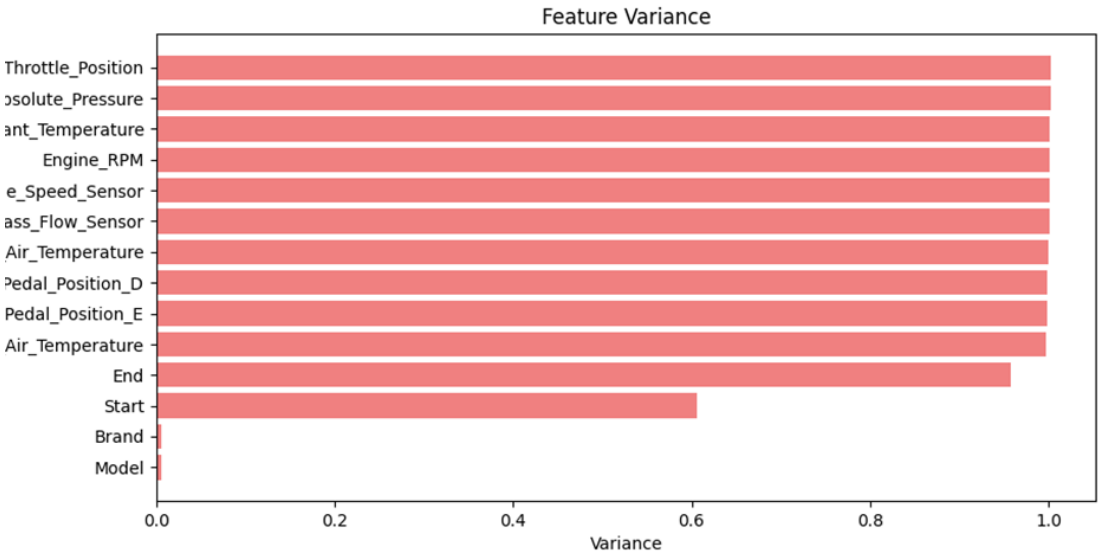

Figure 40 – Feature variance analysis.

6.5.1.3 Machine Learning Model Training for performance testing

Following the data processing and the feature analysis, four machine learning models were evaluated using their default parameters to classify the dataset, and their performance was compared based on accuracy, test error, and detailed classification metrics. The LightGBM model emerged as the top performer with an impressive accuracy of 94.13% and the lowest test error of 5.87%. It demonstrated strong precision and recall across all classes, particularly excelling in identifying class 0 with a recall of 98.3%, and performing reliably on classes 1 and 2 as well. The Neural Network (MLP) also showed strong results, with an accuracy of 92.72%, performing well on class 0 and reasonably on classes 1 and 2, though recall for class 2 was slightly lower (71.5%) compared to LightGBM. On the other hand, Gradient Boosting achieved an accuracy of 85.73%, but struggled notably with class 2, where recall dropped to 34.9%, indicating limited effectiveness in detecting this class using default settings. The Logistic Regression model performed the worst, with an accuracy of only 72.62%, largely because it failed to detect classes 1 and 2, with recall values below 8%, and resulting F1-scores nearly negligible. This highlights that Logistic Regression, in its default form, is inadequate for this multi-class, likely non-linear classification problem. Overall, even without hyperparameter tuning, LightGBM proved to be the most capable model in handling the classification task effectively across all target classes.

6.5.2 Adversarial Attacks

Following the initial training of the models, we have to update the models in order to handle adversarial attacks, which can be present in such systems. Towards this, three adversarial perturbation strategies were studied:

Fast Gradient Sign Method (FGSM): A white-box attack, requiring access to the model's architecture and parameters. The main idea is to perturb the input data by adding a carefully calculated noise that exploits the model's gradients, pushing the input toward a direction that maximises the model's loss function.

DeepFool: An untargeted white-box attack that minimises distortion in adversarial examples. Unlike methods that rely on loss function gradients, DeepFool iteratively approximates the decision boundary as a linear hyperplane and projects the input orthogonally onto it, finding the shortest perturbation needed to cross the boundary.

Gaussian Noise: A statistical perturbation with σ-scaled random noise. A is random noise sampled from a normal (Gaussian) distribution, often with zero mean and some standard deviation σ. An adversarial attack in machine learning is a technique where small perturbations are deliberately added to input data to fool a model into making incorrect predictions.

The flow of actions applied during each attach scenario is summarized in Table 12.

Table 47 – Action flow of each attack type.

Gaussian Noise	FGSM Attack	DeepFool Attack
a. 40% of the data extracted → gauss_data b. Add Gaussian noise (mean=0, std=σ) to gauss_data c. For each numeric column:	a. 40% of the data extracted → fgsm_data b. Apply FGSM attack (epsilon = ε) to fgsm_data using a trained model	a. Take 40% of the data → deepfool_data b. Apply DeepFool attack to deepfool_data using a trained model

i. Standard deviation is computed. ii. Add noise_level*std scaled Gaussian noise to each value. d. Combine with 60% untouched original data → dataset1 = [gauss_data + remaining_data]	c. Train a RandomForestClassifier on the clean data. d. For each sample: i. Approximate the gradient by finite differences using the model's predict_proba. ii. Compute signed gradient and perturbs each feature by epsilon * sign(gradient). e. Combine with 60% untouched original data → dataset2 = [fgsm_data + remaining_data]	i. Train a feedforward neural network (ANNModel) on the training set. ii. Wrap the model using Foolbox. iii. Apply L2DeepFoolAttack to test samples to generate adversarial examples Combine with 60% untouched original data → dataset3 = [deepfool_data + remaining_data]

Following the attack scenarios, the mean value and the standard deviation of original versus the perturbed samples is computed.

The ***FGSM (Fast Gradient Sign Method)*** perturbation results show that only minimal changes are needed to deceive the model, with most features experiencing extremely small perturbations. Interestingly, the feature Absolute_Throttle_Position stands out significantly with a much higher mean difference (5.94e-04) and standard deviation, indicating that the model is particularly sensitive to changes in this feature under FGSM attacks. Similarly, Accelerator_Pedal_Position_D and Accelerator_Pedal_Position_E also exhibit relatively larger perturbations compared to the rest, suggesting their importance in model decision-making. In contrast, categorical features like Brand, Model, Start, and End show negligible differences, implying they contribute very little to the adversarial vulnerability, or were largely unaffected due to their encoding or static nature. Overall, these results suggest that FGSM targets a narrower set of features with more precision compared to DeepFool, and they help highlight the most attack-prone areas of the feature space, guiding where defensive measures like adversarial training should be focused.

The ***DeepFool*** perturbation results reveal how sensitive each feature in the dataset is to adversarial manipulation. Features such as Vehicle_Speed_Sensor, Air_Flow_Rate_from_Mass_Flow_Sensor, and Engine_RPM exhibit the highest mean differences, indicating that only slight changes to these values can significantly impact the model's predictions. This suggests that the model heavily relies on these features when making decisions, making them more vulnerable to adversarial attacks. In contrast, features like Absolute_Throttle_Position and the accelerator pedal positions (D and E) show minimal perturbation, implying they are less influential in the model's classification process. Overall, this analysis can help guide adversarial training efforts by highlighting which features should be prioritised for robustness enhancement, and it also provides insight into the model's implicit feature importance from an adversarial perspective.

The results from applying ***Gaussian noise*** show a consistent pattern of low mean perturbation values combined with relatively high standard deviations across most continuous features. This

suggests that while the average effect of the noise is minimal, individual samples may still experience noticeable variation. Features like Ambient_Air_Temperature, Intake_Manifold_Absolute_Pressure, and Engine_RPM display the highest mean differences, indicating they are more susceptible to Gaussian perturbations and could be more influential in the model's decision-making. In contrast, categorical or static features such as Brand, Model, Start, and End show extremely small mean and standard deviation values, reflecting their limited vulnerability or sensitivity to random noise. Overall, the Gaussian noise introduces a broad, untargeted disturbance, highlighting the model's general resilience to small random fluctuations but also underscoring the need to assess feature-level robustness under noisy real-world conditions.

6.5.2.1 Hyperparameter tuning - Grid Search

The ***dataset*** utilised in this study comprises a diverse set of network traffic flows. It includes normal (benign) flows as well as malicious instances categorised into two primary attack types: Stau and Frei. Furthermore, both attack categories are augmented with adversarial noise variants to simulate more complex and evasive threats. Specifically, Stau and Frei attack samples have been modified using Fast Gradient Sign Method (FGSM) noise, Gaussian noise, and DeepFool perturbations. This enriched dataset aims to evaluate model robustness not only against conventional attacks but also against adversarially manipulated traffic.

This section focuses on identifying the optimal ***hyperparameters*** for three widely used ensemble learning algorithms, XGBoost, Random Forest, and LightGBM, and assessing their classification performance on the dataset. Hyperparameter optimisation is performed using a grid search strategy combined with 5-fold cross-validation to ensure generalizability and prevent overfitting. Model performance is evaluated using several metrics, including macro-averaged F1-score, Accuracy, Cohen’s Kappa, Precision, and Recall. These metrics collectively offer a comprehensive view of the models' effectiveness, covering aspects such as overall correctness, agreement beyond chance, and sensitivity to class imbalances.

The ***XGBoost*** classifier is configured using the multi:softprob objective function, which is tailored for multi-class classification tasks. This objective enables the model to output a probability distribution over all possible classes for each instance, rather than just the predicted class label. This probabilistic output is particularly valuable for scenarios requiring uncertainty estimation or further probabilistic reasoning.

For the ***Random Forest*** model, the hyperparameter grid includes variations in the number of trees (n_estimators ∈ {20, 50}), the maximum depth of trees (max_depth ∈ {10, 30}), the splitting criterion (criterion = 'entropy'), the feature selection method (max_features ∈ {'sqrt', None}), and class weighting (class_weight = 'balanced') to handle class imbalance during training.

Similarly, the ***LightGBM*** classifier is initialised with the multiclass objective, which mirrors the multi-class probability output behaviour of XGBoost. The hyperparameter grid for LightGBM explores values for n_estimators ∈ {20, 50}, max_depth ∈ {10, 30}, learning_rate ∈ {0.01, 0.1}, and includes class_weight = 'balanced' to ensure fair treatment of minority classes. This structured optimisation enables a fair comparison of each model’s strengths under the same evaluation framework.

For each classifier, XGBoost, Random Forest, and LightGBM, hyperparameter tuning was performed using grid search with 5-fold cross-validation. The search aimed to identify optimal model configurations for accurately detecting both benign and adversarially perturbed attack flows

within the dataset. A consistent evaluation framework was applied using accuracy, macro-averaged recall, precision, F1-score, and Cohen's Kappa to assess model performance comprehensively.

XGBoost was evaluated across a grid comprising n_estimators ∈ {20, 50}, max_depth ∈ {10, 30}, and learning_rate ∈ {0.01, 0.1}, resulting in eight candidate configurations and a total of 40 model fits. The best-performing parameter combination was n_estimators=50, max_depth=30, and learning_rate=0.1. On the test set, XGBoost achieved an accuracy of **94.49%**, a recall of **89.25%**, precision of **93.78%**, F1-score of **91.38%**, and a Cohen's Kappa score of **0.8666**. These results demonstrate XGBoost's strong ability to generalize across classes, including those affected by adversarial noise.

Random Forest was tuned over a broader search space with 16 parameter combinations, encompassing n_estimators ∈ {20, 50}, max_depth ∈ {10, 30}, criterion='entropy', max_features ∈ {'sqrt', None}, and class_weight='balanced'. After fitting 40 models, the optimal configuration was found to be n_estimators=50, max_depth=30, criterion='entropy', max_features='sqrt', and class_weight='balanced'. The model achieved a test set accuracy of **94.35%**, recall of **88.78%**, precision of **93.87%**, F1-score of **91.15%**, and a Cohen's Kappa of **0.8626**, reflecting a performance level comparable to XGBoost.

LightGBM was optimised using a grid that included n_estimators ∈ {20, 50}, max_depth ∈ {10, 30}, learning_rate ∈ {0.01, 0.1}, and class_weight='balanced', yielding eight candidate combinations and 40 total fits. The best hyperparameters were n_estimators=50, max_depth=30, learning_rate=0.1, and class_weight='balanced'. Despite its computational efficiency, LightGBM underperformed relative to the other models, with a test set accuracy of **78.84%**, recall of **83.29%**, precision of **67.80%**, F1-score of **72.17%**, and a Cohen's Kappa of **0.5938**. This suggests potential challenges in handling the class complexity or adversarial variations in the dataset.

In summary, both XGBoost and Random Forest delivered high and comparable performance across all metrics, whereas LightGBM, although efficient, demonstrated weaker classification robustness in this multi-class, adversarial detection context.

After selecting the best hyperparameters for each classifier, a **Repeated Stratified K-Fold Cross-Validation** strategy was applied, with **8 splits and 2 repeats**, to assess the final model performance on unseen data. The results are summarised in Table 48.

Table 48 – Final test set performance of the tuned models.

Metric	XGBoost	Random Forest	LightGBM
Accuracy	0.9448	0.9432	0.7864
Recall	0.8923	0.8870	0.8321
Precision	0.9377	0.9390	0.6764
F1-score	0.9136	0.9111	0.7201
Cohen's Kappa	0.8664	0.8620	0.5910

As seen in Table 48, XGBoost slightly outperformed Random Forest across all metrics, particularly in precision and overall agreement (Cohen's Kappa). Both models demonstrated strong robustness

to adversarial noise and class imbalance. In contrast, LightGBM showed significantly lower precision and overall accuracy, suggesting limitations in handling the complexity and variability of the adversarial dataset under the current configuration.

To compare model effectiveness, **XGBoost**, **Random Forest**, and **LightGBM** were evaluated across key performance metrics using repeated stratified cross-validation. Metrics included accuracy, precision, recall, F1-score, Cohen's Kappa, and training time. The results clearly showed that **XGBoost** and **Random Forest** outperformed **LightGBM** in predictive performance across all evaluation criteria.

Both XGBoost and Random Forest achieved high and stable accuracy, with median scores above 0.94. Their recall, precision, and F1-scores were similarly strong, indicating that the models were not only accurate but also consistent in correctly identifying all classes. XGBoost showed a slight advantage in recall, while Random Forest had comparable precision and F1-score. Cohen's Kappa for both models was also high (~0.86), reflecting strong agreement with the ground truth.

In contrast, LightGBM performed significantly worse in predictive metrics. Its accuracy dropped to around 0.79, with lower precision (~0.68) and F1-score (~0.72). Cohen's Kappa also decreased to approximately 0.59, suggesting lower classification reliability. However, LightGBM had the clear advantage in training speed, completing model training in a fraction of the time required by XGBoost or Random Forest—making it highly suitable for time-sensitive applications.

In conclusion, **XGBoost** and **Random Forest** are better suited for scenarios that demand high accuracy and robustness. **LightGBM**, despite lower classification performance, offers exceptional training efficiency and may be preferred when computational speed is critical.

6.5.2.2 Ensemble Stacking

To further optimize our model, a stacked ensemble model was constructed for a multiclass classification problem, leveraging a two-level architecture (see Figure 41). At Level-0, three high-performing machine learning models, XGBoost, Random Forest, and LightGBM, were trained independently on the same training dataset (as explained above). Each of these base models' outputs probability predictions for three classes, resulting in a total of 9 features (3 models × 3 class probabilities). These probabilities serve as inputs to the Level 1 model, forming a compact but rich feature representation of the ensemble's collective opinion.

At Level-1, a simple Artificial Neural Network (ANN) is used as the meta-learner. The ANN takes as input a vector of shape (9,), corresponding to the stacked probability outputs of the base learners. The neural network comprises two hidden layers: the first with 32 neurons and ReLU activation, and the second with 16 neurons and ReLU activation. The output layer uses the softmax activation to produce class probabilities across the three classes. The model is compiled with the Adam optimiser and sparse categorical cross-entropy loss, suitable for multiclass problems where target labels are integers. The ANN is trained for 15 epochs with a batch size of 126, and training verbosity set to a moderate level for tracking progress.

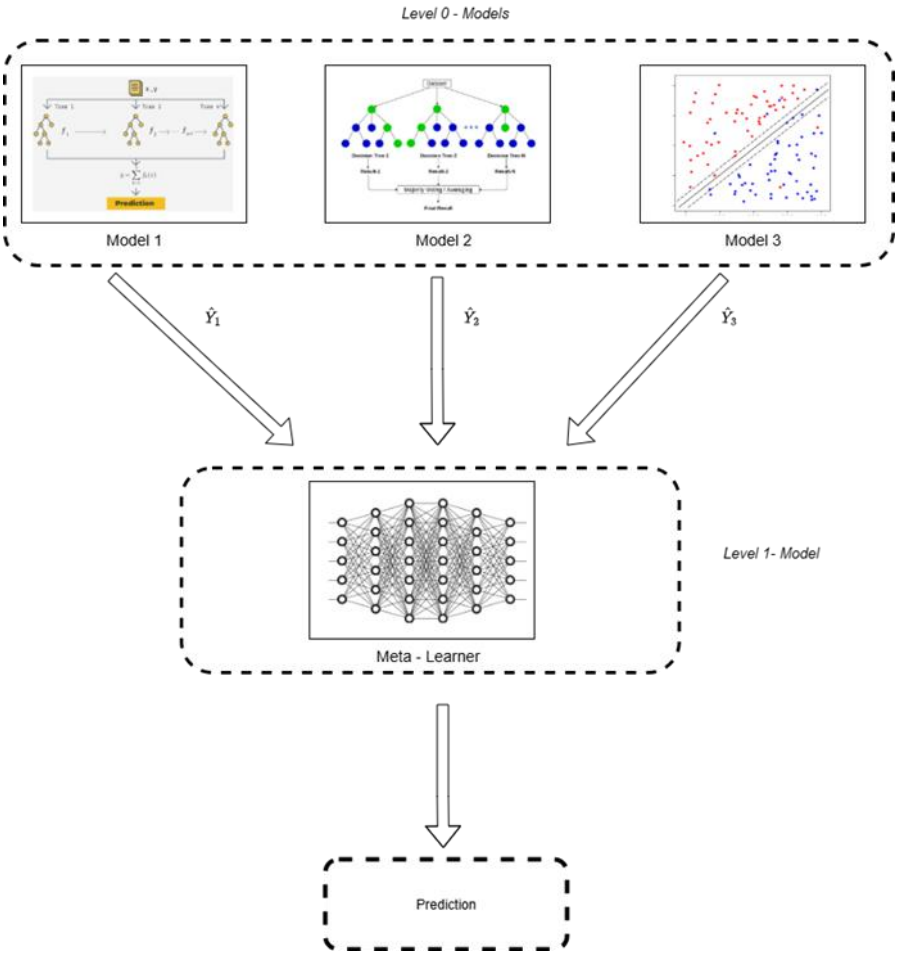

Figure 41 – Ensemble model flowchart.

The ensemble performed exceptionally well. The classification report shows an accuracy of 94.95%, macro-averaged precision of 95.65%, recall of 89.16%, and an F1-score of 92.13%. These scores are supported by the multiclass ROC-AUC curves shown in Figure 42, which indicate an outstanding AUC of 0.99 for each class, meaning that the model almost perfectly separates all three classes. Particularly, class 0.0, which has the largest representation, was predicted with extremely high accuracy, while classes 1.0 and 2.0, though slightly lower in recall, still achieved strong F1-scores of 0.90 each. Moreover, the Cohen's Kappa score of 0.8762 confirms strong consistency between predictions and ground truth, significantly better than random classification.

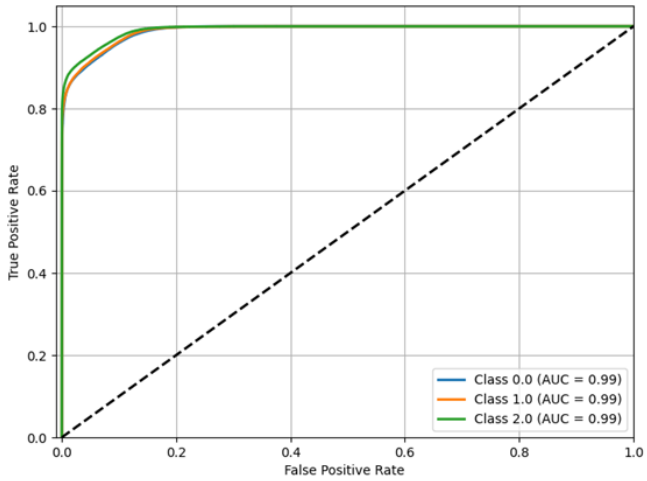

Figure 42 – Multiclass Roc-AUC Curve.

The confusion matrix (see Figure 43) provides a deeper look into how well the model distinguishes between the three classes. Each row of the matrix represents the actual class, while each column represents the predicted class.

For class 0.0, which is the most dominant in the dataset, the model correctly classified 316,641 samples out of 320,291. Only a small number were misclassified as class 1.0 (2,549) or class 2.0 (1,101). This shows the model's very high precision and recall for class 0.0, aligning well with the ROC AUC of 0.99 and f1-score of 0.97.

For class 1.0, out of 78,902 samples, 66,290 were correctly classified. However, 12,234 were misclassified as class 0.0, and a small number (378) as class 2.0. This explains the relatively lower recall (0.84) and slightly lower F1-score (0.90) for class 1.0. It suggests some overlap in feature space between classes 0.0 and 1.0, where the ensemble is more confident in predicting the majority class.

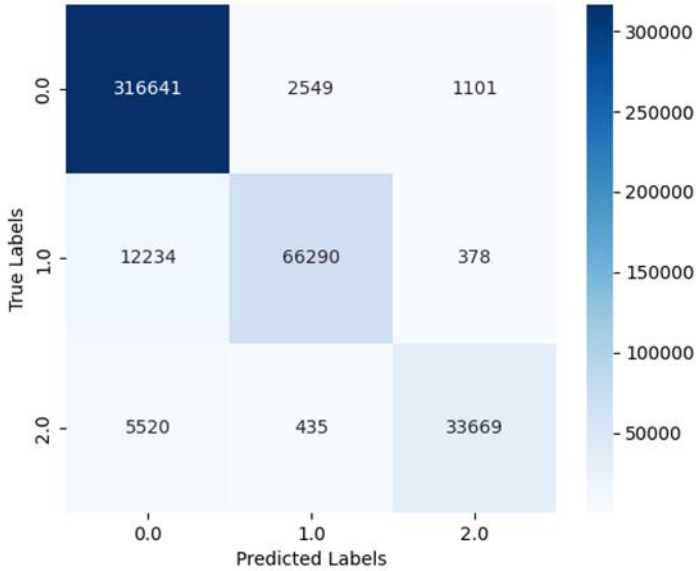

Figure 43 – Confusion Matrix between True and Predicted Labels.

For class 2.0, which has the smallest representation (39,624 samples), the model correctly predicted 33,669 samples. Misclassifications included 5,520 labelled as class 0.0 and 435 as class 1.0. Despite this, the model still maintained a high AUC of 0.99 and an F1-score of 0.90 for this class, indicating effective class separation even under class imbalance.

This matrix highlights that while the model is excellent overall, with high accuracy, AUC, and Cohen's kappa, it still shows some confusion between neighboring or similar classes, particularly from class 1.0 and 2.0 into class 0.0. This could potentially be addressed through class rebalancing, misclassification penalty adjustment, or feature engineering to improve class boundary clarity.

In summary, the confusion matrix reinforces the strong performance observed in your metrics and ROC curves. It also provides actionable insights into class-specific weaknesses that can guide the next steps for fine-tuning the model or preprocessing pipeline. Overall, this architecture shows how combining diverse tree-based models with a neural network meta-learner can yield highly accurate and robust multiclass predictions. While the model already performs at a high level, particularly for the dominant class, slight recall improvements for minority classes could be achieved through techniques such as threshold tuning or cost-sensitive learning. Nonetheless, the stacking framework used here clearly demonstrates its effectiveness on large, real-world datasets.

6.6 Forecasting Component for Capacity Planning

This section details the architecture and contribution of a Forecasting Component designed to enable proactive capacity planning and resource orchestration within the ENVELOPE framework. The primary innovation lies in transitioning from a reactive to a predictive scaling paradigm, ensuring that mission-critical edge applications maintain stringent performance and reliability requirements even under volatile load conditions.

6.6.1 Architectural Overview

The proposed solution is architected as a set of Closed-Loop (CL) functions integrated within the service orchestrator. This system relies on machine learning to forecast demand and autonomously trigger pre-emptive infrastructure adjustments. The high-level architecture, depicted in Figure 44, consists of three core functions: Analysis, Decision, and Execution.

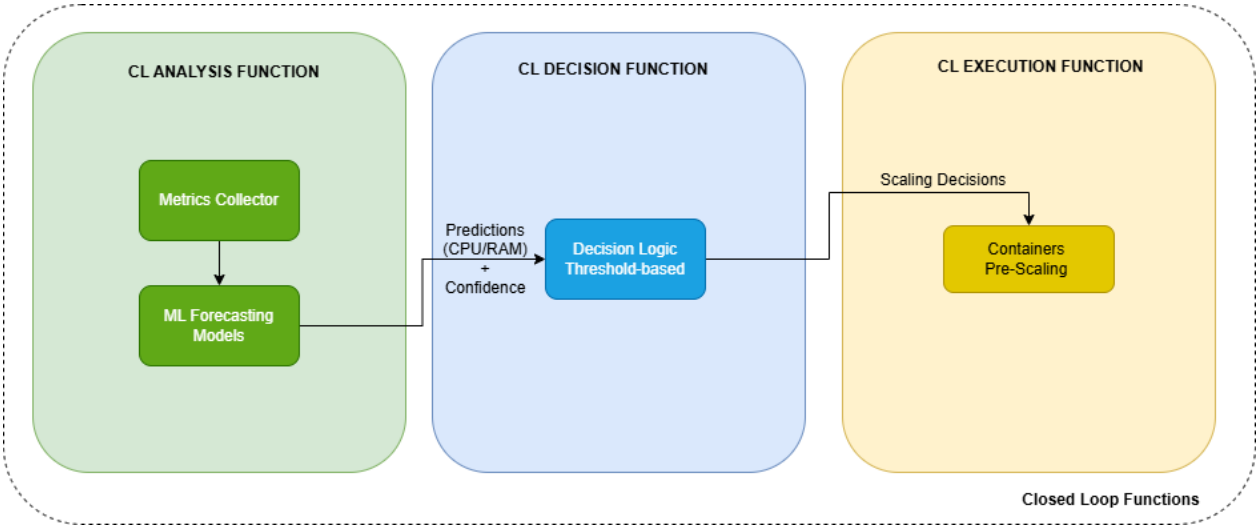

Figure 44 – Closed-Loop Architecture for Predictive Scaling.

The operational flow is as follows:

1. **CL Analysis Function (Forecasting):** This function is responsible for generating predictive insights. It operates on a continuous stream of infrastructure metrics (CPU and RAM utilization of containers) collected by a Prometheus-based monitoring stack. A Machine Learning (ML) forecasting model then analyses this historical and real-time data to predict future resource demand. The model's output includes both the predicted values and a confidence score, providing a probabilistic view of future system states required for proactive decision-making.

2. **CL Decision Function (Logic):** This component consumes the predictions and confidence scores from the Analysis Function. It employs a threshold-based decision logic, to determine if a proactive scaling action is warranted. For instance, if the forecasted resource usage for a service exceeds a predefined threshold with high confidence, a decision to pre-scale is made.

3. **CL Execution Function (Actuation):** This function translates the scaling decisions into concrete orchestration commands. In the context of a Kubernetes (k8s) environment, this results in commands to pre-emptively scale the number of container replicas for a target service, thereby allocating resources before the predicted load materializes.

The entire process is designed for closed-loop automation, with Grafana providing visualization capabilities for monitoring the system's performance and the accuracy of the forecasts.

6.6.2 Contribution and Value in ENVELOPE Use Cases

The contribution of this Forecasting Component is its ability to provide proactive, data-driven orchestration, directly addressing the dynamic resource demands of heavy-edge applications. This is particularly critical for the Italian Trial Site use cases, where delayed resource allocation could lead to service degradation or failure.

The value is demonstrated in two primary scenarios:

- **It-UC1 (Advanced In-Service Reporting - 3D Reconstruction):** When an Autonomous Vehicle (AV) detects an accident, it must immediately transmit large volumes of sensor data for 3D reconstruction. This sudden burst in data traffic can starve critical applications of necessary CPU and RAM if the system is only reactive. The Forecasting Component anticipates this surge in demand. By predicting the need for high-throughput computational resources, it triggers the CL Execution Function to pre-scale the relevant data processing containers proactively. This ensures the infrastructure is prepared to handle the load, guaranteeing reliable and timely data transmission and reconstruction.

- **It-UC2 (Dynamic Collaborative Mapping - HD Map Update Service):** This use case involves continuous dynamic resource allocation for processing HD map updates. A key challenge is the sudden increase in computational load on an edge server when one or more Autonomous Vehicles (AVs) connect to it, such as during a MEC handover. The Forecasting Component analyzes infrastructure metrics to predict these surges in demand. By forecasting the increased CPU and RAM requirements for the map update services on the target server, the CL Decision Function can trigger the proactive scaling of the relevant containers before the AVs' workloads arrive. This pre-emptive action ensures that the service maintains low latency and high reliability during critical mobility events.

Problem Solved: Traditional auto-scaling mechanisms react to load after it has already occurred, creating a window during which services can be starved of CPU, RAM or both. This solution closes that window by forecasting demand and executing the pre-scaling of containers. It empowers the "The Orchestration framework replies to the request of the resource allocation" workflow, as explicitly mentioned in the use case 2 in D2.1, with predictive intelligence, transforming it from a **reactive** to a **proactive** entity.

6.6.3 Summary

In summary, this contribution integrates a predictive closed-loop control system into the orchestration layer. By using infrastructure metrics to forecast demand and trigger pre-scaling actions, it ensures that the stringent requirements of advanced edge applications like 3D reconstruction and collaborative HD mapping are met with high reliability and performance.

6.7 Lessons Learned

The Management and Orchestration layer has been fully deployed within the infrastructure of the Italian site and, for the majority of its primitives as well as its service, network, and edge/cloud resource management APIs, it remains inaccessible to experimenters through direct exposure. However, the zero-touch management framework is tightly coupled with the EaaS to enable programmability and full automation in experiments execution.

The Decision Engine was built on top of the zero-touch management framework to be tightly associated with It-UC-1 (Advanced In-Service Reporting for Automated Driving Vehicles). A limitation is that it does not configure network parameters, only application related variables. The Forecasting Component for Capacity Planning integrates a predictive closed-loop control system into the orchestration layer, forecasting demand and executing the pre-scaling of containers.

The Intent Engine can be used in order to support a variety of applications/decision engines/closed loops that can configure app and/or network parameters. In order for an interested experimenter (Open Caller) to expand it and use it, he/she should:

- Provide the logic/app that will configure specific parameters (related to the network or not)[12].
- Train the Intent Engine using custom intents, with the help of IQU.

[12] For example (as mentioned above) in the context of ENVELOPE, a helm chart was provided for the Decision Engine.

7 Inter-PLMN Handover Capability

7.1 Session and Service Continuity (SSC) modes

The Inter-PLMN Hand-over (HO) capability of the Greek Trial Site is designed to demonstrate the Local Break-Out roaming (LBO) architecture. For latency sensitive applications, LBO establishes a new Protocol Data Unit (PDU) session in the visited network and the SMF and UPF in the visited network handle the user's session management. As the user traffic is anchored to the VPLMN (Visited PLMN), the User Equipment (UE) gets a new IP from the visited network aligned to the VPLMN network plan and the allocated UPF's UE IP Pool.

3GPP defines Session and Service Continuity (SSC) modes to manage session connectivity. Considering the LBO architecture, the **SSC Mode 1** (IP address continuity guaranteed) is not applicable due to the IP changes when moving to the VPLMN UPF. Applicable for the LBO architecture are:

- **SSC Mode 2,** where the IP address changes, but the PDU session remains active and this is enabled through the N16 (SMF-to-SMF) and N4 (SMF to UPF) interfaces. However, the UE needs to handle the new IP address as informed through the NAS Signaling (and the PDU Session Modification Command).
- **SSC Mode 3,** where a new PDU session is established with a new IP and the old session is terminated. By default, SSC Mode 2 follows a *break-before-make* approach which means the old PDU session is released before the new one is established. Alternatively, *make-before-break* can be implemented with SSC Mode 3 to minimize disruption but this requires:
    - UE Capability to handle multiple active PDU Sessions
    - SMF coordination to allow overlapping sessions and
    - Policy control to permit temporary dual sessions

It is important to mention that HPE currently does not support SSC Mode 2 or SSC Mode 3 in its present 5G Core implementation and these capabilities are not listed in the official development roadmap. This means that the system cannot manage an IP address change while keeping the PDU Session active, nor can it create a new session and then smoothly replace the old one. Moreover, SSC Mode 3 (make-before-break), to the best of our knowledge, has not been implemented by any commercial 5G core vendor, nor is it seen on vendor roadmaps as of current releases.

Open5GS, supports SSC Mode 1 by default during PDU session establishment, as evidenced in its NAS signaling traces showing "Selected SSC mode = 0x1 (1)". While the Open5GS SMF codebase includes SSC mode configuration structures and retrieves "ssc mode list" during session context creation and theoretical support for SSC Mode 2 handling is referenced in 3GPP-aligned session re-establishment procedures, practical make-before-break functionality (SSC Mode 3) requiring coordinated dual-session management, overlapping UPF anchor transitions and seamless IP continuity mechanisms, to the best of our knowledge, has not been validated or documented in Open5GS deployments.

In LBO architecture, communication service disruption occurs once the UE IP change triggers session breaks. Known mitigation strategies, including their feasibility in the Greek Trial Site are presented below:

1. SSC Mode 2 and 3 for applications that can tolerate IP change: The HPE Core serving the HPLMN, as mentioned, is not supporting natively SSC Mode 2 and 3, nor make-before-break. From the network perspective, the *break-before-make* experience is implemented, however triggered by the radio HO, through a full session re-establishment process including network reselection (i.e. UE completely loses radio access connectivity with the HPLMN, scans the radio signal quality to determine available networks, selects the most appropriate visited network to camp, establishes connection with the target cell and initiates the user registration and PDU establishment process).
2. Use application-layer resilience. Leveraging the LBO architecture, it is expected that the establishment of the new PDU session must result in steering the user traffic to the appropriate application server at the new edge. The CAM application (involving MQTT and WebRTC traffic) of the Greek site is instead designed to explicitly handle traffic steering, by selecting the appropriate edge server when detecting a network change.

As a result, the Greek site Inter-PLMN HO is focusing on the demonstration of LBO and evaluate the appropriate traffic steering to the edge application servers considering the inevitable service disruption. In this case, LBO and edge application management, enabled through continuum orchestration based on the aeriOS Meta-OS, significantly reduce service disruption at the application layer when changes occur at the network layer, such as PLMN transitions. APIs operating on top of the continuum support both the tailored deployment of application instances across integrated domains and the real-time discovery of deployed instances within the domain that is co-located with the currently serving PLMN.
In the absence of such mechanisms, the OBU handler would be required to manually discover service endpoints or request service access information from operators of the newly entered domain, resulting in increased latency and operational complexity. This manual process would significantly delay service reattachment following a PLMN change.
ENVELOPE-integrated APIs, including CAMARA edge-application management APIs and aeriOS APIs, enable real-time queries that return deployed application instances together with their operational properties. aeriOS, acting as a Meta-OS for the continuum, maintains and federates infrastructure resource capabilities and availability, as well as continuum-hosted service metadata and exposes this information through standardized NGSI-LD interfaces. These APIs allow consistent access to service and infrastructure information across federated domains.
CAMARA edge-application management APIs further simplify this process for application developers by providing abstractions for application onboarding, instantiation and discovery. Applications can query for onboarded application instances deployed in specific zones, where CAMARA zones correspond to aeriOS domains within the continuum. This enables seamless identification of the application instance deployed in the domain associated with the currently serving PLMN.
Given a zone identifier and a reference to the onboarded application, these APIs can report all information required by the OBU-hosted application to reconnect to the appropriate service instance after a PLMN transition. The CAMARA edge-application management APIs return the details of the application instance deployed in the specified zone, where the zone corresponds to the aeriOS domain associated with the currently serving PLMN. Information about the instance deployed in the new zone, mapped to the corresponding aeriOS domain and PLMN, is made available on demand through federated continuum data. This information can then be used to dynamically reconfigure OBU functionality and restore service connectivity.

With this automated approach, end-to-end service rediscovery and reconfiguration can be completed within sub-second timescales. In contrast, without such automation, the same process would typically require on the order of minutes, leading to unacceptable service disruption in latency-sensitive edge and vehicular scenarios.

7.2 Proposed inter-PLMN Handover solution

This section describes the end-to-end Inter-PLMN mobility process realized at the Greek Trial Site, aligning the LBO roaming architecture with the aeriOS edge continuum and the CAM application stack. The process corresponds to the topology depicted in Figure 45 below, where a connected vehicle moves from the COSMOTE campus (Home PLMN) towards the NCSR Demokritos campus (Visited PLMN), while a latency-sensitive CAM service maintains service continuity.

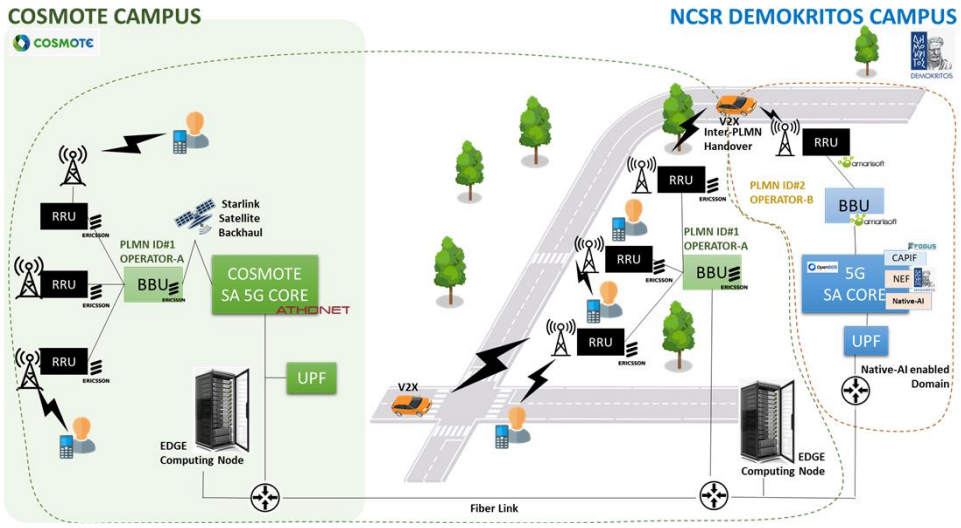

Figure 45 – Overview of the topology at the Greek trial site.

The complete Inter-PLMN mobility process can be summarized as follows:

1. Vehicle enters COSMOTE campus; UE attaches to ENVELOPE-OTE gNB and registers with HPE 5G core.
2. A PDU session is established towards the local HPE UPF; CAM traffic is served from the OTE edge/aeriOS domain.
3. Vehicle moves towards NCSR Demokritos; HPLMN signal gradually degrades according to planned coverage profile.
4. UE performs PLMN selection, discovers ENVELOPE-NCSRD cell, camps on Amarisoft gNB and registers with the Open5GS core. In this setup, PLMN selection can also be explicitly controlled (“forced”) on the UE side by using the Quectel AT+COPS command to dynamically select the target network/operator for attachment, registration and PDU session establishment. By issuing e.g., AT+COPS=1,2,"20202" (**OTE**) or AT+COPS=1,2,"99970" (**NCSRD**), the UE is instructed to attach only to the specified PLMN, thereby enabling deterministic and dynamic network selection. More details will be provided in WP4 delivera-

bles according the manual network selection shown in Section 5 of the 5G modem AT commands configuration options[13]. This is in line with the Quectel chipset capabilities available at the commercial OBU provided by ISFM (see also Section 8) for the autonomous shuttle that will be exploited at the Greek site Trials.

5. According to LBO, a new PDU session is established towards the NCSRD UPF, with a new UE IP address from the VPLMN IP pool.
6. The mobility enabler in the CAM application detects the PLMN/IP change and invalidates previous MQTT/WebRTC connections.
7. The UE maps the current PLMN to the NCSRD aeriOS domain and discovers the local CAM service instance via NGSI-LD and/or CAMARA Edge-Application-Management.
8. New MQTT and WebRTC sessions are established towards the NCSRD edge node using the resolved NODE_IP:NodePort endpoints.
9. The CAM service resumes normal operation, now fully anchored in the Visited PLMN, while the OTE-side application instance may be kept active for other users or gracefully released by the continuum orchestration.

In more detail, aeriOS supports service continuation across domains by ensuring that application state and operational context remain accessible even when a service is re-anchored to a different aeriOS domain due to mobility. In the considered setup, an OBU-hosted application initially operates under **PLMN 20202**, served by the **OTE-domain**, and later roams to **PLMN 99970**, served by the **NCSRD-domain**. The application is assumed to interact only with the local NGSI-LD endpoints (see also Section 5.2.3.2) of the domain currently serving the UE. Continuity is achieved by modelling application state as NGSI-LD entities and synchronizing this state across domains using NGSI-LD federation mechanisms.

At runtime, the application (or a local adapter component) transforms its internal state into a JSON-LD/NGSI-LD representation and persists it to the local Orion-LD Context Broker. In this example, the application state is represented by a globally referenced entity with identifier `urn:ngsi-ld:obu:001` and type `obu`. While the UE is attached to the OTE-domain, the entity is created and updated via the OTE NGSI-LD API endpoint:

Create entity (OTE-domain)

```
POST http://ote_ngsi-ld-endpoint/ngsi-ld/v1/entities
```

```
{
  "id": "urn:ngsi-ld:obu:001",
  "type": "obu",
  "plmn": { "type": "Property", "value": "20202" },
  "servingDomain": { "type": "Property", "value": "ote-domain" },
  "serviceStatus": { "type": "Property", "value": "running" },
  "mqttTopic": { "type": "Property", "value": "telemetry/obu/001" },
  "webrtcSession": { "type": "Property", "value": "sess-9f12" },
  "lastUpdate": { "type": "Property", "value": "2025-12-30T12:00:00Z" }
}
```

[13] https://files.waveshare.com/upload/8/8a/Quectel_RG520N%26RG52xF%26RG530F%26RM520N%26RM530N_Series_AT_Commands_Manual_V1.0.0_Preliminary_20220812.pdf

Subsequent updates are performed locally using the standard NGSI-LD attribute update operation:

```
PATCH          http://ote_ngsi-ld-endpoint/ngsi-ld/v1/entities/urn:ngsi-ld:obu:001/attrs
```

Service continuation across domains are supported by two federation actions, which need to have been configured in advance.

First is the **Context Source Registration** which is established so that each domain is aware of where relevant entity types are managed and can resolve them through the federation layer. For example, to allow the NCSRD-domain to discover obu entities managed in the OTE-domain, the following registration is created on the NCSRD Orion-LD broker:

Context Source Registration (created on NCSRD-domain):

```
POST http://ncsrd_ngsi-ld-endpoint/ngsi-ld/v1/csourceRegistrations
```

```
{
  "id": "urn:ngsi-ld:ContextSourceRegistration:obu:OTE",
  "type": "ContextSourceRegistration",
  "information": [
    {
      "entities": [
        { "type": "obu" }
      ]
    }
  ],
  "endpoint": "http://ote_ngsi-ld-endpoint/ngsi-ld/v1"
}
```

This registration enables federation-aware discovery so that queries issued in the NCSRD-domain for obu entities can be resolved against the OTE-domain when needed.

Second is the **NGSI-LD Subscriptions** which can be configured to propagate entity updates between domains. This ensures that when the application updates its state locally in one domain, the change can be notified and reflected in the peer domain—an essential requirement given that the OBU can only write to local endpoints. For example, to propagate updates from the OTE-domain to the NCSRD-domain, the following subscription is created on the OTE broker:

Subscription (OTE-domain → NCSRD-domain):

```
POST http://ote_ngsi-ld-endpoint/ngsi-ld/v1/subscriptions
```

```
{
  "id": "urn:ngsi-ld:Subscription:obu:OTE-to-NCSRD",
  "type": "Subscription",
  "entities": [
    { "type": "obu" }
  ],
  "notification": {
    "endpoint": {
      "uri": "http://ncsrd_ngsi-ld-endpoint/ngsi-ld/v1/notify",
      "accept": "application/ld+json"
    }
  }
}
```

A symmetric subscription can be configured in the opposite direction (from NCSRD-domain to OTE-domain) to support bidirectional mobility scenarios.

With these federation mechanisms in place, service continuation proceeds as follows. While the OBU is attached to PLMN 20202, it writes updates for `urn:ngsi-ld:obu:001` to the OTE NGSI-LD endpoint. The subscription ensures that updates are propagated to the NCSRD-domain. When the OBU roams to PLMN 99970, the application interacts only with the local NCSRD NGSI-LD endpoint (`http://ncsrd_ngsi-ld-endpoint/ngsi-ld/v1`). Because the entity state has been federated, the application can retrieve and update the latest context locally and continue operation without reinitialisation. Updates performed in the NCSRD-domain are propagated back to the OTE-domain, ensuring that the most recent state is preserved if the OBU returns to PLMN 20202.

This approach provides a standards-based, NGSI-LD-driven mechanism for service continuation across PLMNs and aeriOS domains, preserving data sovereignty and locality while enabling seamless mobility-aware operation across the federated continuum.

7.3 Edge setup

7.3.1 Continuum deployment

The deployed aeriOS-based continuum consists of two fully functional aeriOS domains, hosted respectively at NCSRD and OTE, federated into a single operational environment in accordance with the aeriOS Reference Architecture. Both domains run on top of Kubernetes clusters and expose the full set of aeriOS basic services, enabling orchestration, federation, data management, and self-awareness across heterogeneous infrastructures.

NCSRD aeriOS Domain (Entrypoint Domain)

NCSRD hosts the **EntryPoint Domain**, providing the unified access point for developers and operators via the aeriOS Dashboard, the Identity Management services and the central API exposure layer. The domain is deployed on **three VMs** provisioned via OpenNebula:

- **3× VMs** each with:
  - 2 vCPUs
  - 12 GB RAM
- Public IP exposure enabling external access to the continuum
- Assigned FQDN endpoints:
  - http://api-continuum.front-research-group.eu – aeriOS API exposure
  - https://keycloak.front-research-group.eu – Keycloak IdM
  - https://portal-continuum.front-research-group.eu – aeriOS Dashboard
- All services reachable through public IP **143.233.227.105**

OTE aeriOS Domain

The OTE domain is deployed over OpenStack and consists of:

- **2 VMs, with:**
  - VM1: 4 vCPUs / 12 GB RAM
  - VM2: 4 vCPUs / 16 GB RAM

The OTE domain exposes its APIs internally within the OTE LAN. The LAN is directly routable to NCSRD through a **dedicated dark-fiber interconnection** and programmed routers, enabling full bidirectional reachability and cross-domain API access.

Deployment of aeriOS Services

Both domains deploy the complete suite of aeriOS basic services on top of their Kubernetes clusters, including:

- **Federation services** (NGSI-LD context brokers, registration services)
- **Orchestration services** (HLO and LLO operators, orchestration APIs)
- **Data Fabric services** (NGSI-LD data model exposure and entity lifecycle)
- **Self-x mechanisms** (IE agents reporting resource availability, capabilities, and status)
- **API exposure layer**, identical across domains

Consistent with the aeriOS architectural design, **both domains operate as fully symmetrical peers**: each exposes identical APIs, supports the same orchestration workflows, and is capable of performing informed placement decisions using the shared continuum-wide knowledge graph.

Federation, Security and Identity Management

The NCSRD EntryPoint Domain hosts the aeriOS Identity Management stack:

- **Keycloak** for authentication and token-based access control
- **LDAP** for identity storage and role management

Proper roles and users were provisioned to:

- Query continuum entities using **NGSI-LD APIs**
- Submit multi-domain orchestration requests through **HLO APIs**

Cross-domain federation is enabled by:

- NGSI-LD–based registration and discovery across the two domains
- Continuous IE status reporting inside each domain
- HLO east–west communication for cross-domain allocation
- Secure API reachability through the private routed NCSRD–OTE network

5G Integration and Edge Accessibility

Each aeriOS domain hosts a service which comprises of 2 service components:

- an MQTT component, and
- a WebRTC component.

Each of these components in exposing its API as a K8s nodeport. Thus, the schema to access these APIs is: NODE_IP:K8s_assigned_port. Both information (IP and port) are registered in the continuum and can be acquired using the proper queries (see next section).

The aeriOS domains, including the exposed APIs and the deployed services (exposed as K8s nodeports), are **directly reachable from the 5G UPF within each site**.

As a result:

- **UEs registered in each organization can access aeriOS services locally**, at the edge closest to the UPF.
- The MQTT and WebRTC service components deployed in each aeriOS domain can be consumed by 5G UEs (as K8s nodeports).
- Each aeriOS domain effectively operates as a 5G-accessible edge compute site, supporting real-time interactions through the continuum.

This integration ensures that aeriOS services are seamlessly available to mobile application operating over the local 5G network.

Service Deployment and Operation

Following aeriOS architectural principles, a distributed two-component service was deployed across both domains, consisting of an **MQTT component**, and a **WebRTC component.**

7.3.2 Application instances

Each aeriOS application domain hosts the application components. The protocols that the applications engage are MQTT, WebRTC and HTTPS.

When the UE transitions between PLMNs, it detects the handover event based on changes in its assigned IP address range. After attaching to the new PLMN, the UE (or the mobility enabler acting on its behalf) must re-establish communication with the service deployed in the **target aeriOS domain**. Since aeriOS services are exposed via **Kubernetes NodePorts**, the UE requires two types of information:

1. **The IP address of a Kubernetes node** in the target domain.
2. **The exposed NodePort** of the service component deployed in that domain.

The aeriOS **NGSI-LD APIs** support this discovery procedure through a sequence of structured queries. The main steps are illustrated below:

- **Step 1 – Resolve the Service (obtain the service URN)**
  Query to retrieve the service entity by its logical name.
- **Step 2 – Retrieve ServiceComponents for that Service**
  Query to retrieve all ServiceComponent entities linked to the service.
- **Step 3 – Retrieve the exposed k8s port (NodePort) from NetworkPort**
  For each NetworkPort ID obtained in Step 2, the corresponding entity is retrieved.
- **Step 4 – Retrieve k8s node IPs (InfrastructureElement) in the Domain**
  Obtains the IP addresses of k8s nodes within a particular domain.
- **Step 5 – Construct the NodePort access endpoints**
  For each service component, the NodePort endpoint is constructed.

CAMARA Edge-Application-Management based Approach

A complementary approach to service discovery in the aeriOS continuum involves the use of CAMARA APIs. In this configuration, the NCSRD domain exposes CAMARA-compliant interfaces that allow the application to be onboarded, instantiated per domain and later discovered dynamically during PLMN changes.

The process begins with onboarding the application using an AppManifest, which describes the application's configuration, capabilities, and deployment requirements:

```
{
    "appId": "550e8400-e29b-41d4-a716-446655440000",
    "name": "MyContinuumApp",
    "version": "1.0.0",
    "appProvider": "aeros_dev",
    "packageType": "CONTAINER",
    "appRepo": {
        "type": "PUBLICREPO",
        "imagePath": "docker.io/library/nginx:stable"
    },
    "requiredResources": {
        "infraKind": "kubernetes",
        "applicationResources": {
            "cpuPool": {
                "numCPU": 1,
                "memory": 1024,
                "topology": {
                    "minNumberOfNodes": 1,
                    "minNodeCpu": 1,
                    "minNodeMemory": 512
                }
            }
        },
        "isStandalone": true,
        "version": "1.28"
    },
    "componentSpec": [
        {
            "componentName": "aeros-component",
            "networkInterfaces": [
                {
                    "interfaceId": "eth0",
                    "protocol": "TCP",
                    "port": 9090,
                    "visibilityType": "VISIBILITY_INTERNAL"
                }
            ]
        }
    ]
}
```

This manifest is submitted to the CAMARA onboarding endpoint *http://{{camara_url}}/apps/*. The response includes an *appId*, which is subsequently used to instantiate the application in each relevant aeriOS domain.

For each domain, an AppInstancePost request is issued. This request contains the previously obtained appId and the selected CAMARA *zone*, representing the target aeriOS domain:

```
{
  "name": "envelopeapp",
  "appId": "9c872fda-14b4-448d-9651-857880bb4828",
  "edgeCloudZoneId": "urn:ngsi-ld:Domain:ncsrd01"
}
```

This request is sent to the designated instantiation endpoint *http://{{camara_url}}/appinstances/*. A successful request produces a CAMARA appDeploymentId, which uniquely identifies the application instance within that domain.

When the UE changes PLMN and receives a new IP address, it can determine the serving domain. At this point, instead of directly querying aeriOS, the UE can first query the CAMARA API to obtain the list of instantiated applications across all domains. This is done by invoking the CAMARA endpoint responsible for retrieving instances of the onboarded application *http://{{camara_url}}/appinstances/?appid={{onboarded_appid}}*.

The response returned by this CAMARA API contains a list of application instantiations, each associated with a specific domain:

```
[
    {
        "appDeploymentId": "1174e748-3184-4a13-b77a-f1dd2539a56e",
        "appId": "9c872fda-14b4-448d-9651-857880bb4828",
        "name": "envelopeapp",
        "appProvider": "ContinuumCamaraAdapter",
        "status": "unknown",
        "componentEndpointInfo": null,
        "kubernetesClusterRef": null,
        "edgeCloudZoneId": "urn:ngsi-ld:Domain:ncsrd01"
    }
]
```

From this structure, the UE selects the appdeploymentId corresponding to the domain that matches the current PLMN.

While CAMARA application instance is identified by appDeploymentId the NGSI-LD resources in aeriOS use aeriOS-specific URNs. To align the CAMARA representation with the aeriOS model, the appDeploymentId must be converted to an aeriOS identifier:

```
service_id = f"urn:ngsi-ld:Service:{continuum_id}"
```

Where "*continuum_id*" is retrieved by sending appDeployedid in the following transformation function:

```python
def hash_uuid(uuid_str, hash_length=12):
    """
    Hash a UUID string using MD5 and return the first `hash_length` characters.

    Args:
        uuid_str (str): The UUID to hash.
        hash_length (int): Length of the returned hash (default 12, max 32 for MD5).

    Returns:
        str: A shortened MD5 hash of the UUID.
    """
    md5_hash = hashlib.md5(uuid_str.encode('utf-8')).hexdigest()
    return md5_hash[:hash_length]
```

The output of this function yields the aeriOS Service identifier required for NGSI-LD queries. Once this aeriOS identifier is obtained, the workflow continues from **Step 2** (see above) of the previously defined process (retrieval of ServiceComponents, NetworkPorts, InfrastructureElements, and construction of the NODE_IP:NodePort access endpoint).

8 CAM Application Development

The following subsections present the developments undertaken as part of the WP3 activities related to the CAM applications. For each site, we summarize the main components and the associated development work that has been completed. In WP4 (Deliverable 4.1 and 4.2) the full integrated implementation and functionality will be documented comprehensively.

8.1 Italian Site

The CAM Applications, which have been developed in the context of the Italian trial site, are to support the demonstration of the use case It-UC1 “Advanced In-Service Reporting for Automated Driving Vehicles” and of use case It-UC2 “Dynamic Collaborative Mapping for Automated Driving” that were introduced in Deliverable D2.1.[14] In this section, we first introduce the system architecture of the CAM Applications that have been developed for both use cases. We then detail the individual components of the CAM applications, and lastly, we describe the interaction between the CAM Applications’ components and the APIs.

8.1.1 Architecture

The concept of It-UC1 is to extend the vehicle in-service reporting to also share raw sensor data in case of accidents, besides the set of vehicle status information expected by the current legislation. The scope of the sharing of raw sensor data from vehicles that are nearby an accident is to exploit these data for creating a 3D reconstruction of the accident scene that can be helpful to first responders.

The system architecture of the CAM application for It-UC1, which has been developed in the ENVELOPE project to support the raw sensor data sharing and the 3D scene reconstruction, is depicted in Figure 46, which shows the main interactions among the different components of the CAM application, as well as the interaction with the APIs.

[14] ENVELOPE Deliverable 2.1, “ENVELOPE use cases description and specifications”, https://envelope-project.eu/wp-content/uploads/2024/04/D2.1-ENVELOPE-use-cases-description-and-specifications-Final.pdf

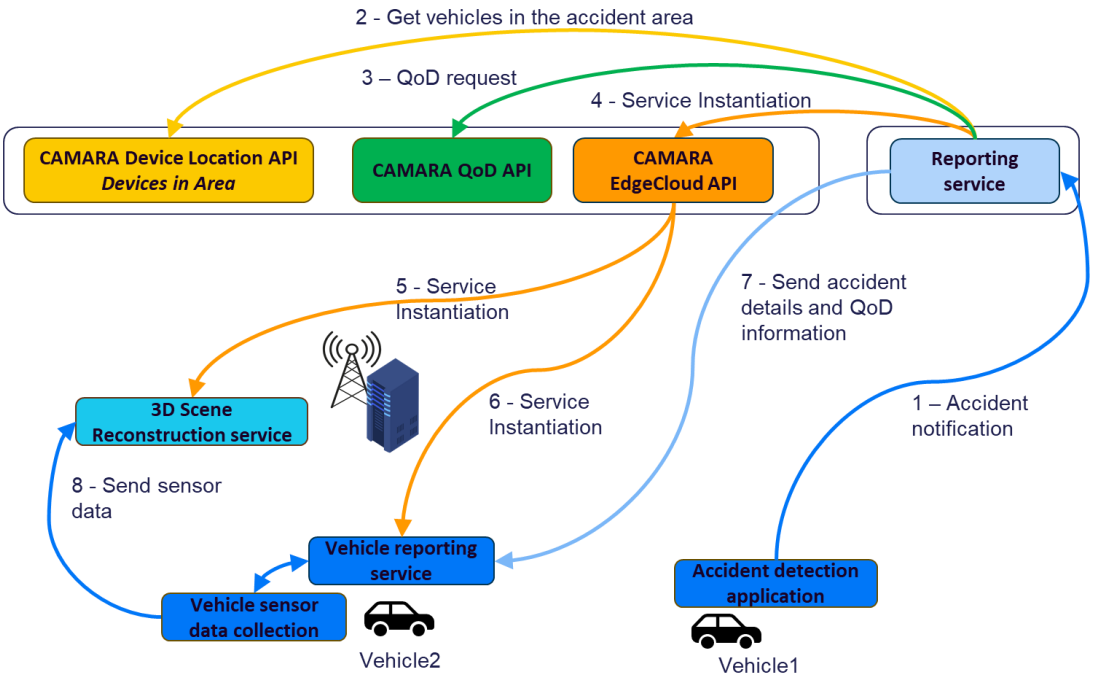

Figure 46 – It-UC1 CAM Application System Architecture.

The CAM Application of It-UC1 comprises the following components:

- *Accident detection application*: it is a service that runs on the vehicle side that detects the occurrence of an accident and notifies it.
- *Reporting service*: it is a service running at the infrastructure side and is responsible for monitoring the occurrence of an accident and taking the required actions to make the In-Service Reporting effectively executed.
- *Vehicle reporting service*: it is an application running at the vehicle side, which manages the interaction with the Reporting Service at the infrastructure side and with other applications on the vehicle side, to enable the vehicle to provide the requested data.
- *Vehicle sensor data collection*: it is the application that interacts with the sensors for gathering the requested raw data, and it performs the data transmission to the edge application.
- *3D Scene Reconstruction service*: it is the service in charge of managing the raw sensor data collection from vehicles and processing them for creating the 3D reconstruction scene.

The focus of It-UC2 is to implement a CAM application that provides a map update service based on a distributed and collaborative approach. The map update service exploits the raw sensor data that is shared from the vehicles that detect modifications in the surrounding environment with respect to the information that they have on board. This service also takes care to inform new incoming vehicles that a map update is available and manages the update process.

Figure 47 and Figure 48 introduce the system architecture of the It-UC2 CAM Application. To highlight the interactions among the components and between the components and the APIs, the map update process and the map download process are represented separately. Specifically, the map update process is shown in Figure 47, and the map download process is shown in Figure 48.

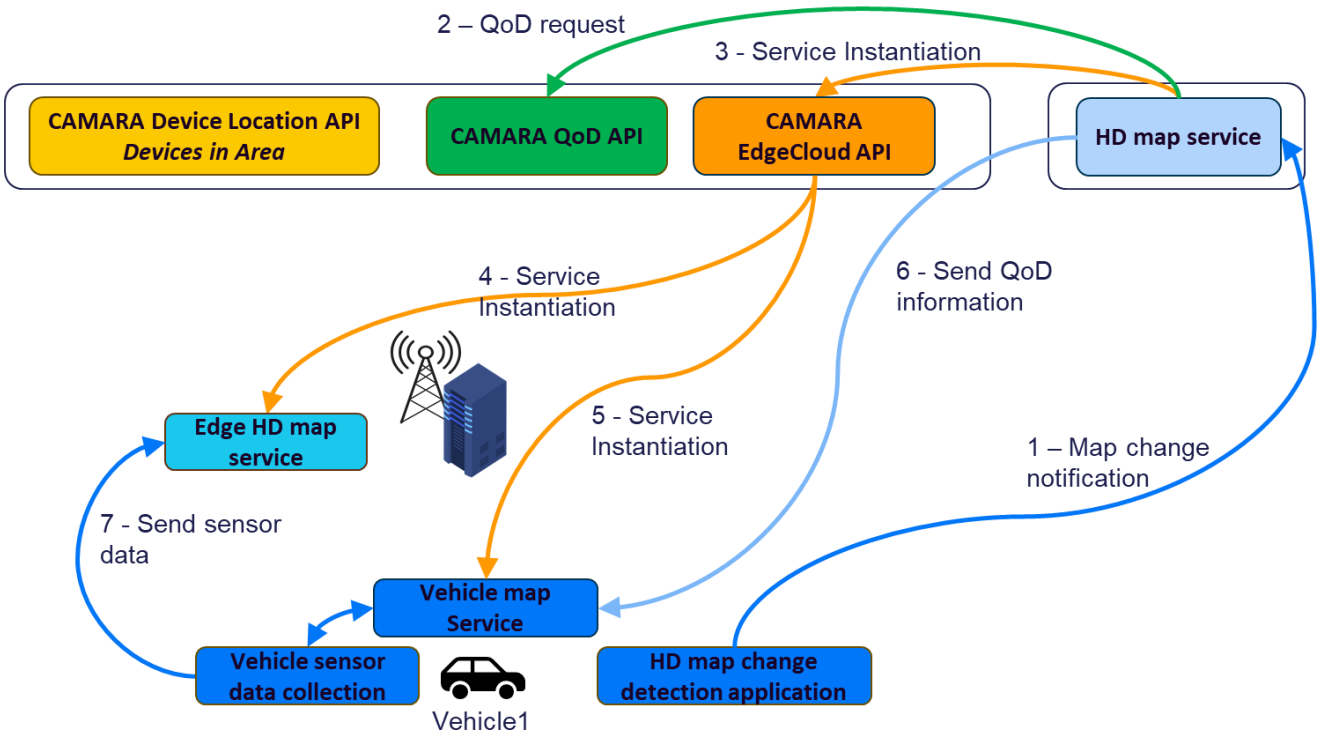

Figure 47 – It-UC2 CAM Application System Architecture – map update generation.

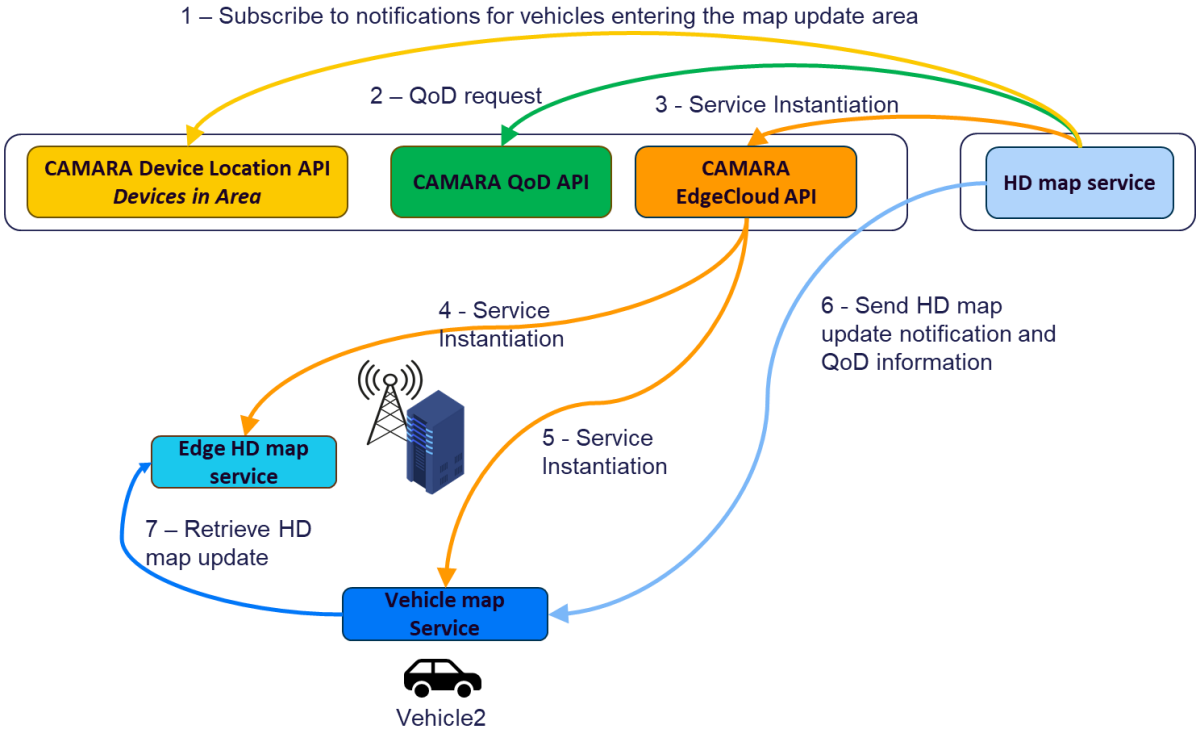

Figure 48 – It-UC2 CAM Application System Architecture – map update download.

The CAM Application of It-UC2 comprises the following components:

- *HD map change detection application*: it is a service running on the vehicle side that detects a change in the environment with respect to the vehicle map knowledge and notifies it.

- *HD map service*: it is a service running at the infrastructure side, it is responsible for receiving map change notifications and taking the required actions to perform the HD map update; it is also responsible for monitoring new incoming vehicles and supporting them in the process of the HD map update download.
- *Vehicle map service*: it is an application running at the vehicle side, which manages the interaction with the HD map service at the infrastructure side and with other applications on the vehicle side, to enable the vehicle to provide the sensor data for the map update.
- *Vehicle sensor data collection*: it is the application that interacts with the sensors for gathering the raw sensor data, and it performs the data transmission to the edge application.
- *Edge HD map service*: it is the service in charge of managing the raw sensor data collection from vehicles and processing them to create the HD map update; it is also responsible as well to managing the retrieval of the HD map update for the new incoming vehicles.

8.1.2 CAM Application Components

The components of CAM Applications for It-UC1 and It-UC2 are introduced in this section, distinguishing them in three categories: i) components running at the vehicle-side, ii) components on the edge server, and iii) components at the infrastructure-side.

8.1.2.1 Vehicle-side components

8.1.2.1.1 Accident detection application

This application component belongs to the It-UC1 CAM application, and it oversees detecting the occurrence of an accident. The detection can be performed by monitoring the dynamics of the vehicle (e.g., sudden deceleration of the vehicle may indicate a crash) or analyzing the sensor data (e.g., processing the video stream of the camera). Testing a specific implementation of the accident detection system under realistic conditions is difficult, as it would require situations that cannot be safely reproduced. A possible implementation is the one introduced by the Decision Engine reported in Section 6.4. In this case, the continuous analysis of OBD-II data can identify accident occurrences thanks to vehicle data monitoring. The accident is therefore simulated to allow a replicable demonstration of It-UC1.

When the application detects the accident, it broadcasts the information using Decentralized Environmental Notification Message (DENM) in compliance with ETSI TC ITS standards. The DENM is a message used in V2X communication to inform V2X players about a specific event. It is possible to indicate the type of event that occurred (e.g., an accident).

The message is sent using an ITS communication software stack that is a background asset of LINKS and is made available for use in the ENVELOPE project. The implementation of this stack complies with the relevant ETSI standards with respect to message payload encoding, structure, and headers.

The communication is performed on the 5G Uu interface, adopting a Vehicle-to-Network approach. The application protocol for the communication is AMQP 1.0, following what has been defined in the context of the C-Roads platform. This approach was introduced and used in previous research projects (e.g., 5G-CARMEN[15]) for exchanging V2X messages on 5G mobile networks.

[15] https://5gcarmen.fbk.eu/

8.1.2.1.2 HD map change detection application

This application component is part of the It-UC2 CAM application, and its scope is to detect variations in the environment (e.g., roadworks or obstructions) with respect to the map information available on the vehicle side.

The change is notified to the HD map service using a RESTful API that is exposed by the infrastructure service. The notification is sent using a POST request whose body contains the details of the change (i.e., the position and the extension of the detected modification in the environment).

The general concept foresees that the first vehicle detecting a structural change compares its local perception-derived representation with the local reference HD map. When a deviation is confirmed, only the affected portion of the point cloud map is extracted and prepared for transmission. This localized update is then reported to the HD map service, which can process the information jointly with those received from other vehicles travelling in the same area and providing as well the updated point cloud map.

8.1.2.1.3 Vehicle reporting service

This application component is included with the It-UC1 CAM application, and it is responsible for managing the In-Reporting Service at the vehicle side. This component is instantiated dynamically on the vehicle when an accident occurs because of the query to the CAMARA Edge Cloud API performed by the Reporting service, which is running on the infrastructure side.

The main role of the Vehicle reporting service is to interact with the Vehicle sensor data collection component and to instruct it about which raw sensor data to provide and how to transmit them. Furthermore, the Vehicle reporting service oversees interacting with the network service at the vehicle side for managing the setup of the network connection for exploiting the QoD resources that are allocated on-demand to the vehicle for sharing the sensor data.

This information is sent by the Reporting service. In the specific, it provides the time window in which the vehicle should provide its raw sensor data, the transfer protocol to be used (i.e., FTPS), the server address and port, the credentials to be used, the file path, and the filename to be used for the sensor data transfer.

The Vehicle reporting service forwards this information to the Vehicle sensor data collection component that actually performs the data transfer. This information is sent with a JSON-formatted message to the internal vehicle network using a UDP socket.

8.1.2.1.4 Vehicle map service

This application component belongs to the It-UC2 CAM application. It is in charge of managing the HD map update on the vehicle side. It is dynamically launched when an HD map update occurs by the HD map service that consumes the CAMARA Edge Cloud API.

The Vehicle map service receives as input the details for the raw sensor data transfer, and it forwards them to the Vehicle sensor data collection in a similar way as performed by the Vehicle reporting service in It-UC1.

Besides this role, the Vehicle map service is responsible for managing the interaction for the download of the HD map update in the case that the vehicle enters the area where an update is available. In this case, this application component exploits the information provided by the HD map service to connect to the Edge HD map service and retrieve the HD map update.

8.1.2.1.5 Vehicle sensor data collection

This application belongs to both the It-UC1 and It-UC2 CAM applications. It is responsible for dialoguing with the sensors for collecting the raw data.

The collected raw data are packed in a ROSBag archive that is transferred to the edge server using File Transfer Protocol Secure (FTPS). The application notifies the edge service (i.e., Edge reporting service for It-UC1 and Edge HD map service for It-UC2) when it begins the transmission, and it notifies the edge service when the data transfer is completed. These notifications are sent to the edge service, which exposes RESTful endpoints for receiving such notifications.

The vehicle is equipped with a heterogeneous perception system composed of one front-facing stereo camera, three monocular cameras (two mounted laterally and one positioned at the rear), one central solid-state LiDAR and two rotating LiDARs installed on the sides. All data streams produced by the perception sensors are continuously acquired and stored as raw messages within ROSBag files. For the camera subsystem, both raw image frames and camera intrinsic parameters (including the calibration matrix K, distortion coefficients d, and auxiliary metadata) are included in the recordings to enable proper post-processing and 3D reconstruction downstream.

For LiDAR data, a dedicated pre-processing step is applied before transmission. This step removes redundant or irrelevant points belonging to the vehicle body itself and filters out points below ground level that may originate from sensor noise or reflection artefacts. A calibration process has been performed on the LiDAR sensors to align their reference frames and ensure consistent spatial correspondence across different points of view. This calibration enables multi-LiDAR fusion at the edge or cloud level. Alongside perception data, the vehicle also records and transmits sensor transformation frames (TF tree information) and GNSS pose data.

All recorded data streams are consolidated into ROSBag archives, where visual, LiDAR and GNSS information is stored in a time-aligned format. Camera frames are compressed to reduce bandwidth and storage usage while preserving a level of image quality suitable for perception, localization and mapping tasks. To support this acquisition workflow, the perception subsystem publishes sensor data through a structured set of ROS topics.

The following topic groups are defined:

1. Camera image topics

/<camera_id>/image_compressed

- Publishes the compressed image stream generated by each camera.
- Contains encoded image frames reduced in size to optimize bandwidth and storage consumption;
- Topics are instantiated per camera (e.g. /stereo/left/image_compressed, mono_left/image_compressed, /mono_rear/image_compressed).

2. Camera information topics

/<camera_id>/camera_info

- Distributes calibration and intrinsic camera parameters required for geometric reconstruction.
- Includes:
  - Camera matrix (K);
  - Distortion coefficients (D);
  - Rectification and projection matrices;
  - Sensor resolution and frame rate metadata.

3. LiDAR raw point cloud topics

/<lidar_id>/points_raw
Publishes raw 3D point cloud messages in ROS sensor_msgs/PointCloud2 format.

- One topic per LiDAR (e.g. /lidar_center/points_raw, /lidar_left/points_raw, /lidar_right/points_raw).
- Contains XYZ coordinates, reflectivity values and timestamp for each laser return.
- A preprocessing stage may be applied before logging:
  - Removal of points belonging to the vehicle body;
  - Filtering of ground-below noise and inconsistent reflections;
  - Alignment through calibration parameters for multi-LiDAR fusion.

4. GNSS positioning topic

/gnss

- Provides absolute geolocation data from the onboard GNSS unit.
- Typically encoded as sensor_msgs/NavSatFix;
- Contains latitude, longitude, altitude, accuracy indicators and timestamp.

8.1.2.2 Edge-side components

8.1.2.2.1 3D Scene Reconstruction service

The 3D Scene Reconstruction service is the edge application component that is in charge of creating the 3D scene reconstruction, exploiting the raw sensor data received from the vehicles. This component is part of the It-UC1 CAM application, and it is dynamically instantiated by the Reporting service using the CAMARA Edge Cloud API.

The core of the 3D Scene Reconstruction service is the multi-stage pipeline, named "*Scene Mapping and 3D Reconstruction Pipeline*", that, starting from the received raw sensor data, creates the 3D scene to be visualized. The description of the operational steps of the pipeline is provided in Section 8.1.2.2.3.

The pipeline outputs a 3D reconstruction of the accident scene, which can be viewed by first responders and other stakeholders, using a dedicated tool, to improve situational awareness before the intervention. An example of the output is provided in Figure 49.

Figure 49 – 3D Scene output example.

8.1.2.2.2 Edge HD map service

The Edge HD map service belongs to the It-UC2 CAM application. It is an edge application component that is dynamically instantiated by the HD map service using the CAMARA Edge Cloud API. This service oversees the creation of the HD map point cloud, which updates the current map available on the vehicle side.

The Edge HD map service exploits part of the Scene Mapping and 3D Reconstruction pipeline, which is also used by the 3D Scene Reconstruction service. In particular, the creation of the fused point cloud map constituted the output of this service. An example for such a point cloud is shown in Figure 50. The details about this process are reported in Section 8.1.2.2.3.

The Edge HD map service, besides the management of the HD map update creation, is also responsible for providing the HD map updates to other vehicles. This task is being supervised by the infrastructure service that provides the information required for the transfer using FTSP to the Edge and to the Vehicle map service. An example of the output is provided in Figure 49.

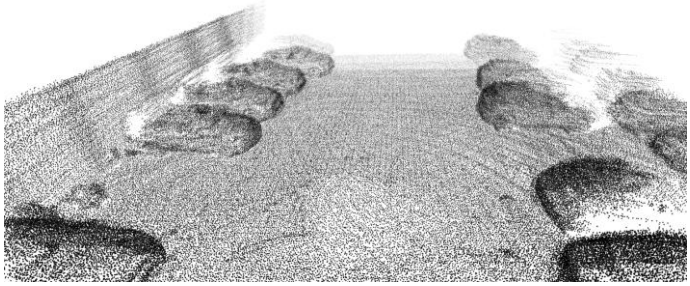

Figure 50 – HD map point cloud.

8.1.2.2.3 Scene Mapping and 3D Reconstruction Pipeline

The Scene Mapping and 3D Reconstruction Pipeline is organised into multiple subcomponents that constitute a multi-stage pipeline that ingests the raw sensor data gathered from vehicles, and it produces as output the 3D reconstruction of the accident scene for It-UC1 and the HD map point cloud for It-UC2. Figure 51 illustrates the details of the multi-stage pipeline, highlighting the subcomponents.

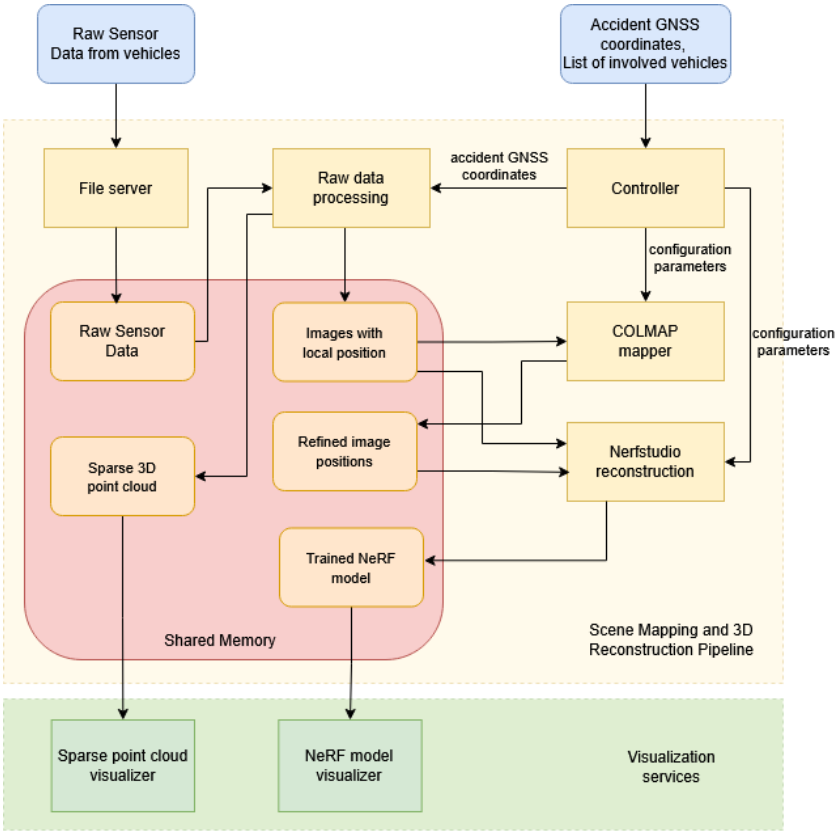

Figure 51 – Architecture of the multi-stage Scene Mapping and 3D Reconstruction Pipeline.

The sensor data reception from vehicles is managed by the *'File server'* subcomponent of the multi-stage pipeline. The file server receives data transmitted by the vehicles using FTPS and stores them in a shared memory area to be accessed by the other components.

Other input information is provided to the *'Controller'* subcomponent, which receives the information about the accident and the vehicles involved in the sensor provisioning process from the Reporting service through RESTful endpoints. It is also responsible for updating the Reporting service about the progress in the 3D Scene Reconstruction pipeline.

The Controller acts as an orchestrator by initialising and monitoring the execution of the other software subcomponents involved in the 3D Scene Reconstruction service. All the processing subcomponents in the pipeline are self-contained in a Docker container, and they are instantiated by the Controller sequentially for each step of the 3D Scene Reconstruction process and then deallocated when their tasks are accomplished.

The first step of the 3D Reconstruction service's pipeline is the raw sensor data processing. The *'Raw data processing'* subcomponent is responsible for the preparation of the raw sensor data received for the other components and for creating an initial coarse reconstruction of the accident scene that can be visualised as a preview. This subcomponent utilises ROS2 libraries to load and interpret the data topics stored in the provided ROSBag archive. The raw data collected from the vehicle are provided as a ROSBag file, which serves as a storage of timestamped messages organized by topic since each sensor stream is recorded in its own topic. These messages are

then processed in chronological order to reconstruct the vehicle's trajectory within the accident scene. The following processing steps take place:

- LiDAR messages are compared with each new message with Iterative Closest Point (ICP) algorithms to compute the transform between each successive position, which gives a precise estimate of the movement of the vehicle inside the scene. These algorithms are used to align two consecutive LiDAR point clouds by estimating the rigid transformation (rotation and translation) that best matches them. This transformation is also used to align all the point clouds coming from the LiDARs to create a coarse representation of the scene. The point clouds are then fused into a single point cloud that can be exported. This output constitutes the HD map update that is needed by the Edge HD map service.
- The GNSS data is used, together with the local position data from the lidar processing, to get the global position of the vehicle inside the accident scene.
- Camera images are extracted from the messages and saved in the shared memory; each image is saved with the information of the camera position relative to the scene at the time of acquisition.

After the raw data processing phase is completed, the Controller instantiates the '*COmputer vision Library for Multi-view Photogrammetry (COLMAP) mapper*' component[16]. COLMAP is an open-source Structure-from-Motion (SfM) and Multi-View Stereo (MVS) framework that is used to estimate the camera poses and to reconstruct coarse or dense 3D scenes from images. In this step, this component uses the COLMAP structure from motion algorithms to refine the image positions obtained in the previous step. The images are first processed by extracting a set of key points using Scale-Invariant Feature Transform (SIFT), which is a classical computer vision algorithm that identifies distinctive and repeatable features in an image. These features are then matched between each frame to find possible correspondence. The last step is to run the COLMAP pose prior mapper that takes as input the image positions as prior information, together with the extracted features, to solve the bundle adjustment problem. The output of the COLMAP module is a set of refined image positions (the positions given by the raw data processing step may be affected by errors from the sensors). These positions can then be used to create a more accurate reconstruction of the accident scene.

In the final step, the Controller instantiates the '*Nerfstudio reconstruction*' subcomponent[17]. Neural Radiance Fields (NeRF) is a deep-learning technique that represents a 3D scene as a neural network that can synthesise novel views from input images. The Nerfstudio reconstruction module takes the camera images with the refined poses computed by the previous module to train a NeRF model of the accident scene. The Nerfstudio library provides a simplified pipeline for creating training and visualising Nerfs. For this use case, the model selected was Nerfacto, the default library model that gave the best reconstruction results using real data.

The last two subcomponents of the 3D reconstruction pipeline are the web visualizers that enable third parties to access and visualise the reconstruction data. The sparse accident point cloud visualizer is a simple web server utilising the THREE.js 3D rendering library to visualise the accident point cloud reconstructed by the raw data processing component. The NeRF model visualizer is provided via the Nerfstudio library via a web viewer.

[16] https://colmap.github.io/
[17] https://docs.nerf.studio/

8.1.2.3 Infrastructure-side components

8.1.2.3.1 Reporting service

The Reporting service is an application component belonging to the It-UC1 CAM application. It is responsible for monitoring whether any accident occurs and, if so, performing the required steps to make the In-Reporting service work correctly. These steps include:

- The reception of DENMs, through the ITS communication software stack, from vehicles and the DENMs payload processing for extracting accident information; the DENM can indeed provide the position of the accident (i.e., *eventPosition* field) and the time at which it has been identified (i.e., *detectionTime* field); furthermore, the DENM provides some indications about the area of relevance of the accident (i.e., *relevanceDistance* field). This information could be useful for defining the area in which to check for vehicles being present.
- Querying the CAMARA Devices in Area API to obtain information about the vehicles present in the selected geographical area.
- Querying the CAMARA QoD API for requesting on-demand network resources for the involved vehicles that need to share raw sensor data.
- Querying the CAMARA Edge Cloud API for instantiating the Vehicle reporting service at the involved vehicles and the 3D Scene Reconstruction service at the edge server
- The forwarding of the accident information, of the details about how to share the sensor data, and of the network-related information to the Vehicle reporting services running on the involved vehicles.

8.1.2.3.2 HD map service

The HD map service is an application component that is part of the It-UC2 CAM application. It oversees listening for map change notifications and informing vehicles about the availability of HD map updates.

This application component performs the following steps when it receives a map change notification:

- It listens for map change notifications, and, upon reception, it processes the notification for identifying the location of the change and its extension.
- It performs a query to the CAMARA QoD API for requesting on-demand network resources for the vehicle that signaled the map change, and it needs to send the raw sensor data.
- It contacts the CAMARA Edge Cloud API for instantiating the Vehicle map service at the vehicle side and the Edge HD map service at the edge server
- It sends the sensor data sharing details and the network-related information to the Vehicle map service.

The HD map service takes the following steps for managing the provision of HD map updates to all the vehicles that are entering an area in which an update is available:

- It contacts the CAMARA Devices in Area API to create a subscription for receiving notifications about vehicles that are entering the area of interest for the HD map update.

- Upon reception of a notification from the CAMARA Devices in Area API, it performs a query to the CAMARA QoD API for requesting on-demand network resources for the vehicle that enters into the relevant area and that needs to receive the HD map update.
- It contacts the CAMARA Edge Cloud API for instantiating the Vehicle map service at the vehicle side and the Edge HD map service at the edge server
- It sends to the vehicle the information for retrieving the HD map update and the network-related information.

8.1.3 Integration of APIs

The CAM Applications of It-UC1 and It-UC2 consume the following APIs, which are available in the Italian trial site:

Table 49 – APIs consumed by CAM applications at the Italian trial site.

Enabler	API	Native API
Quality on Demand (QoD)	CAMARA Quality-on-Demand	3GPP AsSessionWithQoS
Device Location	CAMARA Devices in Area	ETSI MEC-013
Edge Cloud (Edge Application Management)	CAMARA Edge Cloud (Edge Application Management)	K8s APIs, Helm APIs

The Edge Application Management API, which is introduced in Section 5.2, is also exploited by the CAM applications for the dynamic instantiation of some components based on the received trigger.

The API consumption is performed specifically by the *Reporting service* component in It-UC1 and by the *HD map service* component in It-UC2.

These two components, after receiving a trigger, interact with the CAMARA QoD APIs to request the needed network resources for the involved vehicles and, thanks to the Edge Application Management API, instantiate the additional services required for the correct execution of the CAM Application.

The CAMARA Devices in Area API is exploited for selecting which vehicles are relevant for the execution of the CAM Application to be considered in the demonstration.

8.2 Dutch Site

The Dutch trial site serves as a dynamic testbed, showcasing three innovative use cases centred on the concept of a digital twin, as detailed in D2.1. This section provides a comprehensive overview of the CAM Application architecture, featuring in-depth descriptions of each component and highlighting the functionality and integration of the relevant APIs.

8.2.1 Architecture

This section presents the Connected Automated Mobility (CAM) application architecture implemented at the trial site. The generic architecture is organised into two tightly connected domains: the Digital Twin and teleoperation domain, and the V2X communication and coordination

domain (see Figure 52). Together, they enable cooperative perception, automated behaviour support, and seamless integration between physical vehicles and cloud-based services.

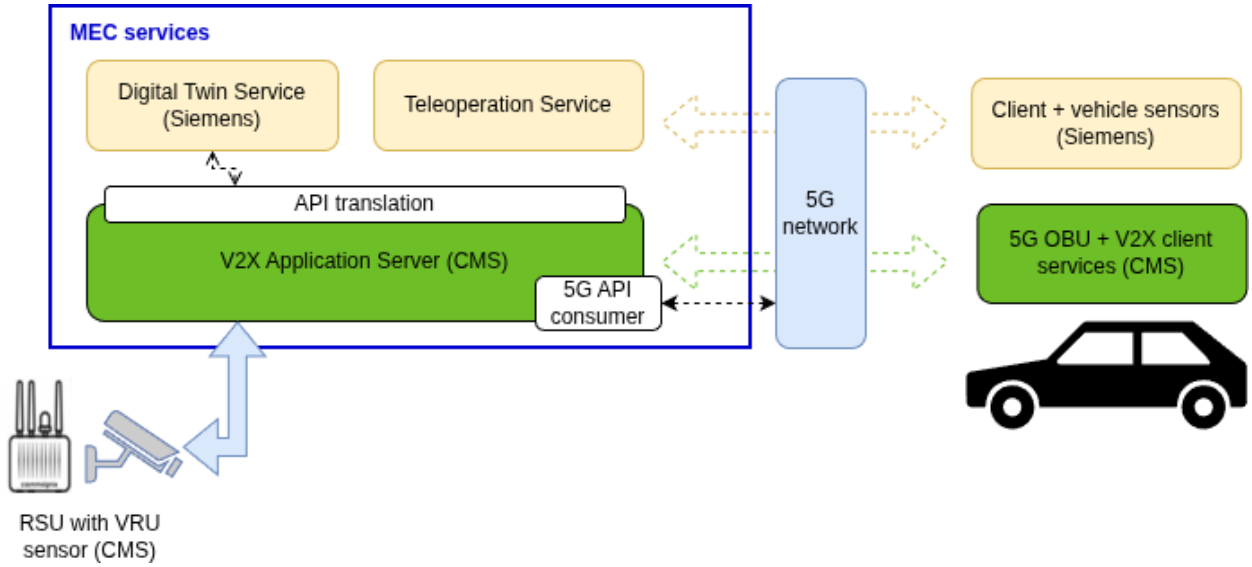

Figure 52 – General CAM application architecture for the Dutch trial site.

The first domain centers on the Digital Twin (DT) and teleoperation services. These components consume real-time vehicle sensor data to maintain an accurate virtual representation of the vehicle and its surroundings. The in-vehicle client interfaces with the vehicle's sensor suite and CAN bus, not only to provide upstream data to the DT but also to inject virtual objects into the vehicle's ADAS stack for testing, validation, and scenario augmentation. This domain also provides the foundation for teleoperation, enabling remote operators to monitor and intervene when necessary.

The second domain is the V2X communication and service ecosystem, where the V2X Application Server plays the central role. While the 5G OBU hosts the V2X protocol stack and ensures local connectivity, it is the V2X Application Server that orchestrates the system-level behavior. It manages the interaction between all connected road users—vehicles, VRUs, and camera-equipped RSUs—primarily via V2N communication. It aggregates perception and state information, performs cross-entity coordination, and exposes selected APIs both towards the 5G system and upwards to the Digital Twin through a translation layer. This makes the V2X Application Server the key integration point that links real-time connectivity and cooperative awareness.

Together, these two domains form the CAM applications of the trial site, enabling synchronized vehicle–cloud interaction, richer cooperative perception, and advanced CAM functionalities such as teleoperation, cross-vehicle awareness sharing, and cloud-assisted mixed-reality testing.

8.2.1.1 Use case-specific architecture

Dt-UC3 "Periodic data collection for improving Digital Twin" focuses on the collection of vehicle sensor data assisted by on-demand Quality-of-Service and other APIs. The realization of the generic CAM architecture for Dt-UC3 is shown in Figure 53. This scenario is the least demanding in terms of network requirements; no low latency or real time requirements, but the throughput can be relatively high due to the transmission of raw sensor data.

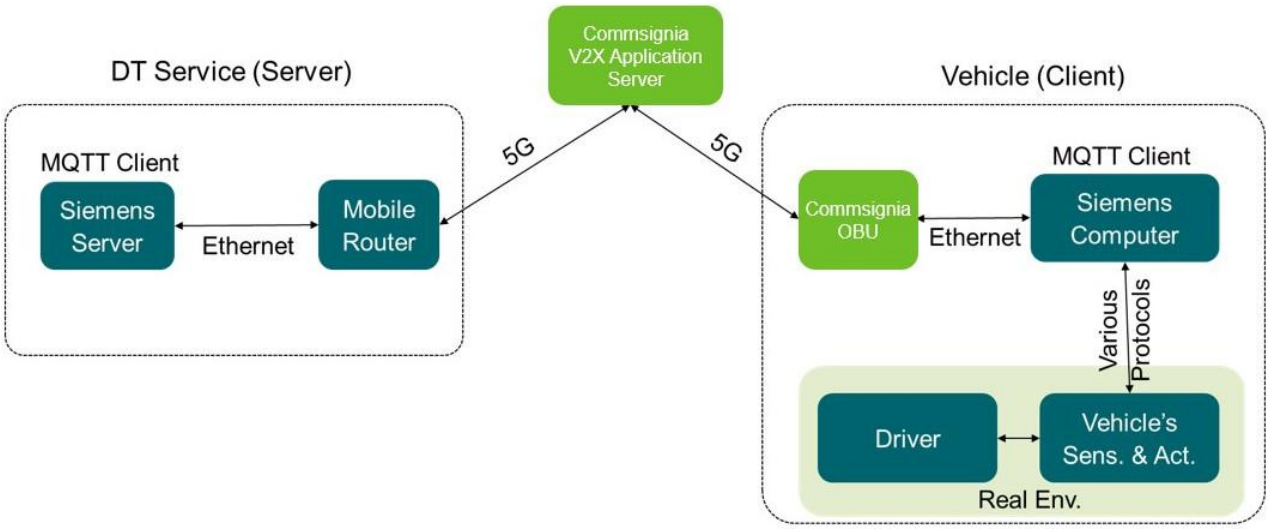

Figure 53 – CAM application architecture of Dt-UC3.

The vehicle client and the Digital Twin server services operated by Siemens are connected to CMS MQTT broker (component of the V2X Application Server) to exchange control information and sensor data over MQTT protocol. The Digital Twin requests the vehicle client to start the data transmission, which in turn collects various sensor outputs (camera, GPS) and start the data feed over MQTT. The 5G-capable Commsignia OBU is the in-vehicle modem providing cellular connectivity for the Siemens client services and the V2X client services running on the OBU.

The V2X Application Server connects the vehicle with the Digital Twin services, while exposing the QoD features to the DT via a simple API translation layer. The DT can request from a set of pre-defined QoS levels, and the V2X Application Server translates the request to the APIs exposed by the 5G network.

As a next step in a set of use cases with increasing complexity and requirements, Dt-UC4 “Vehicle testing with mixed reality” puts more emphasis on the real-timeliness of data transmission, introducing a use case where low latency is crucial. Connectivity is handled the same way as in Dt-UC3, the main difference is in the direction and properties of the network traffic, and the introduction of vehicle actuator components within the vehicle client. Figure 54 summarizes the CAM application architecture for Dt-UC4.

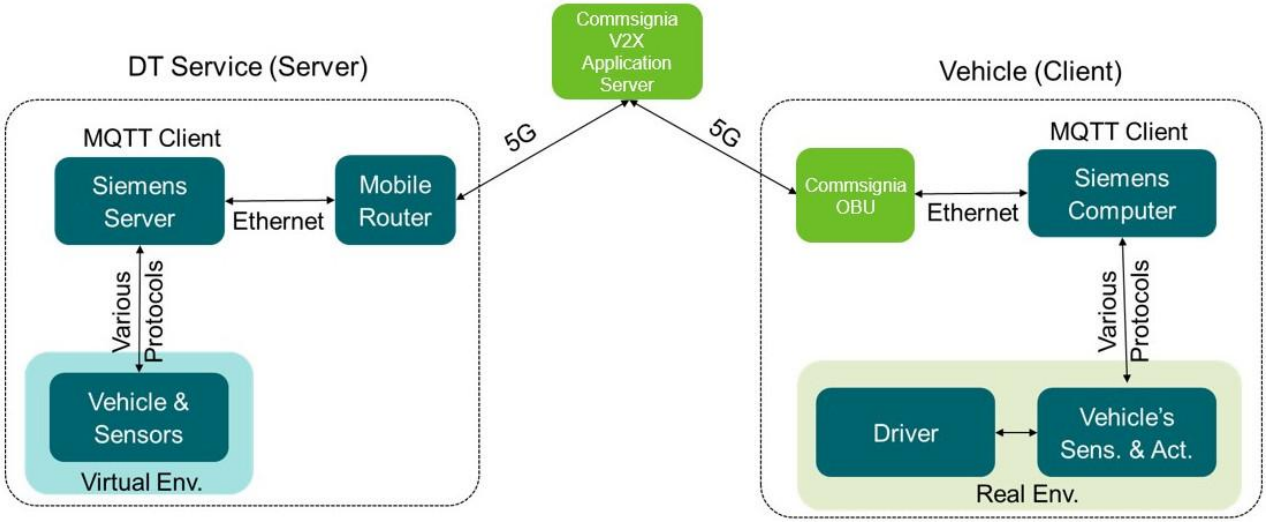

Figure 54 – CAM application architecture of Dt-UC4.

In this use case, virtualized objects are injected by the Digital Twin into the environment model of the vehicle over the network to test how the vehicle's automated driving system reacts to dangerous situations. The virtualized objects can be synthetically created, or they can be pre-recorded detections obtained from the V2X Application Server (e.g., pedestrians detected by an RSU equipped with a smart camera). The Digital Twin transmits the object feed to the vehicle over MQTT. The in-vehicle client computer can then transform and feed the received objects into the vehicle CAN bus, triggering the ADAS systems and actuators.

The third, most demanding use case is Dt-UC5 "Tele-operated driving aided by Digital twin" is a combination of upstream and downstream traffic with the strictest network requirements. As seen in Figure 55, a virtual copy of the vehicle's real environment must be maintained in real time with the Digital Twin to aid the teleoperation process. Sensor data from the vehicle and driving commands from the tele-driver must be transmitted with the lowest possible latency and high reliability. Additionally, the V2X Application Server can provide information about additional road users like connected VRUs using smartphones or cars/pedestrians detected by roadside infrastructure for maximum situational awareness for the tele-driver.

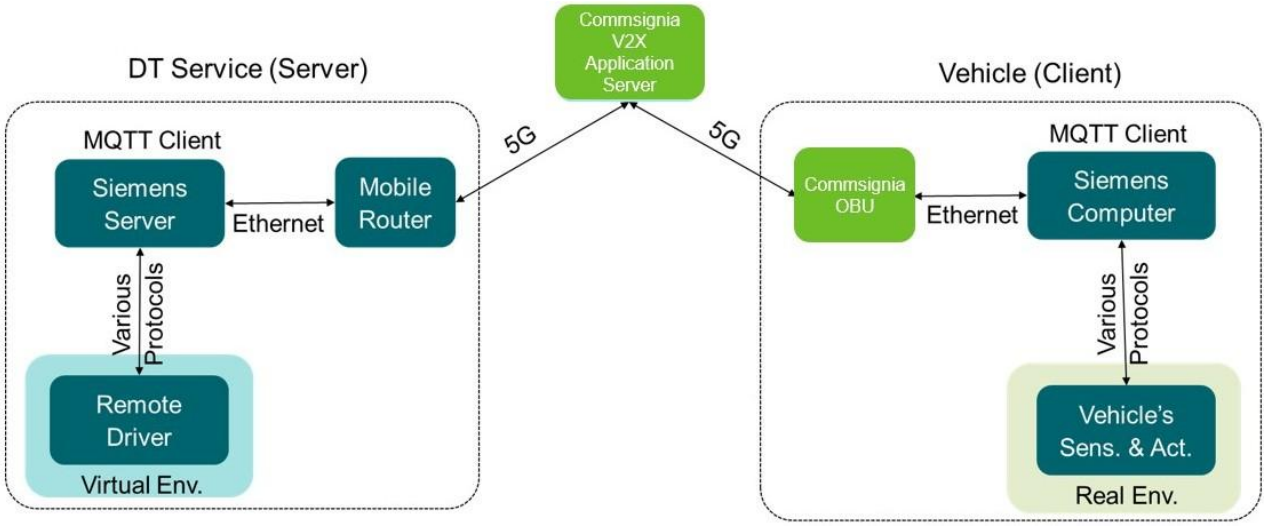

Figure 55 – CAM application architecture of Dt-UC5.

8.2.2 CAM Application Components

This section describes the CAM application components in further detail. The CAM application components are divided into two categories: i) V2X service components, and ii) Digital Twin and teleoperation components.

8.2.2.1 V2X service components

8.2.2.1.1 Commsignia On-board unit (OBU)

The OBU hardware running a proprietary ITS protocol stack (both listed as background assets) provides 5G connectivity for the vehicle (including the other devices and services).

8.2.2.1.2 Commsignia V2X Application Server

The Commsignia V2X Application Server, provided as a background asset for the ENVELOPE Dutch trial site, serves as the primary application-level connectivity backbone for the implemented use cases. Its role is central and twofold.

First, as a core system component, the V2X Application Server interconnects all V2N-enabled road users—including OBUs, RSUs, and VRUs equipped with smartphones—within a unified communication environment. Acting as an intelligent coordination layer, it continuously monitors connected clients, manages their registration and presence, and performs optimal C-ITS message routing between relevant traffic participants. By aggregating and correlating information from all connected road users, the server can enrich the Digital Twin with data on entities not directly observed by the vehicle client. This extended awareness significantly enhances safety, particularly during teleoperation scenarios where blind spots or occluded road users pose elevated risks.

Second, the V2X Application Server has been augmented with dedicated modules that consume 5G APIs and selectively expose their outputs to the Digital Twin through a translation and adaptation layer. These extensions allow the server to act as a mediator between the communication network and the cloud-based vehicle representation, enabling the Digital Twin to access network-derived insights in a harmonized, application-friendly format. Moreover, by orchestrating the setup and maintenance of on-demand QoS sessions for the vehicle, the V2X Application Server actively supports the Digital Twin service in ensuring that latency-sensitive functions—such as teleoperation or enhanced perception—receive the appropriate network performance levels when needed.

The V2X Application consumes the APIs available at the Dutch trial site according to the API descriptions listed in D3.2 and in this deliverable. The simplified exposure interface toward the Digital Twin service is an MQTT-based interface for enabling the Digital Twin the monitoring the available QoS and making requests. This interface is translated into API interactions related to the QoD enabler.

8.2.2.2 Digital Twin and teleoperation components

In this section, we introduce a five-dimensional digital twin as presented in Figure 56.

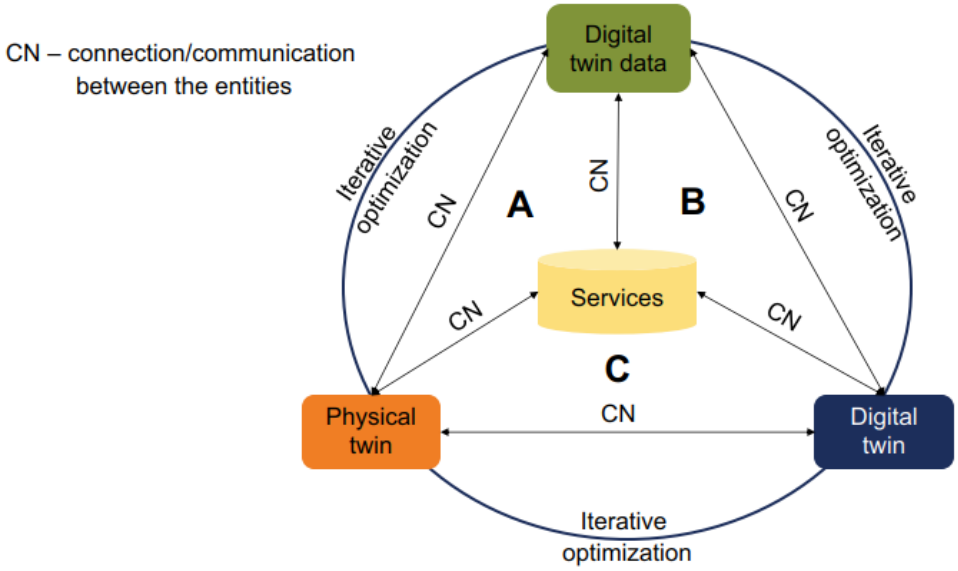

Figure 56 – Five-dimensional digital twin.

Services: This represents the digital twin system's services.

Physical twin: This is the real product, the physical entity that users can operate.

Digital twin: This is the mirror image of the physical product (the virtual entity).

Digital twin data (DTD): This is data collected from physical and virtual spaces, which can be analyzed (data analytics), integrated and visualized.

Connections: These are interconnections among the previous four components, which various technologies, including network communication, IoT and network security.

In Figure 56, with the DT services in the middle, we can identify three triangles:

- Triangle A is defined by the following vertices: the physical twin, services and digital twin data, which are mainly exercised to remotely monitor the operation of the physical twin and to collect digital twin data (UC3).
- Triangle B is defined by the following vertices: digital twin data, services and the digital twin, which are mainly exercised during development, building and validating the digital twin model using the DTD and for virtual verification and validation (UC4).
- Triangle C is defined by the following vertices: the digital twin, services and the physical twin, which are mainly exercised during physical testing and after deployment. For example, over-the-air (OTA) updates, safe teleoperation (UC5).

The safe teleoperation component diagram is presented in Figure 57, where the “VehicleClient” is composed by:

- “VehicleCameraClient” sending timestamped camera frames from the teleoperated vehicle
- “VehicleGPSClient” sending timestamped GPS coordinates of the teleoperated vehicle
- “VehicleControlClient” controlling the vehicle based on the received teleoperation commands.

The “DigitalTwinClient” and the “DigitalTwinControlClient” run on the teleoperator (remote driver) side, where the first component is remotely monitoring the vehicle and the communication delay and the second component is broadcasting the teleoperation commands.

At the start of the safe teleoperation the “DigitalTwinControlClient” via the MQTT broker requests a QoD/QoS profile from the “CMSClient”. If the requested QoD/QoS is received the safe teleoperation can start, where the “VehicleClient” components and “DigitaTwinClient” components communicate via the MQTT broker.

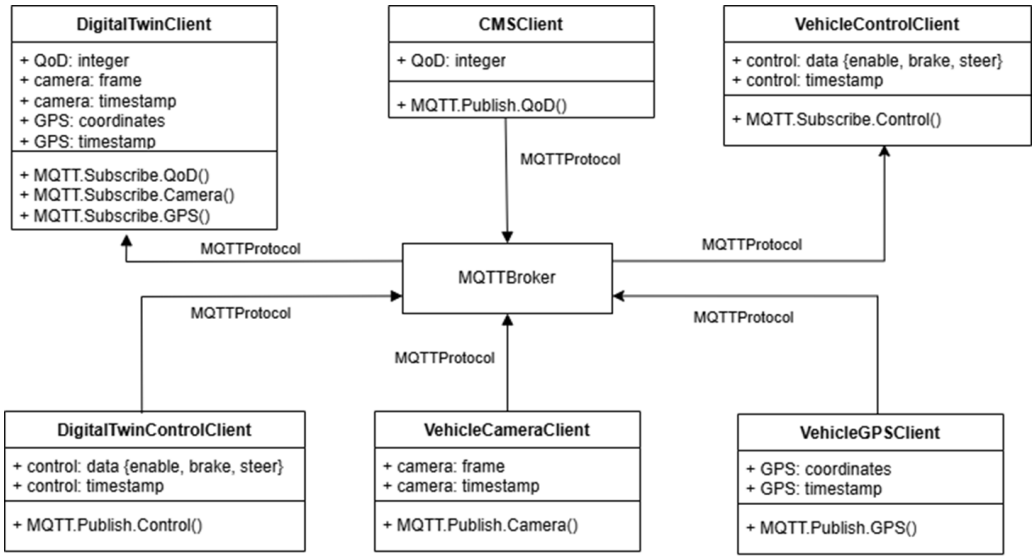

Figure 57 – Safe teleoperation – component diagram.

8.2.3 Integration of APIs

The CAM applications consume the following APIs available at the Dutch trial site using the API consumer module of the V2X Application Server.

Table 50 – APIs consumed by CAM applications at the Dutch trial site.

Enabler	API	Native API
Quality on Demand (QoD)	CAMARA Quality-on-Demand	3GPP AsSessionWithQoS
Device Location (Geofencing)	CAMARA Device Location (Geofencing Subscriptions)	3GPP MonitoringEvent
Performance Metrics	ENVELOPE Performance Metrics API	3GPP AnalyticsExposure (ueCommInfos)
Predictive QoS	CAMARA Predictive Connectivity Data	3GPP AnalyticsExposure (qosSustainInfos)
Edge Cloud (Edge Application Management)	CAMARA Edge Cloud (Edge Application Management)	K8s APIs, Helm APIs
Edge Cloud (Traffic Influence)	CAMARA Edge Cloud (Traffic Influence)	3GPP ServiceParameter

The *CAMARA QoD API* is mainly triggered after a request from the Digital Twin is received using the translation layer exposed by the V2X Application Server. Additionally, the V2X Application Server can use the QoD API in combination with other APIs, e.g., by requesting a different QoS profile if there is low throughput indicated by the *Performance Metrics API* notifications, or it can request for a QoD session if a monitored road user enters a designated area, triggering a notification from the *Device Location API*.

Predictive QoS and *Edge Cloud* are used in Dt-UC5, enabling timelier QoS adaptation of the teleoperated vehicle's data traffic, and dynamic edge discovery with service instantiation of the V2X services and traffic redirection to the target edge's local UPF.

8.3 Greek Site

The Greek Site CAM Application builds around the concept of 'Data Sharing for Situation Awareness' as described in D2.1. In short, the use case envisages the following scenario that is realized by applications in the vehicles, servers at the Local Traffic Centre and servers at the Vehicle's Supervision Centre as depicted in Figure 58:

1. The vehicles maintain subscription with their Traffic management server (Local Traffic Center) for data sharing, sending/receiving relevant notifications over the edge.
2. Vehicles in proximity share information (breaking alerts). The cars are equipped with sensors and applications to monitor and interact with the network.
3. The vehicle in front ("1st vehicle") is the MILLA autonomous car that also maintains subscription to its supervision center sending/receiving sensors' data over the cloud.
4. The autonomous vehicle (1st vehicle) is requested to stop immediately (either on its own or by the supervision center). A breaking alert message is generated and sent instantly over the 5G network to the traffic management system.
5. The vehicle that follows ("2nd vehicle"), receives the "alert" from the traffic management system and acts accordingly. The alert message is presented on the mobile application (text, red flashing indication) along with an audio alert.
6. The 2nd vehicle can "connect" to the camera of the 1st vehicle for a live view of the environment.

7. The mobile application displays in real-time Location Monitoring (e.g. cellId) & predictive QoS (notifications for service degradation) for informative purposes.
8. On certain thresholds and approval from the vehicle passengers, the application utilizes QoD to ensure that the performance standards are maintained.

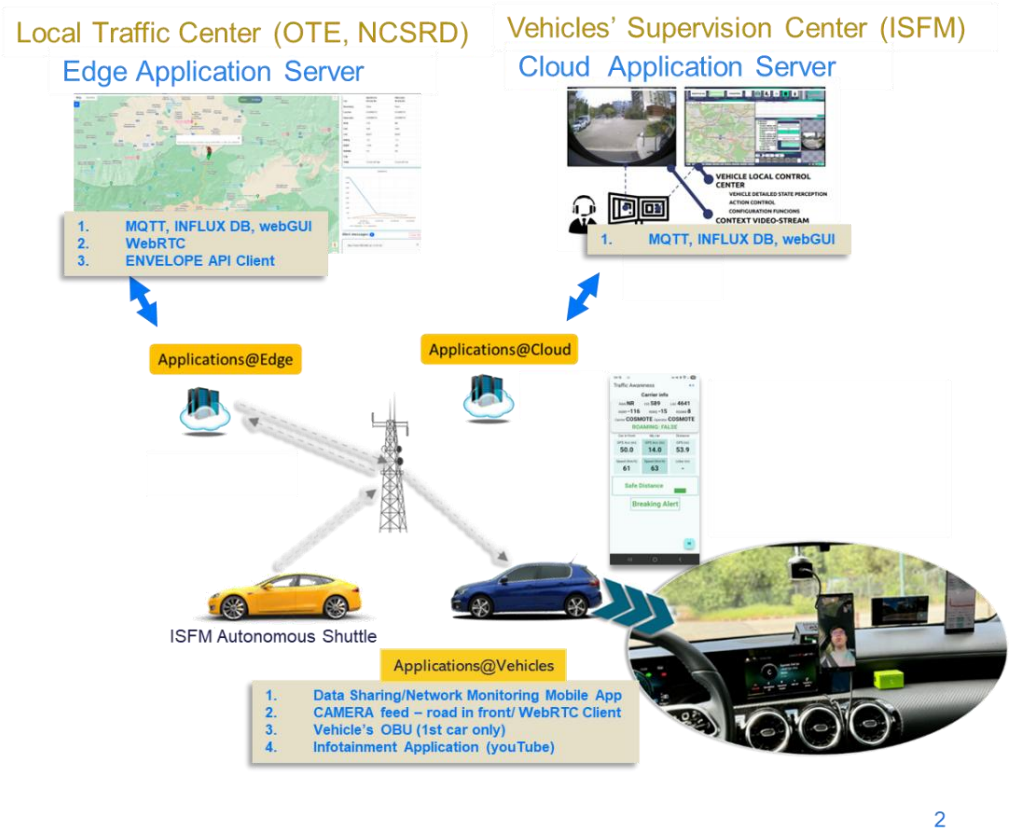

Figure 58 – Data Sharing for Situation Awareness Execution Scenario.

The execution simulates a complex scenario at the cross-borders and assesses the **inter-PLMN Hand Over** (see Section 7) **and Edge Continuum** (see Section 5.2.3.2) capabilities developed by the project. Practically, it considers that as the vehicles move from the home country/HPLMN ("OTE") to a visited country/VPLMN ("NCSRD") the CAM service is not disrupted, and the session is migrated from the starting traffic center (OTE edge) to the visited traffic center (NCSRD edge). For the CAM application developments, the trigger condition to migrate from one traffic center to another is the indication of network change (PLMN Change and/or UE IP Change), as this is a typical situation that occurs during border crossings.

As such the application spans in three different segments:

- **Client applications** running on the **vehicles**. These are:
  - The ISFM OBU installed in the L3+ MILLA autonomous van, collecting and dispatching vehicle's sensors information
  - A mobile application (Android) running on a passenger's User Equipment (UE) visualizing the application information
- **Local Traffic Centre Application Servers** running at the **edge**: These server applications are hosted on the edge server designated by each road operator and/or serving MNO. In

the ENVELOPE deployment two different edge sites are foreseen, hosted by OTE (representing the home edge) and NCSRD (representing the visited edge).

- **A Supervision Centre Application Server** running on the **cloud**: These relate to the maintenance & control services provided by the vehicle manufacturer and typically reside centrally in its supervision centres. In this case these services run at the MILLA premises in Paris, France.

8.3.1 Architecture

The system architecture of the application developed for the Greek site can be seen in Figure 59, depicting the positioning of the various components either in the cloud or the local edge. Each component is described in detail in the paragraphs that follow.

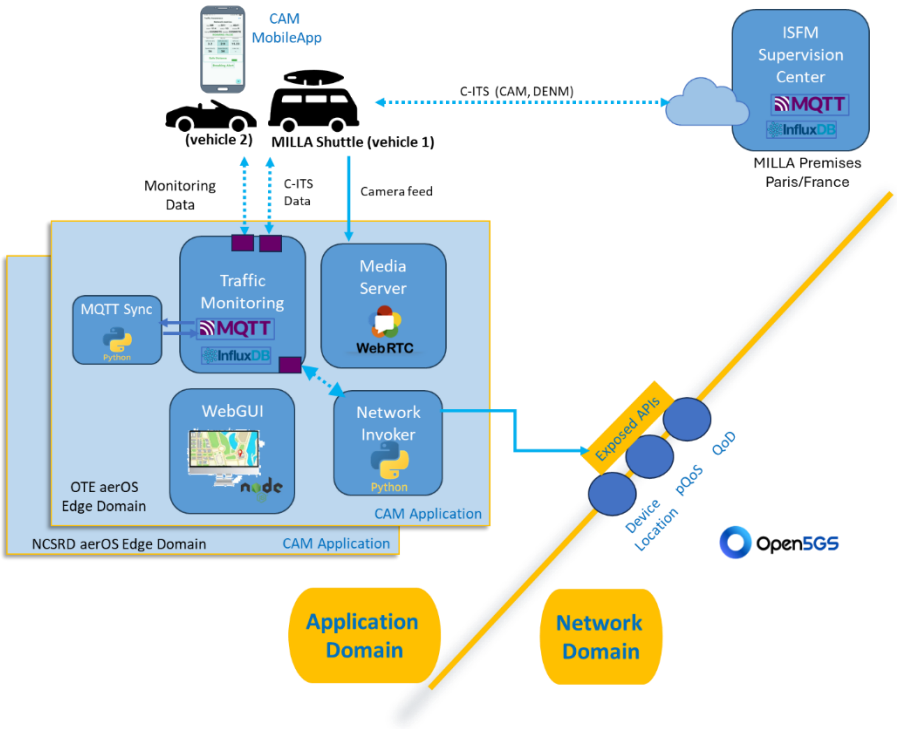

Figure 59 – CAM Application System Architecture.

To address service continuity considering mobility scenarios that mandate migration between the edge domains, the CAM Application is installed as an Eclipse aeriOS[18] Infrastructure Element that is instantiated in both edge domains, on the OTE and NCSRD sites. As an aeriOS installation, the CAM application is built over Kubernetes and through the aeriOS high-level (HLO) and low-level orchestration (LLO) leverages the dynamic edge deployment capabilities, and easy extensibility to new edge domains.

8.3.2 CAM Application Components

The CAM application components are classified as back-end, denoting the server-side developments located at the edge or cloud, and front-end that are denoting end-user interfaces and interactions, such as the vehicle's OBU and the passenger's mobile application.

[18] https://projects.eclipse.org/projects/iot.aerios, not to be confused with the aerOS edge-cloud platform.

8.3.2.1 Backend Components

8.3.2.1.1 Traffic Monitoring MQTT Server/INFLUX DB

At the heart of the ENVELOPE CAM architecture lies the MQTT message broker, implemented using Eclipse Mosquito, which serves as the central communication backbone for all data exchanges between system components. This component has been designed as part of ENVELOPE to enable real-time, low-latency, and scalable message distribution among the MILLA's On-Board Units (OBUs) by processing the generated C-ITS[19] messages. It has also been extended to process data provided through the ENVELOPE simplified network layer, such as predictive QoS, forming the foundation of an all-encompassing data orchestration layer.

The MQTT broker operates using dynamic topic generation, allowing the platform to automatically adapt to new vehicles, locations, and data types without requiring manual reconfiguration. Two main categories of topics are defined:

1. **C-ITS message topics**, which carry vehicular messages such as CAM (Cooperative Awareness Messages) and DENM (Decentralized Environmental Notification Messages) over dedicated MQTT topics. Each topic follows a structured format derived from the quadtree indexing logic, which spatially partitions the Earth's surface into progressively smaller tiles to enable location-specific data routing as graphically depicted below. The naming convention of a C-ITS topic is as follows:

   envelope/<message_type_id>/message_version/<quadtree_path>/<sender_id>, where:

   - *envelope*: Identifies that MQTT topics belong to the ENVELOPE ecosystem
   - <*message_type_id*>: Defines the message type in uppercase either CAM or DENM.
   - <*message_version*>: Specifies the ETSI EN/TS standard version used for encoding, such as 1_4_1 (CAM) and 1_3_1 (DENM)
   - <*quadtree_path*>: Represents the geospatial tile in which the vehicle is located. This path is constructed using a **quadtree data structure** (see Figure 60), composed of the digits 0, 1, 2, and 3, which recursively subdivide the world map into four quadrants:
     - 0: Northwest (top-left)
     - 1: Northeast (top-right)
     - 2: Southwest (bottom-left)
     - 3: Southeast (bottom-right).

     Each level of depth in the quadtree increases the spatial precision by dividing the area into four smaller tiles. The quadtree uses a **Mercator projection** to maintain consistent tile dimensions at all zoom levels, where the world at level 0 is a single tile and level *z* contains 2^z tiles in both horizontal and vertical directions.
   - <*sender_id*>: Unique identifier of the transmitting entity, typically corresponding to the vehicle or OBU.

[19] www.itsstandards.eu/app/uploads/sites/14/2020/10/C-ITS-Brochure-2020-FINAL.pdf

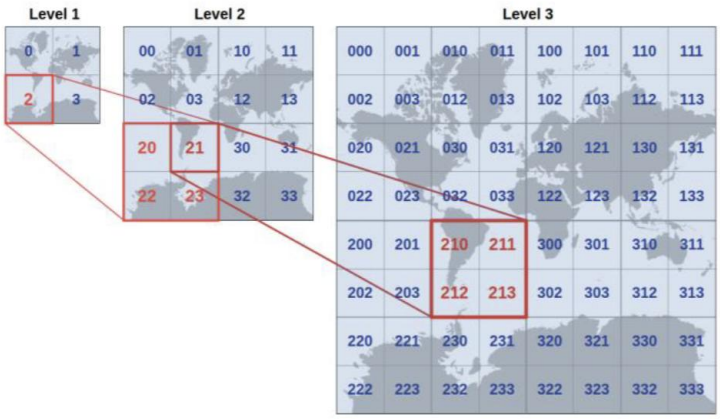

Figure 60 – The quadtree-based topic structure enables geographically bounded message distribution.

2. **Heartbeat and Informational topic**, a dedicated topic for system monitoring and data synchronization between APIs and the vehicles. This channel serves two purposes:
   - Heartbeat signalling, allowing the backend and connected clients to confirm that a vehicle is active and reachable in real time. Also, through heartbeat the UE publishes configuration parameters such as the acceptable QoS thresholds.
   - Informational exchange, where the backend publishes auxiliary data such as predictive QoS metrics or responses from the Location and PQoS APIs.

   The respective topic follows the format: *netstats/<vehicle_id>,* where:

   - *<vehicle_id>* is the MSISDN subscription number, as necessary to interact with the ENVELOPE exposure functions.

All messages exchanged through the CAM MQTT Server are in parallel stored to a time-series data base (MQTT) that allow historical data processing and can be used for the CAM Application KPI's validation.

The CAM MQTT servers/Influx DB are located to the nearest edge.

8.3.2.1.2 Network Invoker /APIClient Service

The Network Invoker forms the core of the ENVELOPE CAM backend, serving as the ENVELOPE API client infusing to the application predictive network intelligence, through network data retrieval and system communication. It integrates specialized libraries — Predictive Quality-of-Service (PQoS), Device Location, Quality on Demand (QoD), ATSSS — into a unified service framework that monitors active devices, retrieves network performance metrics, and distributes them across the ecosystem via MQTT.

This service not only consolidates data from diverse network sources but also ensures interoperability with both external systems (e.g., Open5GS, NEF and CAMARA APIs) and internal components (such as the CAM application and Traffic Control WebGUI). Through an asynchronous design and modular implementation, it delivers continuous real-time insights into the network conditions for each connected unit.

It is built as a python script that runs continuously as the back-end component ready to receive requests or apply scheduled actions. It coordinates MQTT communication, tracks active

subscribers (UEs), retrieves network-related data, and republishes aggregated information back to MQTT topics. Its asynchronous event-driven architecture, built with Python's asyncio, enables efficient parallel execution of data retrieval and message handling tasks.

Each connected user equipment (UE) or OBU periodically sends heartbeat messages to the MQTT broker. The Network Invoker listens to wildcard topic patterns (netstats/#) to dynamically detect active entities. Upon heartbeat reception, it registers or updates each subscriber using their unique MSISDN/ Mobile Number identifier and carrier information.

For every active subscriber, the service retrieves (when applicable and available):

- Location Data via the CAMARA Location Library,
- PQoS Data, via the PQoS CAMARA Library
- Quality on Demand (QoD) metrics from the QoD CAMARA Library.
- ATSSS Data via the multi-connectivity Library.

All collected data are combined into a unified JSON structure, timestamped, and republished to the relevant subscriber's MQTT topic for consumption by the CAM App and WebGUI.

The service also handles self-message prevention, message differentiation (heartbeat vs. stats), and automatic cleanup of inactive subscribers. Its resilience mechanisms ensure continuous operation, even when external APIs are unavailable, by generating valid fallback data structures.

The Python Service is fully containerized and optimized for Kubernetes deployment, with configuration handled via environment variables for flexibility and scalability. Core parameters, such as MQTT broker address, NEF API supported communication, CAMARA API communication, update intervals, and timeout durations, are configurable.

The Network Invoker is located within the Traffic Monitoring CAM MQTT at the nearest edge.

8.3.2.1.3 Traffic MQTT Sync Service

The Greek site developments consider the mobility requirements of the CAM services, that mandate the session migration from one Traffic Management edge to another as network or highway borders are crossed. In the use case this is inherently supported through the aeriOS edge-cloud infrastructure that allows the dynamic, adaptable service installation and federation. However, it is necessary for the applications to handle the session context migration, so that to ensure service continuity when the vehicle in front reach the new edge while the vehicles that follow are still in their home. In the context of the Greek site, two neighboring, however independent MQTT Brokers operate—one on each aeriOS Kubernetes cluster, at the OTE edge (Home Network) and NCSRD edge (Visited Network). To ensure unified data availability and prevent any loss of information across the two domains, a dedicated forwarding *broker* has been implemented. This *broker* listens to all relevant MQTT topics on *its* cluster (*e.g. OTE MQTT*) and securely republishes *on neighboring CAM MQTT clusters* (*e.g. NCSRD*). In effect, the mechanism synchronizes both environments, allowing the WebGUI to display all UE network metrics, CAM messages, safety events, and WebRTC presence data regardless of which cluster the device is attached to.

8.3.2.1.4 WebRTC Server

The WebRTC subsystem is designed following a three-layered architecture, enabling low-latency, real-time media exchange between the MILLA vehicle and the CAM ecosystem. At its core lies a central media server, deployed within the backend infrastructure, which supports multiplexed streaming — allowing multiple concurrent sessions and stream types, including WebRTC, RTSP,

and other compatible protocols. This server acts as an intermediary for session management, signaling, and secure forwarding of media between endpoints. This enables, for example, a rear vehicle to visualize what the leading MILLA car's camera sees in real time — a key feature for cooperative and assisted driving scenarios. In parallel, the Traffic Control WebGUI also has access to the same media streams through the backend media server, allowing operators to monitor multiple vehicles simultaneously through a browser interface.

The On-Board Unit (OBU) installed in the MILLA car serves as the stream initiator. It interfaces with the vehicle's onboard camera and establishes an outbound connection to the central media server, publishing its video feed using the WebRTC protocol. Each stream is uniquely identified using the vehicle's ID, which functions as a persistent identifier within the signaling framework. This design allows the media server to register active video streams and expose them to authenticated clients for discovery.

The WebRTC server is located with the Traffic Monitoring CAM MQTT at the nearest edge.

8.3.2.1.5 MILLA/ISFM Supervision Center

The ISFM supervision center is peripheral to the ENVELOPE developments is depicted in Figure 61, follows a similar design with the CAM MQTT Server described in 8.3.2.1.1 however is located at the ISFM headquarters. Within ENVELOPE its purpose is to ensure that project developments do not interfere with its service, which is pivotal for autonomous vehicles. Furthermore, being accessed through the public internet, it serves as a benchmark for comparing the performance with the edge-traffic-monitoring application server.

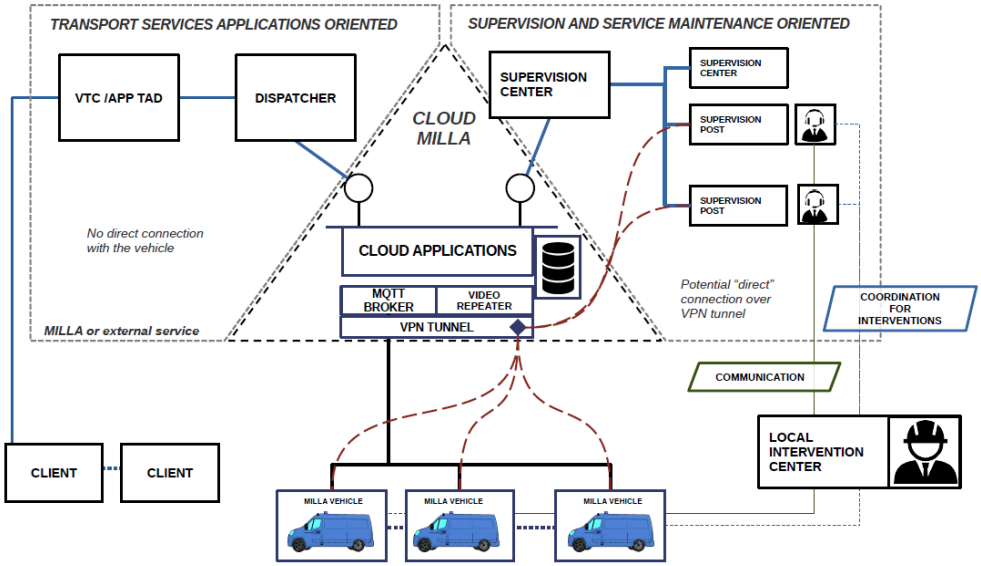

Figure 61 – MILLA ISFM Supervision Center.

The MILLA supervision server is located within the headquarters of MILLA in Paris, France at the cloud.

8.3.2.2 Frontend Components

8.3.2.2.1 Real-time Situation Awareness CAM Android Application

The CAM Android application serves as the primary mobile interface within the ENVELOPE CAM ecosystem, providing drivers and passengers with real-time visibility of both network performance

metrics and vehicular communication events. Running natively on Android devices, the application integrates multiple data sources and communication protocols to deliver an end-to-end situational awareness tool for connected mobility scenarios.

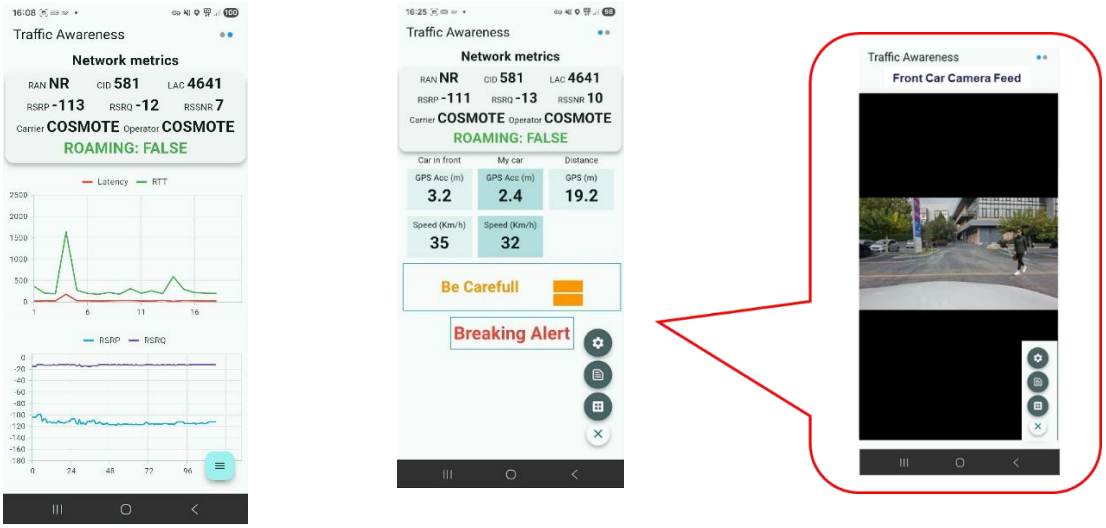

Figure 62 – CAM Application GUI Tabs.

The App, as shown in Figure 62, offers several tabs dedicated to:

- Real-time monitoring of the access network (such as signal strength, RAT type, serving gNB, serving carrier)
- Information regarding own car (such as location and safe-distance notifications) as well as information shared by the vehicles in-front, such breaking alerts.
- Visibility through the network integration, such as predictive QoS, location and interaction for QoD.

At its core, the CAM app maintains continuous communication with the MQTT broker, enabling it to both publish and subscribe to dynamic topics associated with the vehicle or user equipment (UE). Through this channel, the application receives **C-ITS messages** (such as Cooperative Awareness Messages — CAM or Decentralized Environmental Notification Messages — DENM) originating from On-Board Units (OBUs) installed in connected vehicles. These messages are processed and displayed to the user in real time, offering immediate awareness of nearby events, network conditions, or warnings such as braking alerts. In parallel, the application periodically transmits heartbeat messages to confirm the device's active presence within the network, ensuring backend services maintain an updated session state for each UE.

In addition to consuming C-ITS messages, the CAM app independently collects and reports **network performance data** from the mobile device itself. Using Android's telephony and connectivity APIs, it measures and logs radio parameters such as RSRP, RSRQ, and RSSNR, as well as latency metrics like Round Trip Time (RTT) and ping. These measurements are locally visualized within the application and simultaneously published to MQTT, allowing them to be aggregated and correlated by the backbend's metrics and visualize the total results on the traffic control webGui.

Another functionality of the CAM app is its **integration with the WebRTC subsystem**, which allows it to establish live video streams with the MILLA car's on-board cameras or other

connected vehicles. By referencing a unique vehicle identifier, the application can retrieve the live camera feed from the front car, effectively providing a “what-the-front-sees” visualization. This feature enhances situational awareness and supports cooperative driving scenarios where drivers or automated systems benefit from shared visibility ahead of their line of sight. The ATSSS multi-connectivity enabler (and corresponding API) will also be utilized to ensure that the video stream is delivered efficiently to terminal devices by seamlessly leveraging both 3GPP and non-3GPP interfaces, together with the MPQUIC framework (see also Section 4).

To ensure operational reliability, the CAM app incorporates **a Safe Mode mechanism** exploiting network intelligence. This feature leverages feedback from the *PQoS* service to monitor the quality and stability of the current data link. When degraded performance or high latency is detected, the application issues a visual alert to inform the user that network quality has dropped below acceptable thresholds and negotiates with the network through *QoD*. This safeguard helps drivers or operators make informed decisions regarding connectivity-dependent functions.

8.3.2.2.2 Traffic Control WebGUI

The Traffic Control WebGUI functions as the centralized monitoring and visualization dashboard for the ENVELOPE CAM ecosystem, located in the Traffic Control Room. It provides operators with a real-time overview of all active vehicles and user devices within the network, combining geolocation, network status, and communication data into an intuitive, map-based interface, as shown in Figure 63.

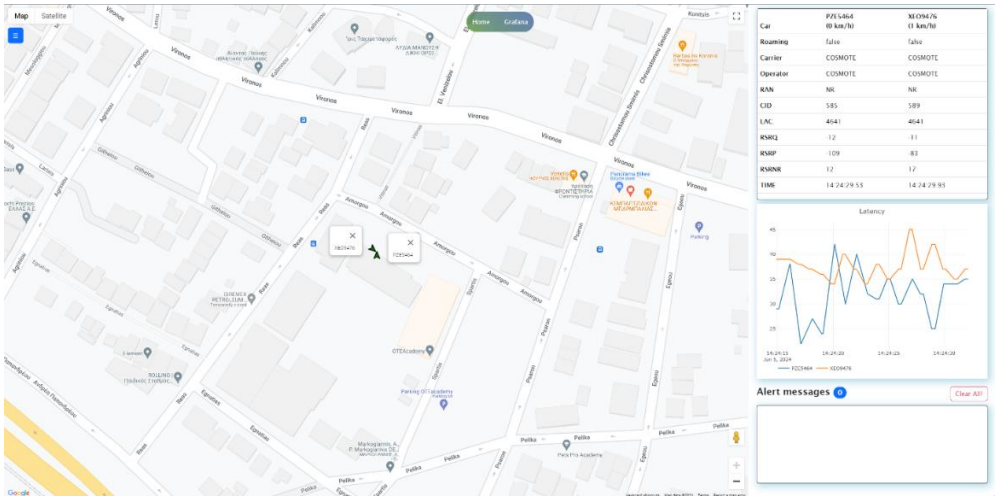

Figure 63 – Centralized Traffic Control WebGUI.

Developed as a web-based front end, the WebGUI connects to the MQTT broker to subscribe to live telemetry streams originating from both the On-Board Units (OBUs) installed in vehicles (e.g., the MILLA car) and the Android CAM applications used by vertical end users. The WebGUI is designed to scale with the ecosystem, accommodating multiple concurrent vehicles and user devices, and leveraging dynamic MQTT topic structures to subscribe to relevant data channels per unit. It serves as a bridge between the backend intelligence—where data is collected, processed, and distributed—and the end-user or control interface—where that data becomes actionable and observable.

8.3.2.2.3 Milla/ISFM On-board Unit (OBU)

The Milla autonomous shuttle through the OBU exchanges data and commands generated by the vehicle with its supervision system and the local traffic management system for the purpose of:

- Real-time vehicle tracking (position, status, route),
- Detection of anomalies and alerts (system errors, blockages, loss of connection),
- Dispatch of missions (next destination, definition of stop line and timetable, route avoidance),
- Communicating with operators and users (information messages, notifications).

The shuttle' real-time operating data (i.e. frequency of 1 Hz) includes:

- Door status (open/close)
- Geolocation
- Speed
- Driving mode (manual/autonomous)
- Emergency stop status (active/inactive)
- Vehicle at passenger stop status
- Vehicle technical problem alerts (alerts sent by the operator)
- The results of the vehicle's technical diagnostics
- Local trajectory of the autonomous vehicle

Within the OBU a WebRTC client is installed to feed the shuttle's camera stream toward the traffic management system and the vehicles that follow.

8.3.3 Integration of APIs

The CAM application at the Greek trial site consumes the following network exposure APIs:

Table 51 – APIs consumed by CAM applications at the Greek trial site.

Enabler	ENVELOPE API	Native API
Quality on Demand (QoD)	CAMARA Quality-on-Demand QoD	3GPP AsSessionWithQoS
Device Location	CAMARA Location Retrieval	3GPP MonitoringEvent
Edge Cloud (Edge Application Management)	CAMARA Edge Cloud (Edge Application Management)	K8s APIs, Helm APIs
Predictive QoS	CAMARA Predictive Connectivity Data	3GPP AnalyticsExposure (qosSustainInfos)
Multi-Connectivity	ENVELOPE ATSSS-T	ETSI MEC 015 MTS

In addition to the QoD and Device Location APIs from Deliverable D3.2, it also consumes service-oriented APIs detailed in this deliverable on Edge Application Management (see Section 5.2), Predictive QoS (see Section 3), and multi-connectivity (see Section 4).

Edge Application Management is an integral baseline to support the expandability of the solution and the mobility support regarding service continuity when crossing borders, as detailed in 8.3.1.

Predictive QoS is an important indication that is used to notify the vehicle passengers for performance modifications, leading to decisions for QoD requests.

Multi-connectivity through ATSSS is leveraged to improve the resilience of the webRTC stream to fluctuating network conditions and congestion through dynamic prioritization of the best path. By

using 3GPP and non-3GPP network paths simultaneously, ATSSS further aims to improve the throughput and hence the quality of the stream over using a single network path.

9 Next steps

This deliverable concludes the development of the ENVELOPE platform as part of WP3.

The next step is the finalization of the integration and testing activities as part of WP4, led by ICCS, which will result in project milestone MS7 at the end of project month M24. This milestone is led by TNO and will conclude the integration activities in all trial sites.

After concluding the development and integration of the ENVELOPE platform, the following step will be to plan and conduct large-scale demonstrations and technical evaluations on all trial sites as part of WP6, led by VICOM. The platform will be validated with respect to the targets defined in the CAM use cases, with the goal of demonstrating improvements in e.g., scalability, reactivity, and reliability. WP6 also includes the on-boarding of open call projects that leverage the ENVELOPE platform to investigate open B5G research questions and an assessment of the impact and user acceptance of the entire ENVELOPE platform.

10 Conclusions

This deliverable documents the implementation of three service-oriented API categories that enable vertical applications to leverage the functionality of the B5G system, as well as the developments of the zero-touch management framework and concrete CAM applications.

The three considered API categories comprise i) Predicted Quality of Service (PQoS) APIs, ii) Access Traffic Steering, Switching, and Splitting (ATSSS) APIs, and iii) Edge Computing APIs. The PQoS APIs are aligned with 3GPP standards and provide a simplified interface for predictive network analytics, including per-UE predictions. The implemented AI/ML pipeline facilitates training and inference of predictive models, and initial experiments show a trade-off between improved predictive performance and higher inference times depending on the model's size. The ATSSS APIs build upon the ETSI MEC 015 specification and provide multi-connectivity functionality to vertical applications. In addition to regular schedulers, the implementation provides an RL agent that learns how to adaptively balance traffic between 3GPP and non-3GPP network paths depending on the network conditions to improve the QoS over single-path transmissions. The Edge Computing APIs abstract three different native APIs and platforms (ETSI MEC, aerOS, and K8s) into CAMARA-conformant APIs that allow to manage the entire lifecycle of vertical edge applications, regardless of the underlying edge platform. Apart from the placement, migration, and orchestration of applications, the Edge Computing APIs also provide functionality to optimize the routing between UE and the deployed edge service instances.

In addition to the development of APIs, the ENVELOPE zero-touch management framework provides functionality for the flexible and efficient deployment of services at the Italian site. Users can express application deployment requests in high-level intent representations to automate experiment execution. The deployed applications are continuously monitored to detect and react to malfunctions and other incidents. Most management APIs used by the zero-touch management framework are internal and not exposed to experimenters, but it is possible to expand the framework with custom decision engines to configure application and network parameters.

The CAM applications developed at the three trial sites showcase how the functionality provided by the ENVELOPE EaaS platform can be leveraged by concrete applications. In various scenarios and use cases, the aim is to utilize advanced B5G functionality through the developed high-level APIs and technical advancements to demonstrate improved collective perception, service continuity, situation awareness, and behavior support.

Together with the foundational APIs documented in Deliverable D3.2, the APIs presented in this deliverable complete the specification and implementation of the ENVELOPE platform. Future work will focus on the integration of the APIs into the trial sites as part of WP4, as well as evaluating the APIs and technical innovations within the experiments and demonstrations in WP6.