ENVELOPE

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

Project deliverable D3.1

ENVELOPE experimentation as a service development description

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	D3.1 – ENVELOPE experimentation as a service development description
Status – version, date	Final – V6, 14/07/2025
Deliverable leader	LINKS
Contractual date of delivery	30/06/2025
Submission date	14/07/2025
Keywords	ENVELOPE architecture, EaaS, Enablers, Implementation

List of contributors

Name	Organisation
Matteo Zattoni	LINKS
Edoardo Bonetto	LINKS
Daniele Brevi	LINKS
Gergely Kovács	CMS
Anam Tahir	LUH
Amr Rizk	LUH
Ramon de Souza Schwartz	TNO
Nikolaos Petropouleas	LNVO
Apostolis Salkintzis	LNVO
Gorka Vélez	VICOM

Valeria Proietti Dante	TEO
Nicola di Pietro	HPE
Davide Montagno Bozzone	HPE
Elvina Gindullina	HPE
Harilaos Koumaras	NCSRD
Dimitris Uzunidis	NCSRD
Gabriele Scivoletto	NXW
Apostolis Siokis	IQU
Giorgos Drainakis	ICCS
Konstantinos Katsaros	ICCS
Pavlos Basaras	ICCS
Fofy Setaki	OTE
Fabrizio Gatti	TIM
Ezio Chiocchetti	TIM
Dimitris Tsolkas	FOGUS
Christos Milarokostas	FOGUS
Alex Kakyris	FOGUS
Zouzias Dimitris	eBOS

Quality control

	Name	Organisation	Date
Peer review 1	Gabriele Scivoletto	NXW	25/06/2025
Peer review 2	Zouzias Dimitris	EBOS	25/06/2025

Version History

Version	Date	Author	Summary of changes
01	20/05/2025	Edoardo Bonetto	Initial draft with ToC
02	30/05/2025	Ramon de Souza Schwartz, Gorka Vélez, Dimitris Uzunidis, Apostolis Siokis, Gergely Kovács, Apostolis Siokis	Initial contributions from Partners
03	10/06/2025	Matteo Zattoni, Anam Tahir, Ramon de Souza Schwartz, Gergely Kovács, Giorgos Drainakis	Further contributions and updates
04	19/06/2025	Daniele Brevi, Nikolaos Petropouleas, Pavlos Basaras	Updated contributions
05	07/07/2025	Edoardo Bonetto, Dimitris Uzunidis, Apostolis Siokis, Konstantinos Katsaros	Stable version
06	14/07/2025	Edoardo Bonetto, Konstantinos Katsaros	Final Version

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 experimentation framework and the main innovations developed in the project are: MEC with service continuity support, zero-touch management, multi-connectivity and predictive QoS.

It will deliver 3 large scale Beyond 5G (B5G) trial sites in Italy, The 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 Automated Mobility (CAM) vertical, the developments resulting from the Use Cases (UCs) 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 activities that implement the ENVELOPE platform and related innovations. The ENVELOPE platform is a Beyond 5G (B5G) System Experimentation as a Service (EaaS) platform that provides simplified and extended interactions with the 5G network. Task 3.1 “Development of the ENVELOPE experimentation as a service” focuses on the EaaS approach that is offered within the ENVELOPE project. This deliverable provides the outcomes of Task 3.1, introducing the refinement of the ENVELOPE platform architecture and its realisation across the trial sites, and it reports on the design and implementation of the EaaS module.

The ENVELOPE platform architecture in Task 3.1 has been refined compared to the one introduced in the ENVELOPE deliverable D2.2 “ENVELOPE platform architecture requirements and specifications” to better detail some architectural aspects and capture the advances brought since D2.2 (M12). In particular, this deliverable provides a greater level of detail about the ENVELOPE Enablers, which are the extensions to the B5G System enabling easy access to the functionalities made available by the B5G System.

Further activity performed in Task 3.1 was intended to identify the common functionalities and the common architectural denominator of the ENVELOPE platform in the project’s trial sites. Starting from the requirements introduced in D2.2, the functionalities have been defined and the common architectural denominator to support these functionalities in all trial sites is identified.

The implementation activity of Task 3.1 mainly concerns the development of the EaaS module. The initial architecture of the EaaS module, that was introduced in D2.2, has been detailed and analysed to identify the functionalities to be offered for the Experimentation as a Service approach. Furthermore, operations and interfaces of the EaaS module have been defined. The following step was the implementation of the inner components of the EaaS module. Details about the implementation are reported in this deliverable illustrating the inner component architecture, the interfaces used and the software dependencies.

The outcomes reported by this deliverable are essential to define a common interoperable ground on top of which the other implementation activities within WP3 are based. This will also be beneficial to WP4 for having a smooth integration of the ENVELOPE platform in all trial sites. The content devoted to the EaaS module implementation will be helpful to the external experimenters, such as the participants to the ENVELOPE Open Calls, to understand how to interact with the ENVELOPE platform and its working principles. This will allow them to understand how to test their use cases exploiting the Experimentation as a Service offered by ENVELOPE.

List of abbreviations and acronyms

Acronym	Meaning
3GPP	3rd Generation Partnership Project
5GS	5G System
5QI	5G QoS Identifier
AE	Application Entity
AF	Application Function
AI	Artificial Intelligence
AMF	Access & Mobility Management Function
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
APPM	Application Package Management
AT-SSS	Access-Traffic Steering, Switching & Splitting
B5G	Beyond 5G
CAM	Connected and Automated Mobility
CAPIF	Common API Framework (3GPP)
CN	Core Network
CPU	Central Processing Unit
DF	Deployment Flavour
DN	Data Network
DOM	Document Object Model
EC	European Commission
EaaS	Experimentation as a Service
EES	Edge Enabler Server

ETSI	European Telecommunications Standards Institute
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTML	Hypertext Mark-up Language
HTTP	Hyper-Text Transfer Protocol
IDAN	Intent-Driven Autonomous Network
JSON	JavaScript Object Notation
JSON-LD	JSON-Linked Data
KPI	Key Performance Indicator
K8s	Kubernetes
Liqo	multi-cluster Kubernetes framework
MANO	Management and Orchestration
MEC	Multi-access Edge Computing
MEO	MEC Orchestrator
MEPM	MEC Platform Manager
MF	Manifest File
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
MP-QUIC	Multipath QUIC
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
N3IWF	Non-3GPP InterWorking Function
NEF	Network Exposure Function

NGSI-LD	Next-Generation Service Interfaces – Linked Data
NWDAF	Network Data Analytics Function
OBU	On-Board Unit
OSM	Open Source MANO
PCF	Policy Control Function
PDU	Protocol Data Unit
PMF	Performance Measurement Function
PQoS	Predictive Quality of Service
PQoS-I	PQoS Inference
PQoS-T	PQoS Training
PU	Public
QoD	Quality on Demand
QoS	Quality of Service
RAM	Random-Access Memory
RAN	Radio Access Network
REST	Representational State Transfer
RL	Reinforcement Learning
RTT	Round Trip Time
SDK	Software Development Kit
SNI	Server Name Indication
SNS JU	Smart Networks & Services Joint Undertaking
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security

TMF	TeleManagement Forum
TOSCA	Topology & Orchestration Specification for Cloud Applications
UC	Use Case
UDM	Unified Data Management
UE	User Equipment
UI	User Interface
UPF	User Plane Function
URL	Uniform Resource Locator
VDU	Virtual Deployment Unit
VM	Virtual Machine
YAML	YAML Ain’t Mark-up Language
YANG	Yet Another Next Generation
ZTM	Zero-Touch Management

1 Introduction

1.1 Purpose of the deliverable

This deliverable reports the outcomes of the activities carried out in Task 3.1 “Development of the ENVELOPE experimentation as a service” whose main objective is to define the common technical aspects of the ENVELOPE platform to support the implementation of the Experimentation as a Service (EaaS) approach. This technical basis is meant to provide a common denominator on which to develop the implementation activities in WP3 and to serve as a reference for an interoperable implementation of the three trial sites.

To achieve this objective, the activities of Task 3.1 focused on two main aspects: the first one was to define the common denominator in the different layers of the platform architecture; the second is the implementation of the EaaS module that enables the interaction of experimenters with the ENVELOPE platform.

The design of the common architectural aspects was based on the analysis of the requirements, that have been elicited in WP2, and on the ENVELOPE platform architecture introduced in the ENVELOPE deliverable D2.2 “ENVELOPE platform architecture requirements and specifications”[1]. This analysis led to the definition of the functionalities that the three trial sites must support and to the design of the common denominator of each layer of the ENVELOPE platform.

The implementation of the EaaS module started from the architectural concept introduced in the ENVELOPE deliverable D2.2. First, this concept was detailed both from an architectural and functional point of view detailing the functionalities to be supported, the operational workflows, and the interfaces of the inner components of the EaaS module. Technical development of the various components followed this step to provide a common implementation of the module to be used in the three trial sites.

This document provides the details of the aforementioned activities, and it will serve as a reference for the other implementation activities in WP3 and in the trial site integration in WP4. Furthermore, it will be helpful to external experimenters that want to understand the architecture of the ENVELOPE platform and the Experimentation as a Service approach offered within the ENVELOPE project.

1.2 Intended audience

The dissemination level of D3.1 is ‘public’ (PU); thus, the deliverable is available to members of the consortium, the European Commission (EC) Services and those external to the project. This document is primarily intended to serve as an internal guideline and reference for all ENVELOPE beneficiaries, in particular the partners involved in the implementation activities of WP3 and in the trial site integration activities in WP4. It is also available to the external experimenters who participate in the ENVELOPE Open Calls and to any interested reader who wishes to learn about the ENVELOPE platform architecture and the Experimentation as a Service approach.

[1] https://envelope-project.eu/resource/?e-filter-549b21b-resource-categories=deliverables

1.3 Structure of the deliverable

The structure of the deliverable is as follows:

- Section 2 presents the evolution of the ENVELOPE platform architecture, introducing additional details with respect to the architecture introduced in deliverable D2.2; furthermore, it provides the information about how the ENVELOPE platform architecture is realized by illustrating the common functionalities and the common denominator in each layer of the ENVELOPE architecture;
- Section 3 describes the ENVELOPE Enablers, which are the add-ons to the B5G system, that are introduced in the ENVELOPE project; the ENVELOPE Enablers make the offered functionalities of the B5G system easily accessible via the ENVELOPE APIs.
- Section 4 describes the implementation activities of the EaaS module.
- Section 5 reports the conclusions and the next steps.

2 ENVELOPE Architecture Realisation

This Section describes the overall architecture of the ENVELOPE Platform. Based on the architecture introduced in D2.2, the overall architecture has evolved to clearly define the distinctive features, in particular the ENVELOPE Enablers.

2.1 Evolution of the ENVELOPE Platform architecture

The ENVELOPE Platform is based on a layered architecture spanning from the infrastructure layer up to the application layer. Two transversal layers are also included: the MANO Layer and the EaaS module. Figure 1 shows a high-level view of the ENVELOPE Platform architecture.

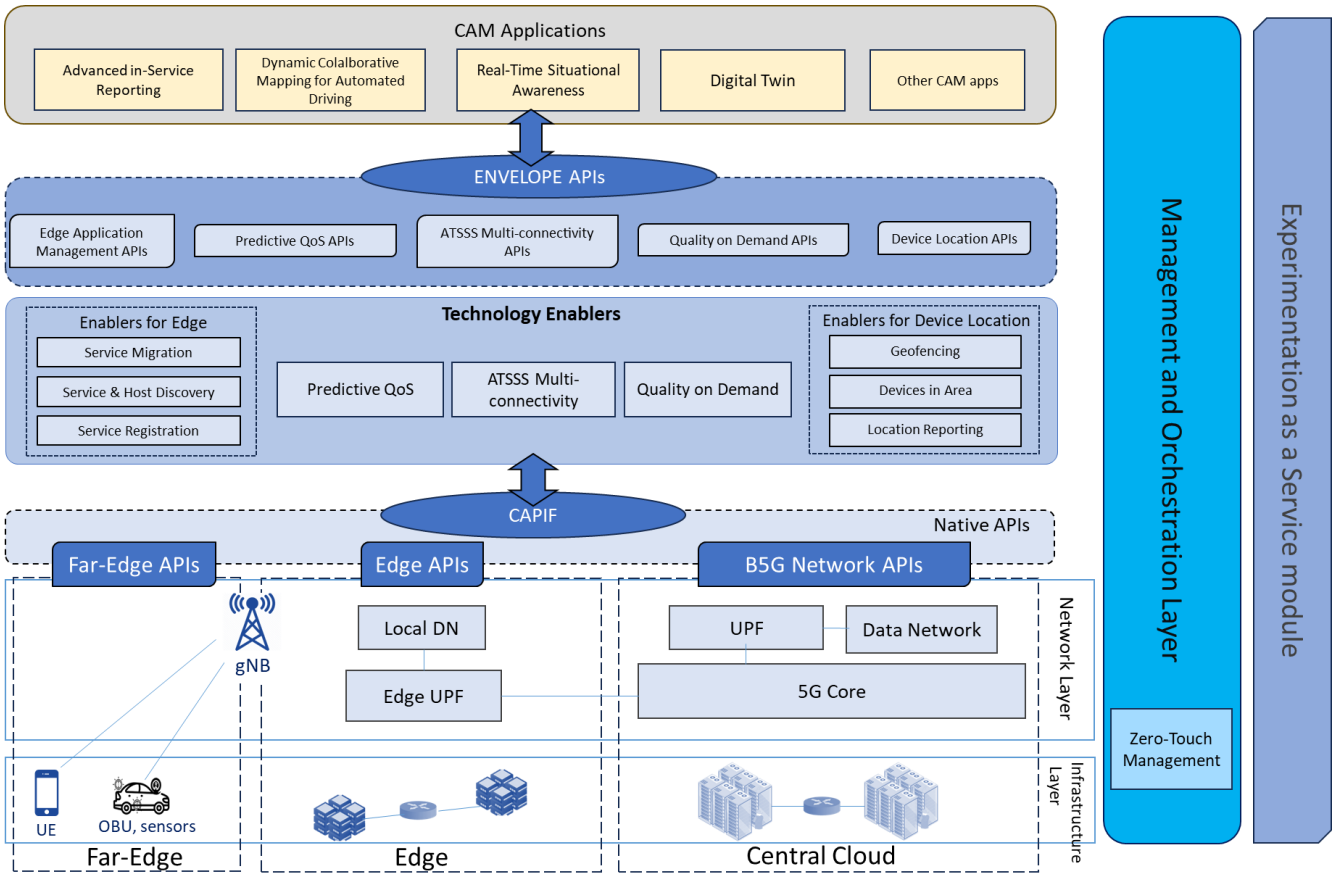

Figure 1 – high-level view of the ENVELOPE architecture

The EaaS module serves as the entry point for experimenters. The EaaS module handles all operations related to the experimentation. The experimenter leverages the functionalities provided by the EaaS module for uploading experiment-related information and material, and managing its lifecycle. An overview of the EaaS module is provided in Section 0, and the details about its implementation are reported in Section 4. The EaaS module exploits the MANO layer for resource allocation and application deployment. An outline of the main characteristics of the MANO layer is reported in Section 2.3.

The ENVELOPE Platform relies on the network and the infrastructure layers that can be distinguished into the Cloud, Edge, and Far-Edge segments. The Cloud segment comprises the B5G Network and the Central Cloud resources. The Edge segment includes at the network layer the local Data Network and the edge UPF, and at the infrastructure layer it is composed of the edge computing resources on which applications can be deployed. At the Far-Edge segment, there is the gNB at the network layer, and UEs, such as the On-Board Unit (OBU) installed on vehicles, at the infrastructure layer.

The Native APIs of each of these three segments of the network and the infrastructure layers are made accessible to applications through the ENVELOPE Enablers layer. The Native APIs therefore encompass: i) B5G Network APIs for interactions with the B5G core; ii) Edge APIs, that provide access to UPFs at the edge level; iii) Far-Edge APIs enabling interactions with RAN and UE resources. The Native APIs can be accessed by the Enablers through an approach based on Common API Framework for 3GPP northbound APIs (CAPIF).

Each Enabler, which consists of one or more software components, interconnects one or more Native APIs to an ENVELOPE API that is exposed northbound to the CAM application. The ENVELOPE APIs are designed to be easy-to-use interfaces hiding the complexity of the Native APIs. To this end, the Enabler aims to facilitate the interaction between an Application and a Native APIs by implementing an intermediate translation layer that transforms the complex Native interfaces into easy-to-consume APIs.

ENVELOPE Enablers include components for edge operations, device location, Quality on Demand (QoD), predictive QoS, and Access Traffic Steering-Switching-Splitting (ATSSS) Multi-connectivity. Details about their realisation are reported in Section 3.

2.2 EaaS module

The experimentation operations in the ENVELOPE project are managed by the EaaS module. The experimentation includes all the actions that are needed to define and launch an experiment, which consists of the deployment of one or more CAM applications that can leverage the advanced 5G features enabled by the ENVELOPE platform.

The EaaS module architecture is presented in Figure 2. The experimenter, which is the entity that aims at testing CAM applications in one of the ENVELOPE trial sites, engages with the ENVELOPE Portal to carry out various actions, such as initiating experiments, accessing data, or loading specific applications. This portal serves as the exclusive gateway to the functionalities provided by the ENVELOPE Platform. It integrates both a Graphical User Interface (GUI) and connections to the backend components of the EaaS module. Before performing any task, the experimenter must complete the authentication process to gain operational access to the portal.

To conduct experimentation on the ENVELOPE Platform, applications are assembled into what is known as a vertical service, essentially a linked sequence of applications. These applications are stored in a repository that holds all necessary deployment artefacts, such as Helm Charts[2]. The Management and Orchestration Layer accesses these artefacts to instantiate the vertical service.

[2] https://helm.sh/docs/topics/charts/

Through the ENVELOPE Portal, the experimenter can manage these artefacts by uploading new versions, making updates, or removing them when needed.

During an experiment, the Monitoring interface enables the experimenter to keep track in real-time of the KPIs and metrics related to the B5G infrastructure and to the ongoing experiment. It offers visibility into application logs and metrics related to both network and computing resources. This information can either be viewed directly or downloaded for further analysis. The monitoring platform module is in charge of gathering the requested logs, metrics, and related data from other components within the ENVELOPE Platform.

The interface dedicated to the experimentation lifecycle provides access to the key functions needed to oversee experimentation processes, including initiating, and terminating experiments. Through this interface, the experimenter can also upload a descriptor file, which outlines the configuration and details of the experiment. This file specifies the composition of the vertical service and indicates the resources required for its deployment.

The implementation details of the EaaS module, including its functionalities and operations, and of its components are reported in Section 4.

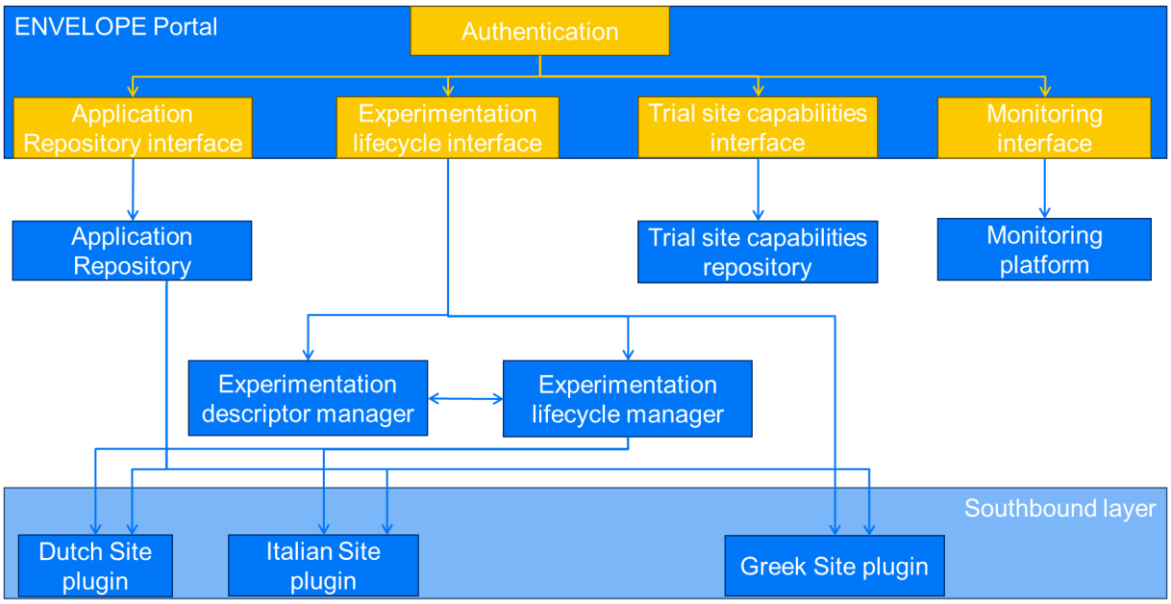

Figure 2 – EaaS module architecture

2.3 Management and Orchestration Layer

The Management and Orchestration (MANO) Layer facilitates agile and efficient service delivery by integrating various components into a unified framework. It manages and coordinates computing and network resources, along with the services operating on virtualized infrastructures, enabling automated application deployment, scaling, and performance optimization. For more information, the reader is referred to D2.2 “ENVELOPE platform architecture requirements and specifications”, paragraph 4.3.1.

In the Italian trial site, the MANO layer includes the trustworthy translation and reconciliation of declarative intents, which represent the cornerstone of Zero-Touch Management (ZTM). Part of this ZTM module is an Intent Engine, a module which enables users to express service requests

in high-level intent representations streamlining network operations and translating user intent into actionable tasks. The actions based on the Intents are handled by the respective Intent Handler (see Figure 3).

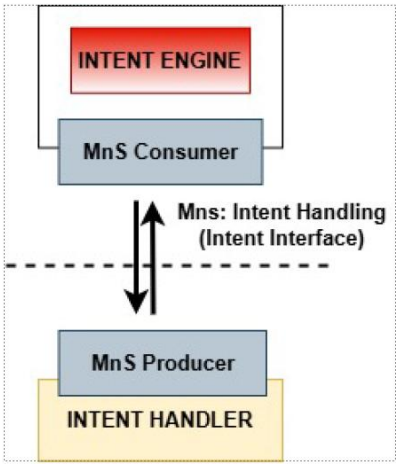

Figure 3 – Intent Handling

Each Intent Handler will be part of a respective Decision Engine. The latter will perform some action related to a specific CAM application. The Decision Engines can be viewed as Zero Touch Closed Loops. Thus, there can be one common Intent Engine and many Decision Engines/Closed Loops (one per application and/or a trial site). This is possible since all intents are translated into a common TMF forum representation (following TMF 921 standard[3], which is part of the TMF IDAN Catalyst project). Only the translation of these intents into a format that it is application specific differs (the Intent handling).

As part of the actual demonstrations, the Intent Engine is used in the Italian trial site. Figure 4 shows the specific integration of the MANO and ZTM modules with the Intent Engine and the specific Decision Engine used in the Italian site. The description of the individual modules can be found in D2.2.

[3] https://www.tmforum.org/resources/how-to-guide/tmf921a-intent-management-api-profile-v1-1-0/

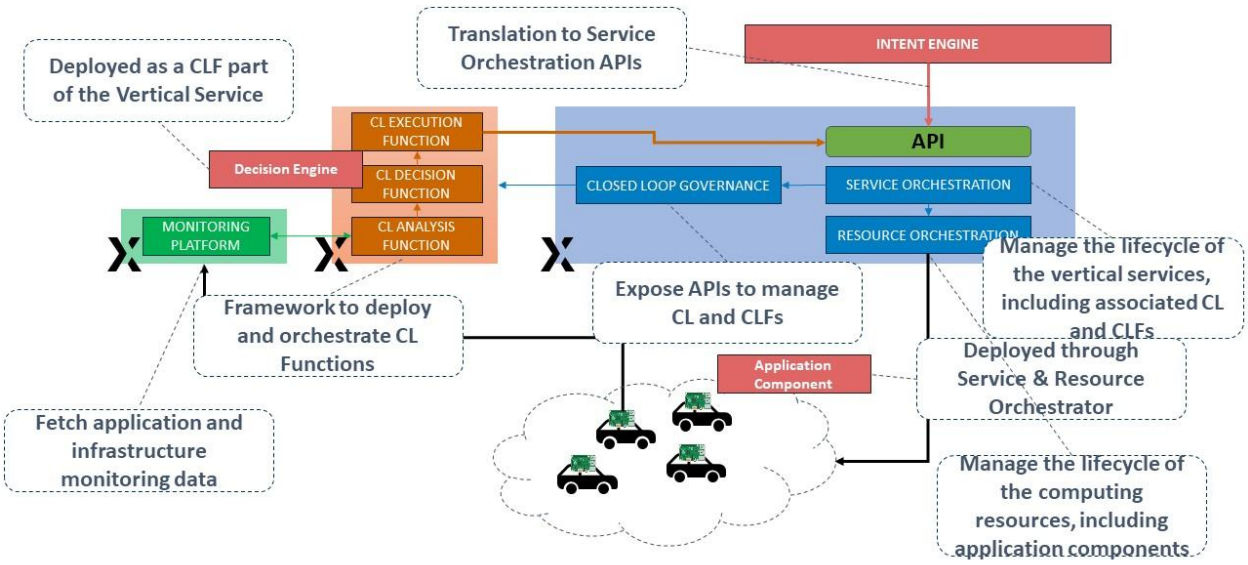

Figure 4 – Integration of the Zero Touch Management modules with the Intent and Decision Engines in the Italian site

2.4 B5G System

The ENVELOPE architecture is applied within the three trial sites through a layered approach where **ENVELOPE Enablers** act as software components that consume **southbound APIs** (Far-Edge APIs, Edge APIs, and B5G Network APIs) to interact with the underlying 5G network infrastructure, edge, and far-edge deployments capabilities. These enablers expose **ENVELOPE APIs,** which, on the one hand, abstract network complexities, and, on the other, combine network functionalities with edge capabilities. These northbound, simplified APIs act as an intermediate abstraction layer that translates the complex 5G interfaces and edge services into easy-to-consume interfaces, which should be easily accessible by vertical services.

Each trial site becomes part of the ENVELOPE Platform by integrating the selected, for the use case, ENVELOPE enablers and thus, exposing the corresponding ENVELOPE APIs. These APIs are further registered with OpenCAPIF, an implementation of CAPIF framework, ensuring standardized and secure API exposure. This modular architecture allows each trial site to selectively deploy specific enablers and supporting technologies (such as ATSSS Multi-connectivity, Predictive QoS, Dynamic Slicing, or Zero Touch Management) based on their infrastructure capabilities and use case requirements, while maintaining interoperability through the common ENVELOPE API layer.

The common B5G system functionalities of the three trial sites are a result of the common requirements (six functional and two non-functional) identified by the trial site leaders. These functionalities are the following:

1. APIs exposure to untrusted applications – N33 (REQ-F-Task3.2-B5GS-1)

- It enables 5G systems to securely expose network APIs to third-party (and potentially untrusted) applications through the NEF via the N33 interface as defined in 3GPP TS 23.501.

2. NEF authorization (REQ-F-Task3.2-B5GS-5)

- It ensures that the NEF can determine whether external Application Functions are authorized to interact with specific network APIs as defined in 3GPP TS 33.501.

3. Containerised and Virtualised Network Functions implementations (REQ-F-Task3.2-B5GS-9)

- It enables dynamic scale-in and scale-out operations of B5G network functions through containerization and virtualization technologies.

4. Quality on Demand (QoD) (REQ-F-Task3.2-B5GS-10)

- It provides dynamic, on-demand QoS reconfiguration capabilities for connected mobile devices (UEs), enabling applications to request specific quality parameters.

5. Device Location (REQ-F-Task3.2-B5GS-11)

- It enables subscription to device location updates, providing applications with real-time mobility information essential for location-aware services.

6. Dynamic UE QoS PDU session configuration (REQ-F-Task3.2-B5GS-16)

- It allows for dynamic modification of QoS parameters for UE PDU sessions, enabling adaptive network behaviour. The NEF should expose AsSessionWithQoS API through N33 interface to untrusted VS AF as defined in 3GPP TS 29.522.

7. NEF integrity, relay, confidentiality protection (REQ-NF-Task3.2-B5GS-25)

- It ensures that all communications between the NEF and external Functions are protected using TLS-based security mechanisms according to 3GPP TS 33501.

8. Management API security (REQ-NF-Task3.2-B5GS-28)

- It ensures that all management APIs exposed by the B5G system implement TLS-based security.

It is worth mentioning that despite the common ENVELOPE architecture and the common functionalities, each trial site is allowed to tailor specific implementations based on their infrastructure capabilities and use case requirements.

2.5 Edge computing architecture

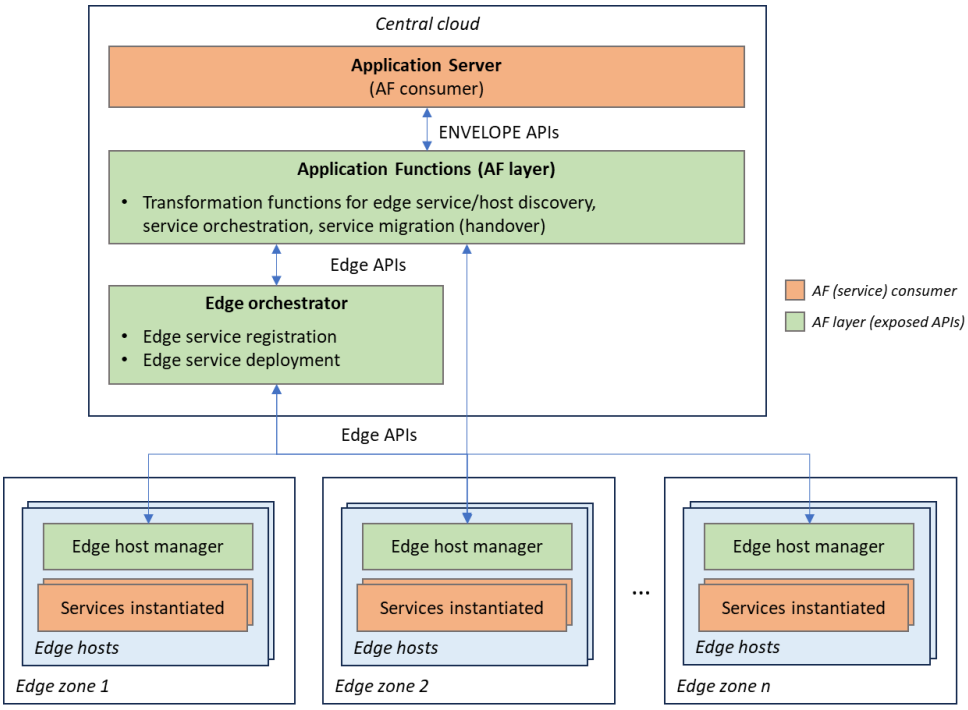

Figure 5 – Common denominator edge architecture across the trial sites

In the ENVELOPE project, a common cloud-native edge architecture, which includes a set of enabling edge features to support the edge computing requirements listed in D2.2, is defined. The ENVELOPE approach relies on a containerised architecture design where services are deployed in containers in the different edge hosts. In this way, the ENVELOPE edge architecture abstracts specific implementation choices with respect to the underlying standardisation specifications (e.g., ETSI MEC[4]) and/or open-source solutions (e.g., Kubernetes-based[5]) selected by each trial site. This sub-section introduces the common denominator edge architecture identified across the three trial sites that includes the following enabling functions: service/host discovery, edge service orchestration (service registration and instantiation), and service migration (handover).

Figure 5 shows the common denominator of the cloud-native edge architecture for all trial sites. Application Functions (AFs) hosted at the central cloud perform transformation functions between the ENVELOPE APIs and edge APIs for the underlying edge infrastructure. In each telco operator domain, edge hosts are logically grouped into different 'edge zones' that could identify, e.g., a group of edge hosts as part of the same physical server infrastructure or as part of the same geographical region within a country.

[4] https://www.etsi.org/technologies/multi-access-edge-computing
[5] https://kubernetes.io/

The ‘edge host manager’ performs tasks similar to those specified in ETSI MEC platform or 3GPP[6] Edge Enabler Server (EES) by keeping a list of available running services in the host and offering the discovery of these services.

The edge orchestrator is responsible for the registration and deployment (instantiation) of services in selected edge zone(s) and in the corresponding edge host within the edge zone. Service migration (handover) is also enabled by the orchestrator by moving a service to a desired edge zone, e.g., by removing a service from the current edge zone and instantiating it in the desired new edge zone. Edge orchestrator interacts with the underlying MANO solution available at the edge infrastructure.

The common edge functionalities of the three trial sites are a result of the common requirements identified by the trial site leaders:

1. MEC service discovery (REQ-F-Task3.4-MEC-1)

- The discovery of services available in the edge platform is enabled so that vertical services can find and retrieve information about services running in the edge infrastructure.

2. MEC handover (REQ-F-Task3.4-MEC-3)

- The handover (migration) of applications among edge servers should be supported. This is done to ensure an instance of a specified application is deployed in the desired edge zone, e.g., to meet computing and network requirements.

3. MEC service registration (REQ-F-Task3.4-MEC-4)

- The dynamic registration of a new service at the edge service registry is supported. This is required to enable service discovery and it contains information of how the registered services should be instantiated (e.g., tools and network ports required).

4. MEC host discovery (REQ-F-Task3.4-MEC-6)

- The discovery of available edge hosts is supported for vertical services. This enables the selection of hosts according to specified computing and network requirements.

[6] https://www.3gpp.org/

3 ENVELOPE Enablers Realisation

3.1 Enablers per site/UC

In this section, an overview of the different enablers developed in the ENVELOPE project is provided. The various enablers are being implemented and tested at the different pilot sites in order to provide some distinctive features for each of them and, mainly, to support each specific use case of the ENVELOPE project. This will also allow for a better targeting of the various open call applications, directing them to the appropriate sites based on the enablers of interest. In Table 1, the reader can find the list of enablers and their mapping to the various pilot sites:

Table 1 – Enablers per Pilot Site

Enabler	Greece	Netherlands	Italy
PQoS	X		
ATSSS	X		
Edge enabler	X	X	X
Geofencing		X	
Devices in Area			X
Location Reporting	X		
QoD	X	X	X

3.2 Enabler PQoS

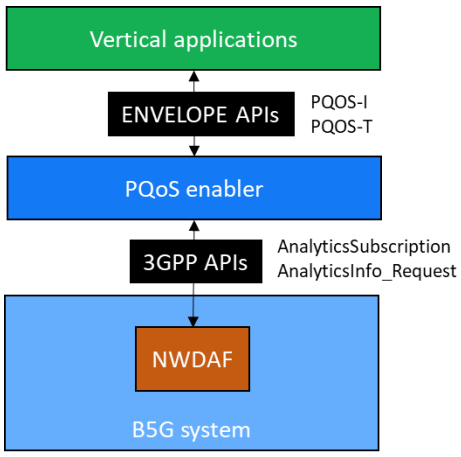

Figure 6 – High-level overview of interfaces used by the PQoS enabler

The PQoS enabler provides two key functionalities:

a. It allows vertical applications to obtain analytics and subscribe to notifications about predicted changes in Quality of Service (QoS) Key Performance Indicators (KPIs) at either the network or application level, e.g., anticipated downlink (DL) throughput degradation (PQoS Inference APIs - **PQoS-I**).
b. It enables vertical services to configure the ENVELOPE platform to train PQoS AI/ML prediction models (PQoS Training APIs - **PQoS-T**).

The enabler supports three distinct PQoS model variants:

- **QoS Sustainability**: Focused on network-level performance metrics, primarily derived from Radio Access Network (RAN) and Core Network features e.g., cell load, 5G Quality Indicator (5QI), RAN User Equipment (UE) bitrate, etc.
- **DN Performance**: Focused on application-level (end-to-end) performance metrics, primarily based on application layer features, e.g., UE location, application identifier, Latency (Round Trip Time – RTT), etc.
- **Hybrid**: Combines both network-level and service-level features for enhanced prediction accuracy.

A high-level overview of the APIs consumed and exposed by the PQoS enabler is depicted in Figure 6. In terms of Southbound APIs, the PQoS enabler interacts with B5G via the NWDAF component using 3GPP-standardized APIs (NWDAF::AnalyticsSubscription, NWDAF::AnalyticsInfo_Request) to gather the necessary information for both inference (e.g., QoS predictions, notifications, etc.) and training phases. The PQoS enabler serves as a transformation layer thereafter that abstracts the complexities of the 3GPP APIs, presenting simplified, developer-friendly Northbound APIs to the ENVELOPE platform. The following Northbound (ENVELOPE) APIs are supported towards vertical applications:

a. PQoS Inference Analytics: Enable ad-hoc/runtime predictions for a selected model variant and QoS KPI (e.g., DL throughput via QoS Sustainability).
b. PQoS Inference Subscriptions: Manage subscriptions to the PQoS-I service for a specified model variant, QoS KPI and target threshold. Such subscriptions trigger event-based notifications to the application's designated server endpoint, when the KPI does not meet the pre-defined target threshold (e.g., DL throughput < 50 Mbps, Latency > 100 ms).
c. PQoS Training Requests: A set of APIs for the development of new AI/ML QoS prediction models, as well as the maintenance (e.g., re-training, tuning, etc.) of existing ones.

3.3 Enabler ATSSS

ENVELOPE ATSSS-like implementation is established through Non-3GPP Interworking Function (N3IWF), enabling both 3GPP and non-3GPP access. Figure 7 below illustrates the ENVELOPE ATSSS architecture based on N3IWF at the Greek trial site, incorporating UE and core network-side proxies (red boxes). It highlights the ATSSS policy function for UE-to-core coordination, which is responsible for establishing rules and policies across the different access paths, as well as the implementation of MP-QUIC functionalities that enforce the ATSSS policy and dynamically manage MP-QUIC subflows, based on dynamic network conditions. This enabler is solely implemented at the Greek trial site with no common functionalities among the other trial sites.

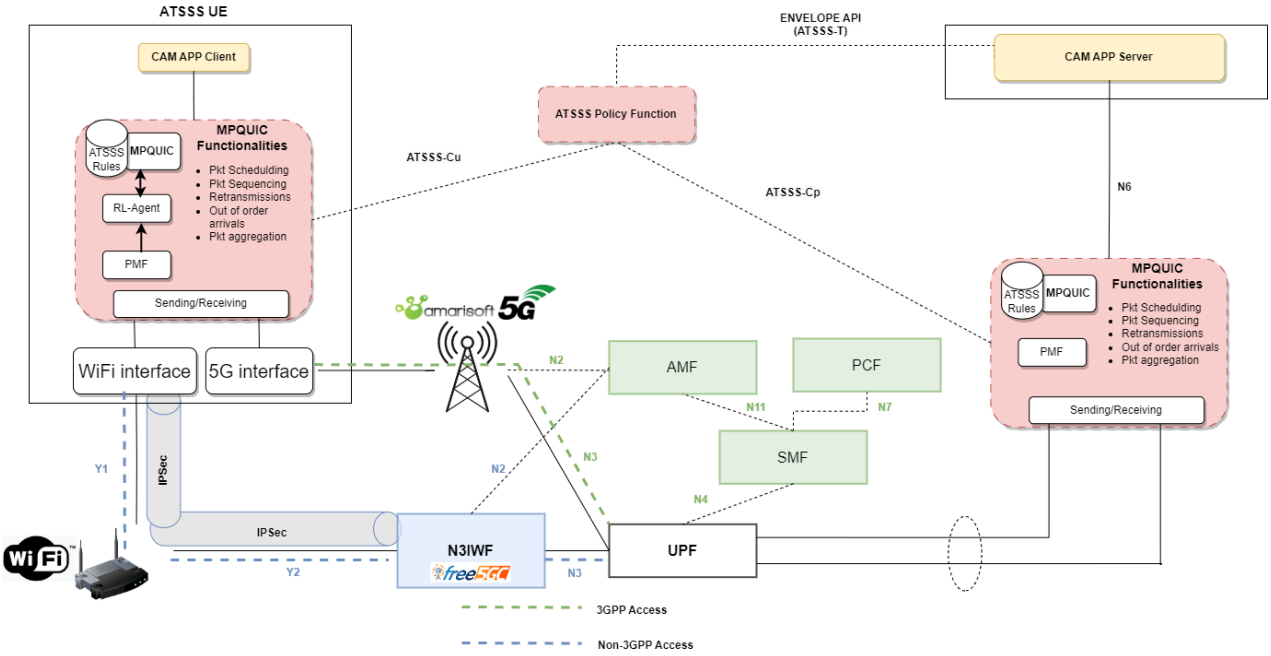

Figure 7 – Architecture overview of the ATSSS enabler showing the exposed and used interfaces

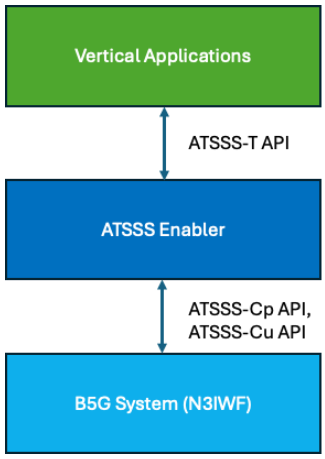

Figure 8 – High-level interfaces used by the ATSSS enabler

The ATSSS enabler provides multi-connectivity functionality at the Greek site to enable the use of multiple network paths and provide redundancy, seamless transition, and improved performance. When triggered by the CAM application, ATSSS-like multi-connectivity activates a non-3GPP compliant (Wi-Fi) path within the network that is utilized by both the UE. We implement MP-QUIC to enable traffic control over both paths.

The CAM application uses the ATSSS-T API to initiate an ATSSS multi-connectivity service, along with the traffic rule that is designated for application by the network. The available set of ATSSS traffic rules includes: (i) minimum RTT/smallest delay, (ii) load-balancing, (iii) active-standby, and (iv) priority-based path selection.

If the rule requested by the CAM application belongs to this set and is applicable at the time of the request, the ATSSS policy function sends back a rule reception acknowledgement and then informs

the UE and Core network proxies about the traffic rule to be applied, via the ATSSS-Cp and ATSSS-Cu APIs, respectively. Figure 8 illustrates both the high-level overview of ATSSS integration into the ENVELOPE architecture, and the utilized (exposed) APIs. The ATSSS-T API is northbound and is exposed to the CAM application function.

Additionally, for the load balancing and priority-based selection rules, the application can provide initial rule parameters to instantiate the rule application. For example, if the CAM application requests the load balancing rule, it can also send the initial traffic split ratio to be applied at the initial rule application. During rule operation the reinforcement learning (RL) agent residing in the UE will determine the appropriate rule parameter, i.e., traffic split ratio based on the network conditions for both paths. The split ratio decided by the RL-based UE will also be communicated to the core network proxy for the downlink traffic through the ATSSS-Cp and ATSSS-Cu APIs. Network conditions measurements per link, are realized mainly via the Performance Measurement Function (PMF) that collects the respective metrics/statistics, that will be used by the MP-QUIC framework and RL agent for the dynamic traffic split. These measurements are exploited by MP-QUICs internal in-band measurements to ensure timely path condition status as input to the RL agent.

3.4 Enablers for Edge

In this Section, the different approaches adopted at the trial sites for the Enablers for Edge are introduced. The implementation of these enablers is closely tied to the specific infrastructure deployed at each trial site. In particular, the edge layer at a given site exposes edge functionalities in different ways. As a result, each trial site follows a different approach for consuming the offered functionalities at the edge layer.

3.4.1 ETSI MEC-based enabler

At the Italian trial site, edge functionalities are enabled through the exposure of low-level ETSI MEC APIs, which are made accessible via simplified CAMARA interfaces. This approach ensures compliance with ETSI MEC standards while also aligning with the CAMARA initiative to provide developer-friendly APIs for vertical applications.

Figure 9 depicts the architecture deployed at the Italian site, highlighting in yellow the components involved in the support of edge functionalities.

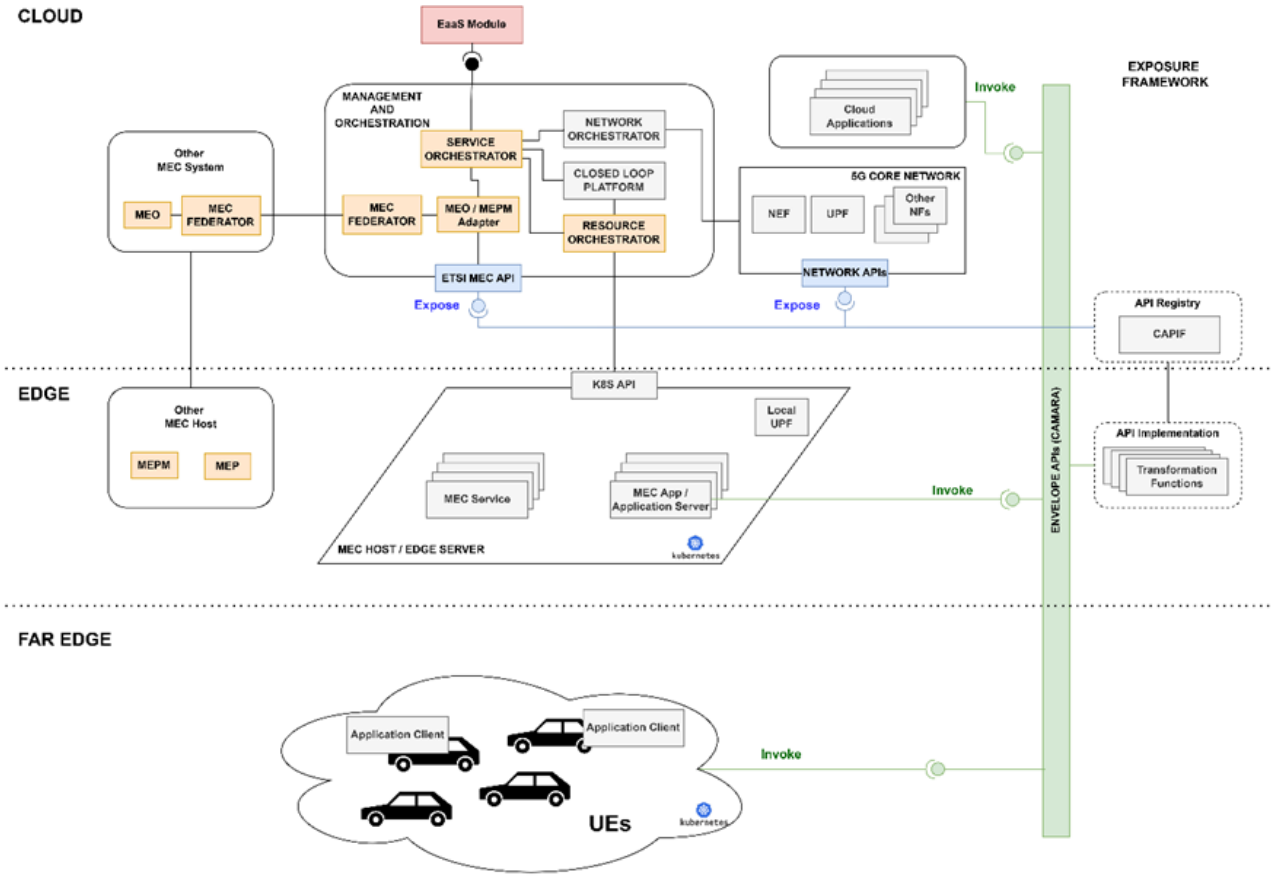

Figure 9 – Italian Site Architecture – components in yellow are the ones contributing to edge enabler.

The following key components are responsible for implementing the required MEC capabilities:

- **MEO/MEPM Adapter**: This component acts as a bridge between the ETSI MEC APIs and the Nextworks Service Orchestrator. It provides the functionality defined in ETSI MEC GS MEC 010-2 for the management of application packages and application instances. Specifically, the adapter handles:
- Onboarding of application packages.
- Instantiation and termination of application instances.
- **Service Orchestrator**: This orchestrator receives requests from the MEO/MEPM Adapter and coordinates with the Resource Orchestrator to manage the underlying edge infrastructure. It offers functionalities to:
- Deploy application components on appropriate computing resources.
- Allocate these components to edge hosts that best match the service's computing and networking requirements.
- **Resource Orchestrator**: Responsible for the management and abstraction of computing resources, including edge hosts. It plays a central role in fulfilling host discovery and service instantiation functionalities by interacting with Kubernetes clusters across the edge zones.
- **MEC Federator**: This component handles MEC federation tasks, facilitating cross-domain orchestration and the support of service migration across edge zones in line with MEC handover requirements.

In the Italian site, ETSI MEC APIs are exposed through the ENVELOPE CAMARA Edge Cloud APIs by means of a dedicated exposure framework. This enables CAMARA-compliant applications to consume edge services in a standardized and simplified manner, ensuring interoperability across the ENVELOPE ecosystem as depicted in Figure 10.

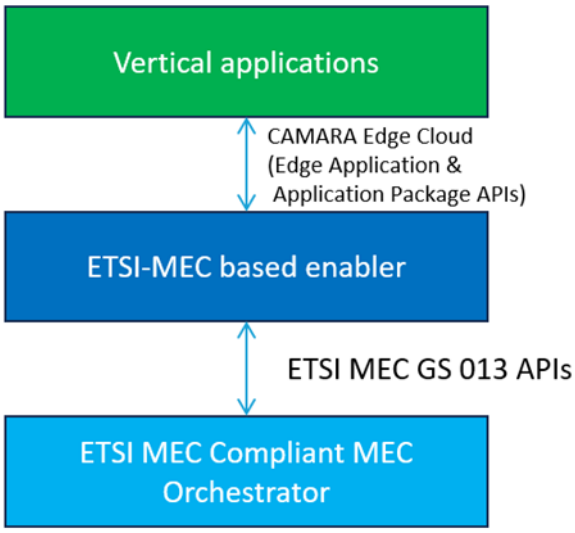

Figure 10 – Interfaces used by the ETSI-MEC based enabler

3.4.2 AerOS[7] enabler

In the Greek trial site, the aerOS enabler provides an abstraction layer over the core aerOS APIs to expose continuum resource management and orchestration functionalities through standardized API interfaces. This enabler allows external actors or applications to interact for querying resource availability, initiating service deployments, and managing orchestration lifecycles across the continuum.

The capabilities abstracted from the enabler are those of a MetaOS[8] approach to intelligently manage and integrate a distributed set of network and computing resources into a seamless computing continuum. aerOS enables the orchestration of hyper-distributed applications across the Edge–Cloud spectrum. Diverse computing resources, whether geographically dispersed or under different administrative domains, are federated and abstracted to uniformly expose their capabilities, allowing dynamic service allocation upon request.

It is a cloud-native ecosystem that exposes federation and orchestration capabilities, based on which each domain of the continuum can be queried for real-time resource availability and capabilities, and it can receive deployment requests to allocate resources for hosting service components, which are expected to be containerized.

The aerOS enabler primarily wraps two categories of APIs that enable interaction with the continuum. The first category includes APIs used to query registered entities within the continuum,

[7] https://aeros-project.eu/
[8] https://meta-os.eu/

such as computing resources (Infrastructure Elements), zones (domains), services, and service components. These aerOS APIs are implemented following the ETSI NGSI-LD protocol, providing responses as JSON-LD objects[9]. Responses may include complete entity descriptions or filtered subsets based on specific attribute requests, ensuring that the state and details of aerOS continuum entities are consistently accessible.

These APIs are tightly integrated with aerOS's underlying continuum ontology, which provides a comprehensive model of the continuum's state. This ontology enables structured representation and real-time monitoring of all relevant entities, including integrated resources, deployed services, and their operational status. Among the various entities defined in the aerOS ontology, the following are the most prominent for effective continuum management and orchestration:

- **aerOS Infrastructure Element (IE):** The fundamental building block of the aerOS ecosystem. An IE represents any physical or virtual resource capable of hosting containerized workloads. aerOS provides dedicated **Low-Level Orchestrators (LLOs)** to manage and abstract this diversity, ensuring consistent control and operation across different types of resources.
- **aerOS Domain:** A complete runtime environment in aerOS, composed of one or more IEs, which share the same instance of core aerOS services. This shared environment ensures cohesive functionality, enabling workload execution and providing essential aerOS integration capabilities—such as network management and resource monitoring—within each domain.
- **aerOS Service:** A user-requested deployment of an application, composed of one or more **service components**. An aerOS service represents the logical grouping of application workloads that need to be deployed and orchestrated across the continuum.
- **aerOS Service Component:** The fundamental deployment unit within a service. Each service component encapsulates all necessary information to fully describe a containerized workload, including networking and computing requirements. These requirements are used by the orchestration system to identify the most suitable IE for hosting the workload, ensuring optimal placement based on available resources and service-level objectives.

While the aerOS ontology also models additional entities—such as **Users**, **Organizations**, **Domain Status**, **Low-Level Orchestrators**, and **Service Component Status**, among others—the entities listed above are the most critical for the integration of aerOS as a continuum runtime in the **ENVELOPE** use case.

The second category consists of orchestration RESTful APIs that handle deployment requests formatted as TOSCA (Topology and Orchestration Specification for Cloud Applications) descriptors. These descriptors define the service to be deployed—potentially composed of multiple (service) components—along with the complete set of user-defined requirements, including allocated resource capabilities and service-level properties. Complementary REST API methods support the full lifecycle management of deployed services, including de-allocation, status retrieval, state transitions, and migration.

[9] https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.08.01_60/gs_cim009v010801p.pdf

The enabler acts as a transformation layer, mapping CAMARA Edge Cloud APIs to the underlying aerOS orchestration and federation APIs. In doing so, it facilitates seamless integration of aerOS capabilities into external platforms and service management systems. The integration of the aerOS enabler along the aerOS continuum is visualized in Figure 11.

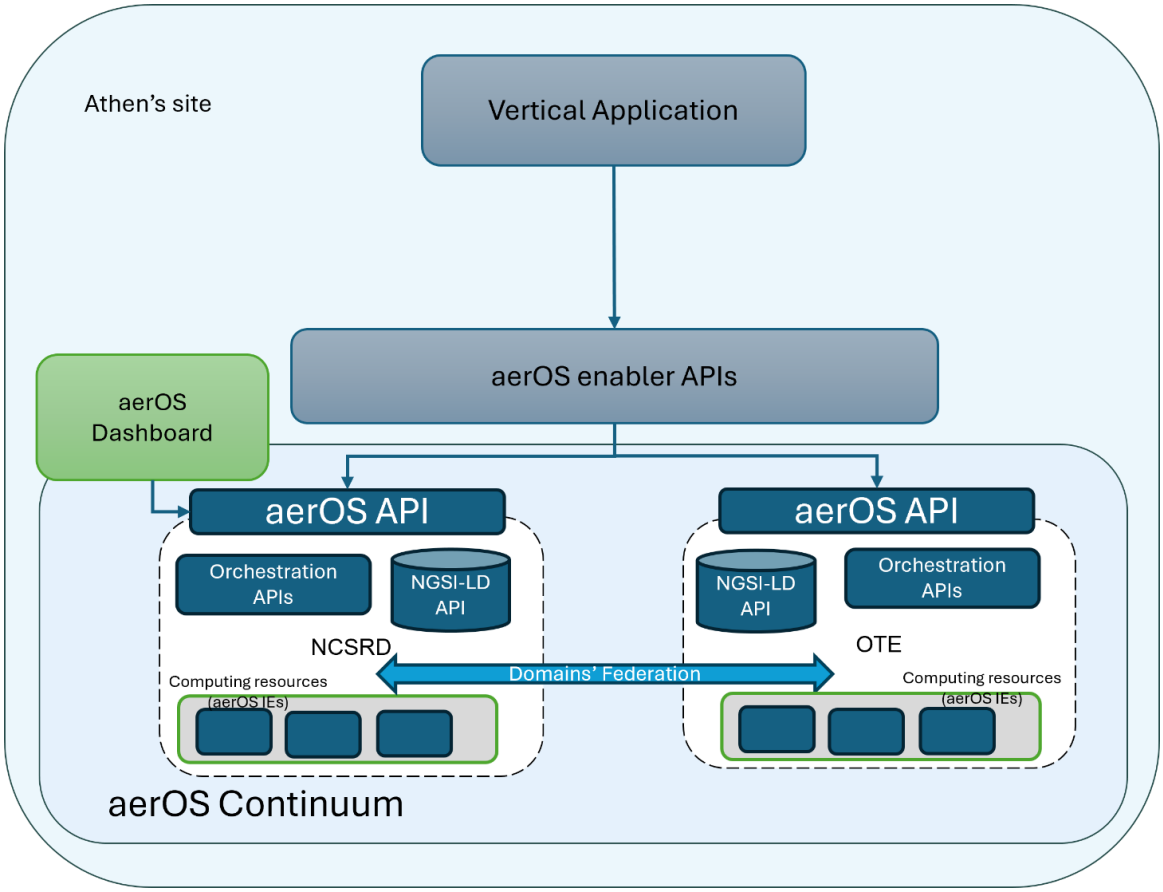

Figure 11 – Athens site continuum

Additionally, Table 2 presents a list of the most relevant aerOS API endpoints, which are accessed indirectly through the CAMARA abstraction provided by the enabler.

Table 2 – 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
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

As part of the Greek site deployment, the aerOS orchestration enabler will support a 5G inter-PLMN mobility scenario, in which a UE moves from one PLMN to another. Given the possibility of this transfer to produce mobility event, this will trigger the need to migrate a running application between two continuum domains. The aerOS enabler has a critical role in enabling this scenario implementation.

The Greek site integrates an aerOS-based continuum composed of two federated domains, each representing a distinct administrative environment. These domains expose their computing and network resources through the aerOS federation layer, enabling seamless interoperability and unified management. Through the enabler, external actors or applications can access real-time resource availability from both domains, using standardized compliant APIs.

To support inter-PLMN handover scenarios, the aerOS enabler provides the functionality to initiate orchestration requests, enabling an application to be dynamically migrated from one domain to the other. aerOS enabler exposes dual orchestration modes, either accept explicit placement instructions, targeting specific infrastructure elements, or interpret application requirements to determine optimal deployment. In the Greek site use case, the former approach is leveraged: applications are explicitly re-deployed onto a selected domain based on the current location of the UE and the target PLMN, ensuring service continuity and context-aware orchestration. Figure 12 illustrates the layered architecture, showing the flow from the application layer through the aerOS enabler to the underlying aerOS continuum.

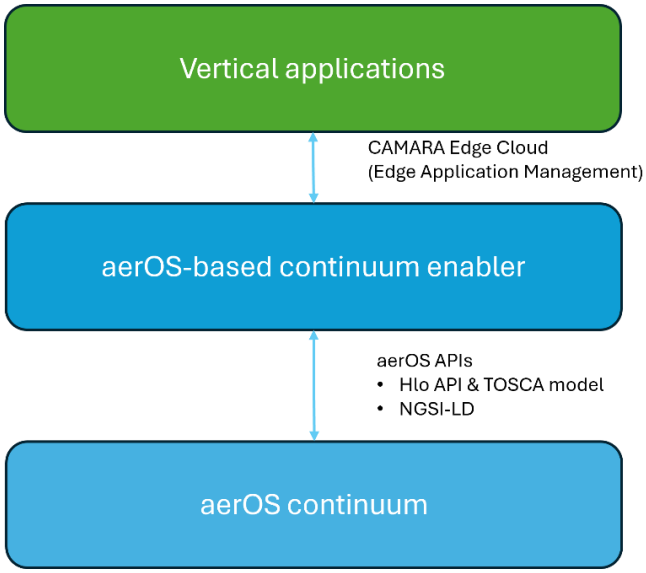

Figure 12 – aerOS enabler for the continuum

All deployments are containerized, enabling seamless portability and runtime consistency across both domains. By abstracting the underlying aerOS capabilities, the enabler ensures that the Greek site can dynamically respond to mobility events by reallocating services based on network context and resource availability, thus supporting reliable and adaptive service delivery across federated edge environments.

3.4.3 Liqo-based[10] Kubernetes multi-cluster enabler

In the Dutch site, the Liqo-based Kubernetes multi-cluster enabler allows vertical services and applications to register, instantiate applications at a particular edge zone. Additionally, it provides means for service migration between edge zones and edge zone discovery based on the edge infrastructure capabilities available in terms of computing resources, e.g., memory, CPU, disk. Figure 13 shows the architecture of the enabler in the context of the underlying edge infrastructure that connects multiple Kubernetes clusters with Liqo. Liqo is an open-source project that enables dynamic and seamless Kubernetes multi-cluster topologies, supporting various infrastructures including on-premise, cloud, and edge. It offers features such as automatic peer-to-peer establishment, seamless workload offloading, transparent network fabric, and native storage fabric. In this architecture, each Kubernetes cluster represents an edge zone where applications can be

[10] https://liqo.io/

instantiated by the edge orchestrator. Applications are registered in a local database that contains all information needed to start/stop a particular application, e.g., with a Helm chart.

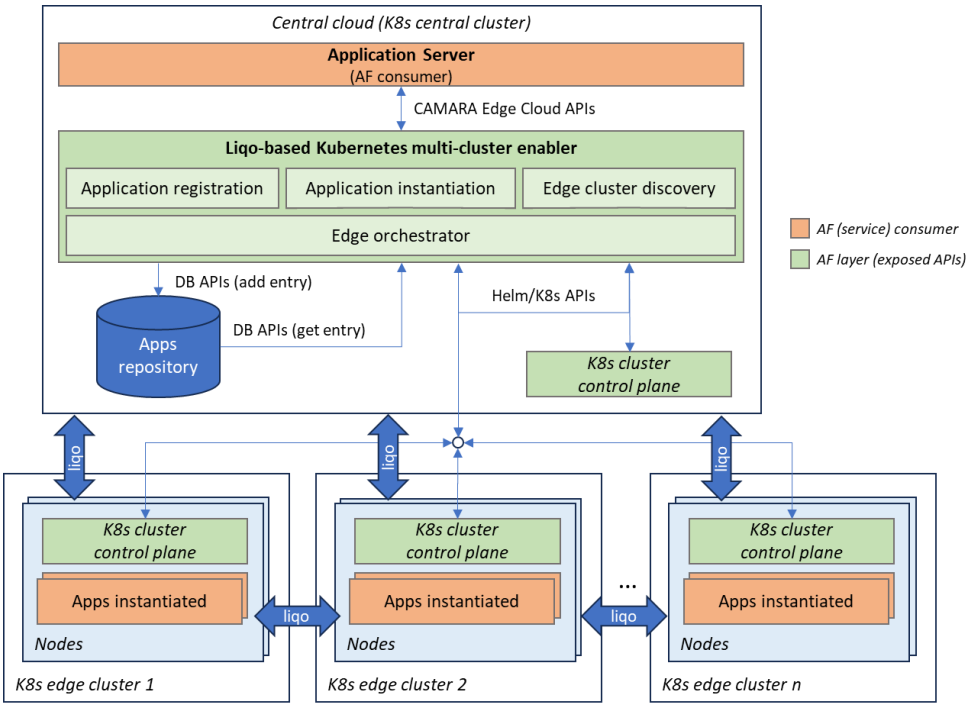

Figure 13 – Architecture overview of the Liqo-based Kubernetes multi-cluster enabler

Figure 14 gives a high-level overview of interfaces used by the Liqo-based Kubernetes multi-cluster enabler. In the northbound, the ENVELOPE APIs are compliant with the CAMARA Edge Cloud (Edge Application Management) APIs[11] which offer registration, instantiation and discovery of applications and edge zones. In the southbound, the enabler makes use of Kubernetes, Helm, and database APIs to interact with the Kubernetes multi-cluster infrastructure. The Liqo-based Kubernetes multi-cluster enabler consists of transformation functions that translate the CAMARA Edge Cloud API requests into the underlying Kubernetes infrastructure APIs. Finally, the enabler monitors the status of the instantiated applications so that vertical applications.

[11] https://camaraproject.org/edge-application-management/

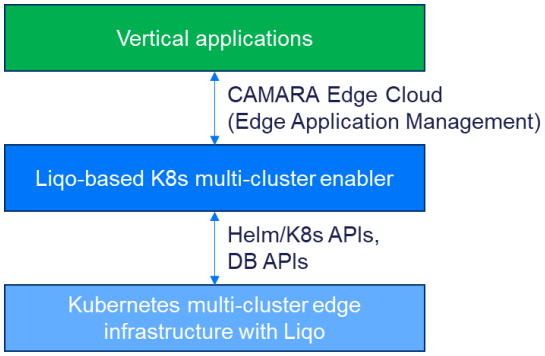

Figure 14 – High-level overview of interfaces used by the Liqo-based K8s multi-cluster enabler

3.5 Enablers for Devices Location

3.5.1 Geofencing enabler

The Geofencing enabler allows vertical applications to subscribe to event-based notifications related to the location updates of specified devices when they enter or leave a certain geographical area. The application consuming the enabler service specifies the geographic area of interest which is defined by a circle with coordinates of the center point and radius. Notifications are posted to the server endpoint defined by the application.

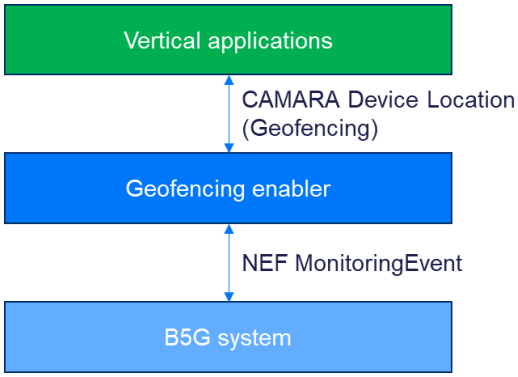

Figure 15 – High-level overview of interfaces used by the Geofencing enabler

Figure 15 gives a high-level overview of interfaces used by the Geofencing enabler. In the northbound, the ENVELOPE APIs are compliant with the CAMARA Device Location Geofencing APIs[12] which offer subscription for geofencing location updates. In the southbound, the geofencing enabler makes use of the NEF *MonitoringEvent* API to subscribe to event monitoring location information. The geofencing enabler consists of transformation functions that translate location updates from the B5G system (e.g., cell id) into geographical area. Finally, it handles the subscription of northbound API consumers and takes care of assessing if a device belonging to a subscription entered or left the pre-defined geofencing geographical area.

[12] https://camaraproject.org/geofencing-subscriptions/

3.5.2 Devices in Area enabler

The Devices in Area enabler allows vertical applications to query which devices are present in a given geographical area or to subscribe to event-based notifications related to the devices that enter or leave the defined geographical area.

The application, upon consuming the APIs exposed by the Devices in Area enabler, can specify the geographical area of interest in a similar way to the geofencing enabler. When the application subscribes to this enabler, in this case as well, notifications are posted to the server endpoint defined by the application.

The main difference with respect to the geofencing enabler is that it is not necessary to know in advance the devices to be followed and to specify them in the query or subscription.

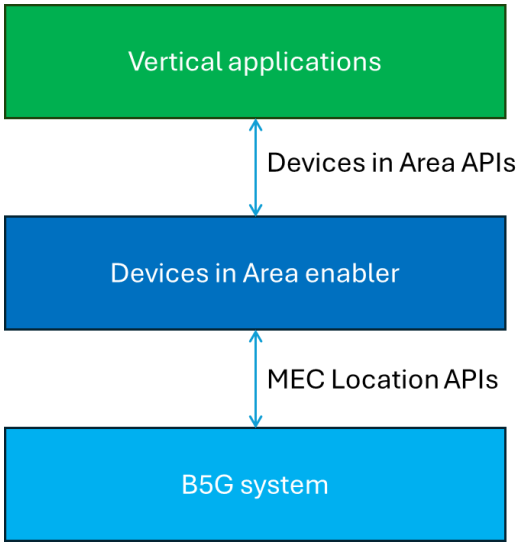

Figure 16 – High-level overview of interfaces used by the Devices in Area enabler

The high-level overview of the interfaces used by Devices in Area Enabler is provided in Figure 16. In the northbound, the Devices in Area APIs follow the same approach of the CAMARA Device Location APIs for what concerns the data model and the subscription procedure. In the southbound, the geofencing enabler exploits the MEC Location APIs to receive information about the position of the devices.

The Devices in Area enabler process the information provided by the MEC Location APIs[13] to understand which devices are in the specified area and to determine which devices enter or leave that area.

3.5.3 Location Reporting enabler

The Location Reporting enabler provides a standardized and secure way to expose real-time and on-demand location updates of UEs to authorized third-party applications within the CAM vertical and beyond.

[13] ETSI GS MEC 013 V3.1.1, “Multi-access Edge Computing (MEC); Location API”

The core functionalities of the enabler include:

1. **Subscription to Location Events:** Enables vertical applications to subscribe to location changes of specific UEs, with options for one-time or continuous location reporting.
2. **Fine-grained Location Data Access:** Supports requests for current or last known location, as well as reporting on cell-level granularity in line with ENVELOPE requirements.
3. **Privacy-aware Reporting:** Integrates with the 5G core's user privacy management to ensure that location data is only disclosed in compliance with user privacy settings.
4. **Open and Secure Access:** Can exploit established network exposure and API frameworks (NEF, CAPIF) for secure, role-based access, supporting both trusted and untrusted domains.

The Location Reporting Enabler exposes an ENVELOPE API that simplifies subscription to and consumption of location update events. The provided endpoints of interest are as follows:

- **/subscribe**
  Allows vertical services to subscribe for location change events for specified UEs. Supports parameters for report type (one-time/continuous), granularity, and duration.
- **/unsubscribe**
  Cancels active subscriptions.
- **/notify**
  Delivers location update notifications to subscribed vertical services.
- **/status**
  Retrieves the current or last known location of a UE on demand.

These APIs are built on top of standardized 3GPP NEF APIs, providing simplified access to lower-level, more technical 5G core network APIs. The underlying NEF functionalities leveraged to provide enabler capabilities are as follows:

- **Subscription Request:** The AF issues a Monitoring Event Subscription to the NEF for LOCATION_REPORTING events. **API:** POST /{scsAsId}/subscriptions (3GPP TS 29.122[14]/29.522[15]).
- **Privacy Related Data:** The NEF queries the UDM (Unified Data Management) for LCS (Location Services) privacy data to verify user consent. **API:** GET /{ueId}/lcs-privacy-data (TS 29.503[16]).
- **Accessing AMF:** NEF requests the UDM for the serving AMF address. **API:** GET /{ueId}/registrations/amf-3gpp-access (TS 29.503).
- **AMF Subscription:** NEF subscribes to the AMF's Event Exposure interface for UE location change events. **API:** POST /subscriptions (TS 29.518[17]).
- **Event Notification:** When a location change is detected, AMF sends a notification to NEF, which forwards it to the AF. **API:** NEF → AF using their agreed callback interface; AMF → NEF via EventExposure Notify (TS 29.518)

[14] 3GPP TS 29.122, "T8 reference point for Northbound APIs"
[15] 3GPP TS 29.522, "5G System; Network Exposure Function Northbound APIs; Stage 3"
[16] 3GPP TS 29.503, "5G System; Unified Data Management Services; Stage 3"
[17] 3GPP TS 29.518, "5G System; Access and Mobility Management Services; Stage 3"

3.6 Enabler QoD

The Quality on Demand (QoD) enabler provides a programmable interface for vertical applications to request prioritized and quality-assured data flows in terms of stable latency (reduced jitter) or enhanced throughput, without requiring detailed knowledge of the underlying network (e.g., 4G/5G systems).

The QoD enabler exposes a set of northbound APIs compliant with the CAMARA Quality-On-Demand specification, allowing applications to create, manage, and delete QoS sessions for specific application data flows. Applications define the data flow by specifying the involved device and application server, including optional port/port-range details. The enabler offers a catalogue of QoS profiles that can be selected based on the application's performance requirements (e.g., low latency, high throughput).

Upon receiving a request, the QoD enabler translates the high-level intent (QoS profile selection) into network-specific instructions by interfacing with 5G Core functions via the standardized Network Exposure Function. These instructions trigger the creation of a corresponding QoS session in the network to prioritize the requested App-Flow.

Notifications related to QoS session lifecycle events (e.g., provisioning, expiration) can optionally be sent to the vertical application's specified callback URL using CloudEvent format, secured via access tokens.

In the **northbound**, the QoD enabler exposes the CAMARA Quality-On-Demand API with the following key endpoints:

- /qos-sessions: Create new QoS sessions based on selected QoS profiles, flow description, and duration.
- /qos-sessions/{sessionId}: Query or delete existing sessions.
- /qos-profiles: Retrieve the list of supported QoS profiles offered by the API provider.

In the **southbound**, the QoD enabler interfaces with the NEF, issuing QoS provisioning requests toward the Policy Control Function (PCF) and other 5GC entities. It may also interact with the User Plane Function (UPF) to ensure the requested data flows are appropriately prioritized.

The core functionalities of the QoD enabler include:

- **QoS Session Management**: Create, query, and delete QoS sessions for specific App-Flows between device and application server.
- **QoS Profile Abstraction**: Abstracts network-level QoS configurations into developer-friendly profiles (e.g., QOS_E).
- **Session Duration and Control**: Enables control over the lifespan of the QoS session with optional early termination.
- **Event Notifications**: Supports subscription to status updates via a secure callback mechanism.
- **Secure and Open Access**: Utilizes OAuth 2.0 for authentication and may integrate with CAPIF for access control and API discovery.

In the implementation of the QoD enabler for the Italian trial site, the QoD enabler creates the new QoS session in a new dedicated slice. The functionality of dynamic slicing is then also included in the Italian trial site's QoD enabler.

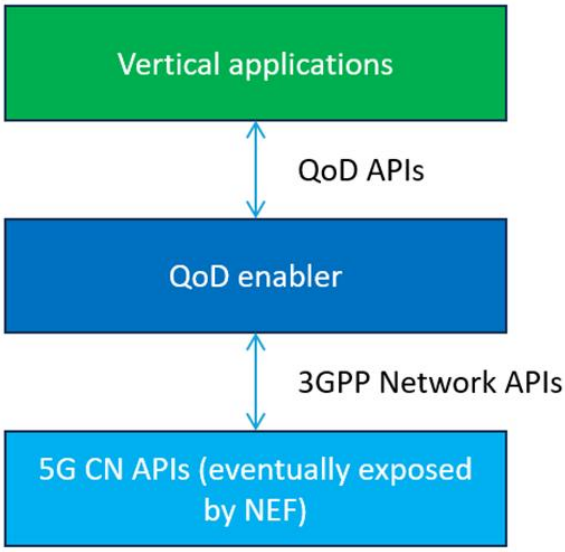

Figure 17 – High-level overview of interfaces used by the QoD enabler

The QoD enabler simplifies the interaction with complex 5G network functions, offering an accessible and standardized interface (depicted in Figure 17) for service developers to program network behaviour in support of high-performance applications.

4 Implementation of the EaaS module

This section reports the implementation activities of the EaaS module. First, in Section 4.1 the functionalities of the EaaS module are introduced, and the sequence diagrams of the operations implementing these functionalities are described in Section 4.2. The sections following Section 4.2 report the implementation details of the components constituting the EaaS module. The internal architecture of each component, its interfaces and dependencies are described.

4.1 EaaS functionalities

The functionalities offered by the EaaS module are listed in Table 3. The first column contains the name of the functionality, whose description is provided in the second column. The EaaS-related requirements, which have been specified in the ENVELOPE deliverable D2.2, lead to the identification of these functionalities. In the third column, the requirements relevant for the related functionality is reported.

Table 3 – EaaS high-level functionalities

Name	Description	REQ-F-Task3.1-*
Login	Authentication	EaaS-1
Trial Site's Information	Specific information about the Trail Site on which the EaaS module is deployed	EaaS-16, EaaS-23
Application Onboarding	The experimenter uploads an application	EaaS-6
New Experiment	Create a new experiment instance	EaaS-14, EaaS-18, EaaS-19
Start Experiment	Start an experiment instance	EaaS-14, EaaS-17, EaaS-20
Stop Experiment	Stop an experiment instance	EaaS-14, EaaS-21
Monitoring Experiment	Monitor the experiment	EaaS-22
Application Onboarded	List the applications available	EaaS-7, EaaS-10
Update application	Update an experimenter's application	EaaS-8, EaaS-10
Delete application	Delete an experimenter's application	EaaS-9, EaaS-10
Experiment Onboarding	The experimenter uploads its experiment descriptor	EaaS-11, EaaS-15

4.2 Sequence diagrams of the operations

The sequence diagrams of the operations that implement the main EaaS functionalities are introduced in this section.

4.2.1 Application Onboarding

The application onboarding operation is depicted in Figure 18. The onboarding process starts when a single application package is generated (step 2) with the CREATED onboarding state, and it ends when its state transitions to ONBOARDED after the user uploads the package contents (step 5) and the Repository successfully processes them.

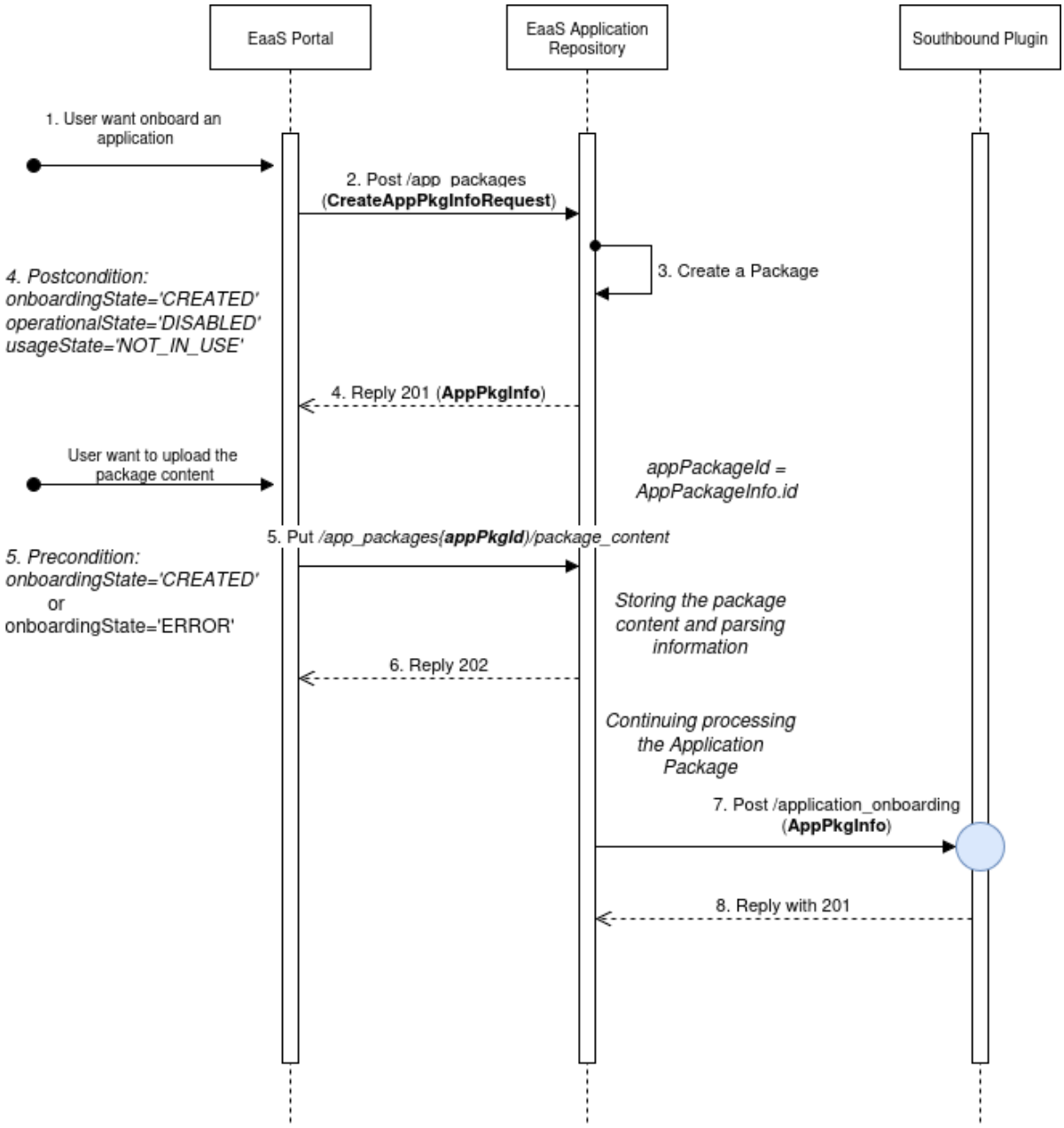

Figure 18 – Application Onboarding sequence diagram

4.2.2 Experiment Descriptor Onboarding

The steps for onboarding an experiment descriptor are illustrated in Figure 19. The experiment descriptor onboarding operation starts when an experiment descriptor resource is generated (step 2) with the CREATED onboarding state, and it ends when its state transitions to ONBOARDED after the user uploads the experiment descriptor (step 5) and the Descriptor Manager successfully processes them.

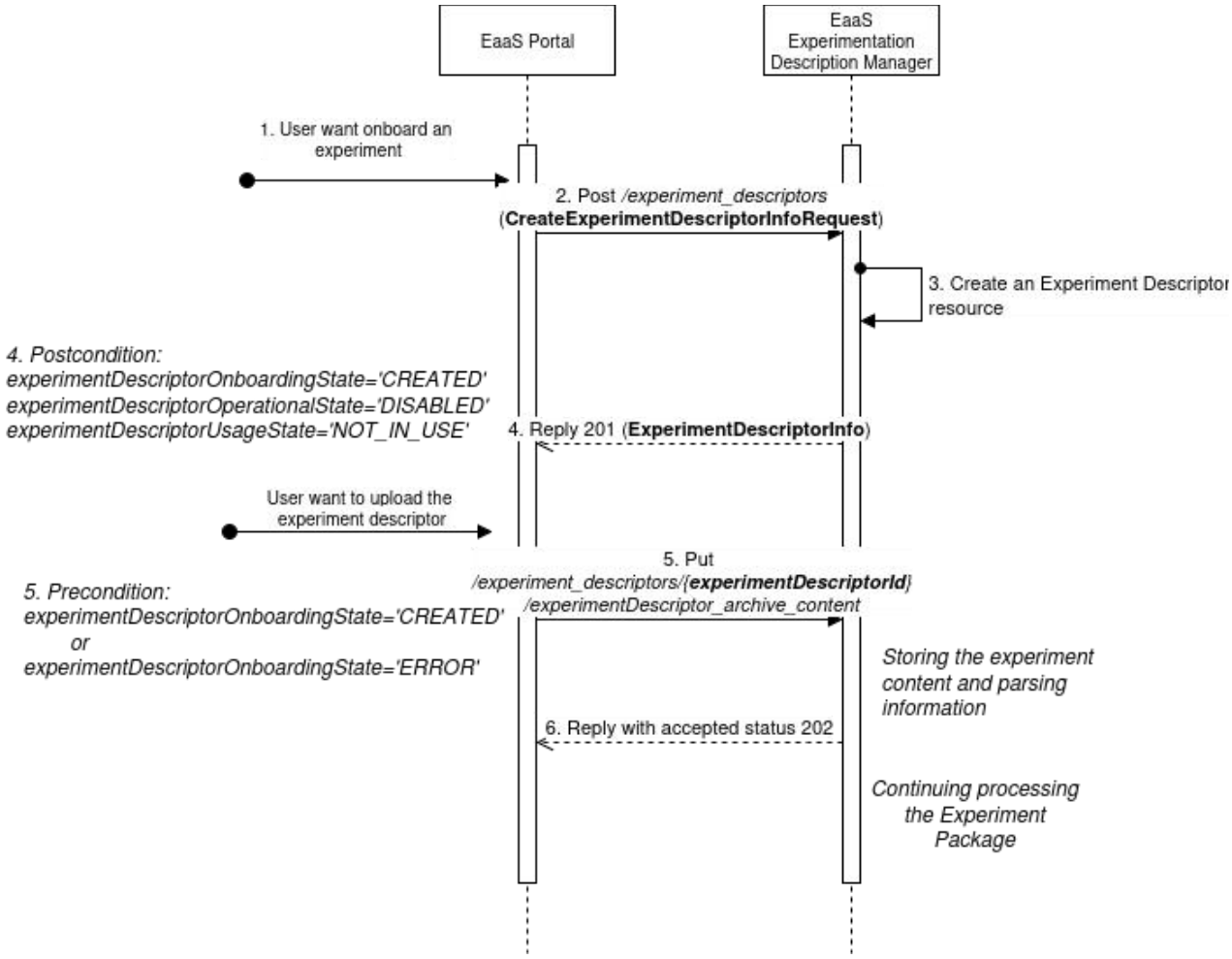

Figure 19 – Experiment Onboarding sequence diagram

4.2.3 Create Experiment

The user creates an experiment based on an experiment descriptor, that was previously onboarded, following the steps shown in Figure 20.

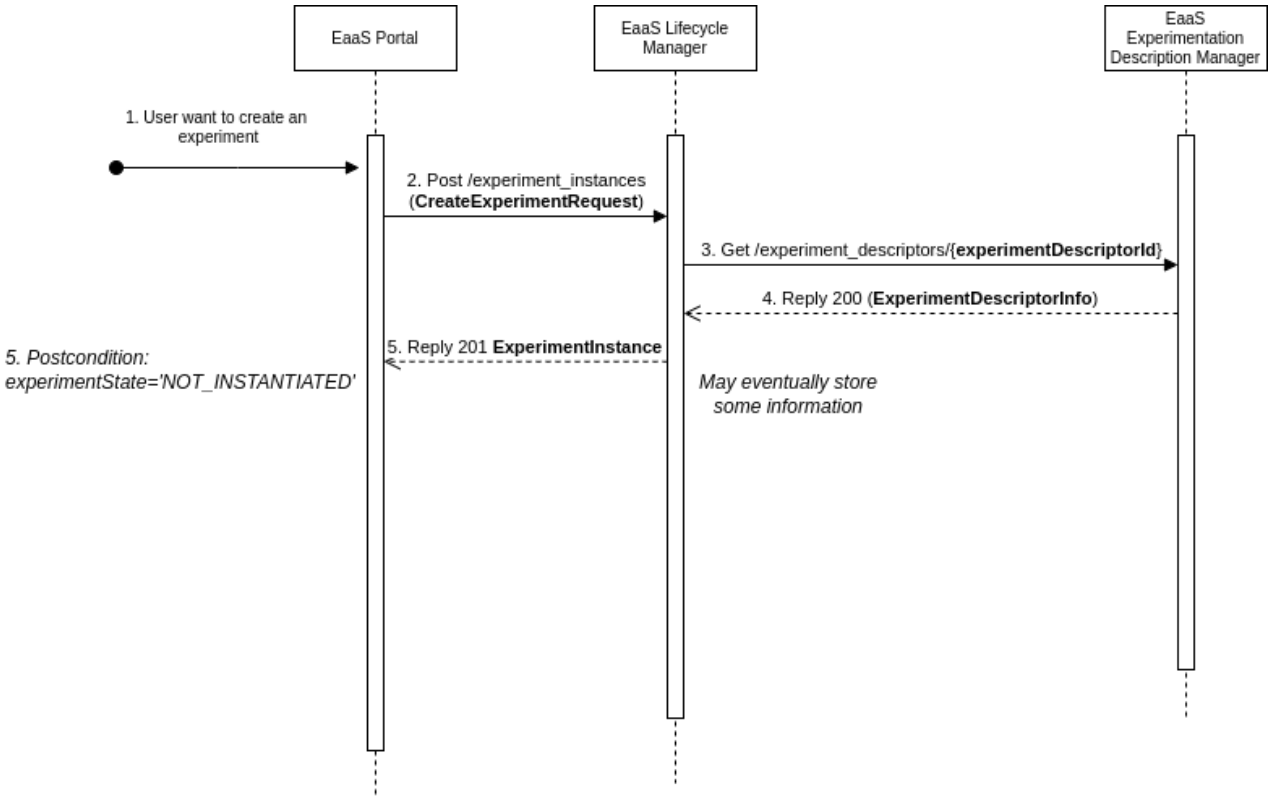

Figure 20 – Create experiment sequence diagram

4.2.4 Start Experiment

In Figure 21, the sequence of operations for starting an experiment is introduced. When the user triggers the start of an experiment, an instance of the experiment is started, and each application referenced in the descriptor is requested to be instantiated as part of the experiment execution.

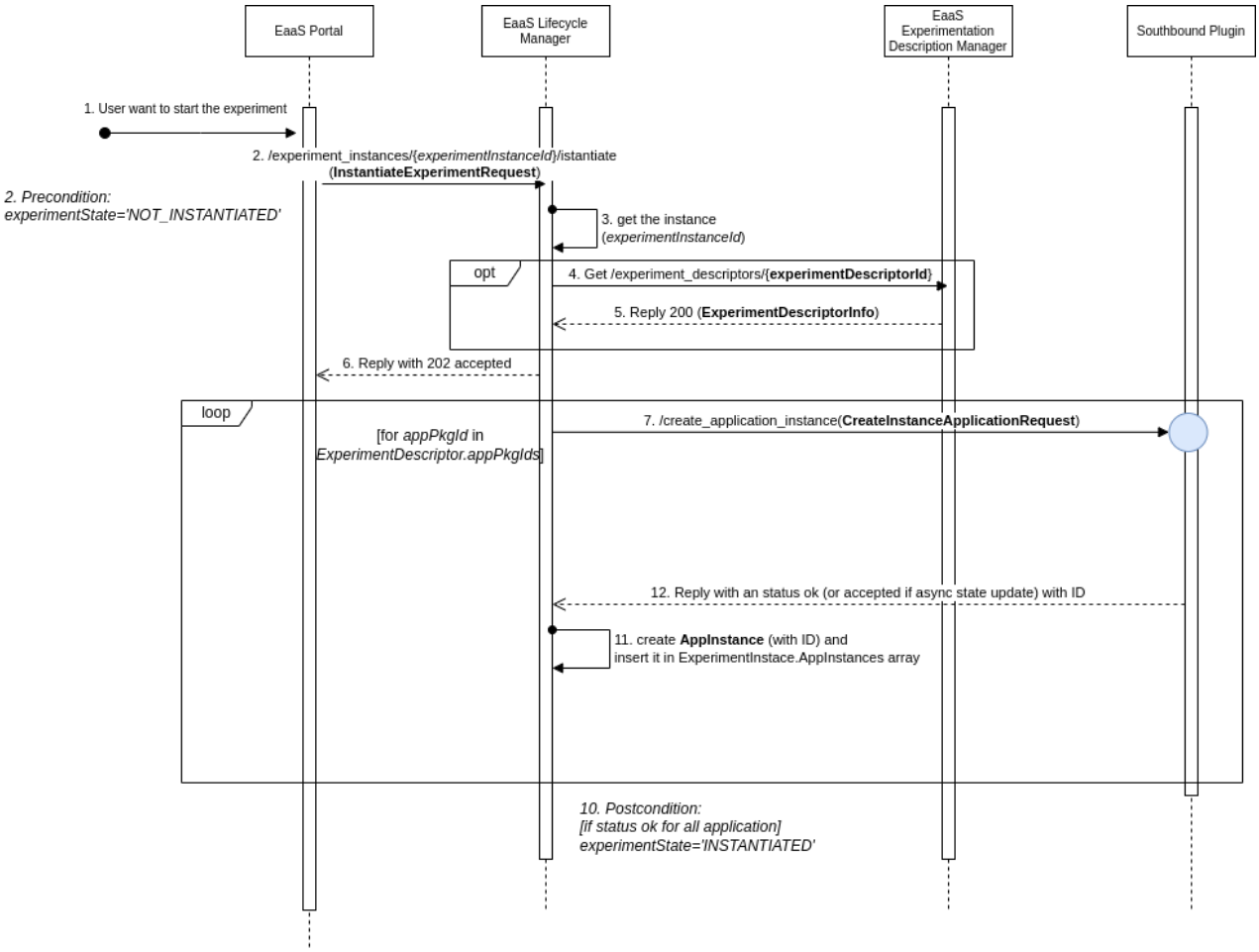

Figure 21 – Start Experiment sequence diagram

4.2.5 Stop Experiment

The steps for stopping an experiment are reported in Figure 22. The selected instance of the experiment is stopped, and each application referenced in the descriptor is requested to stop as part of the experiment termination process.

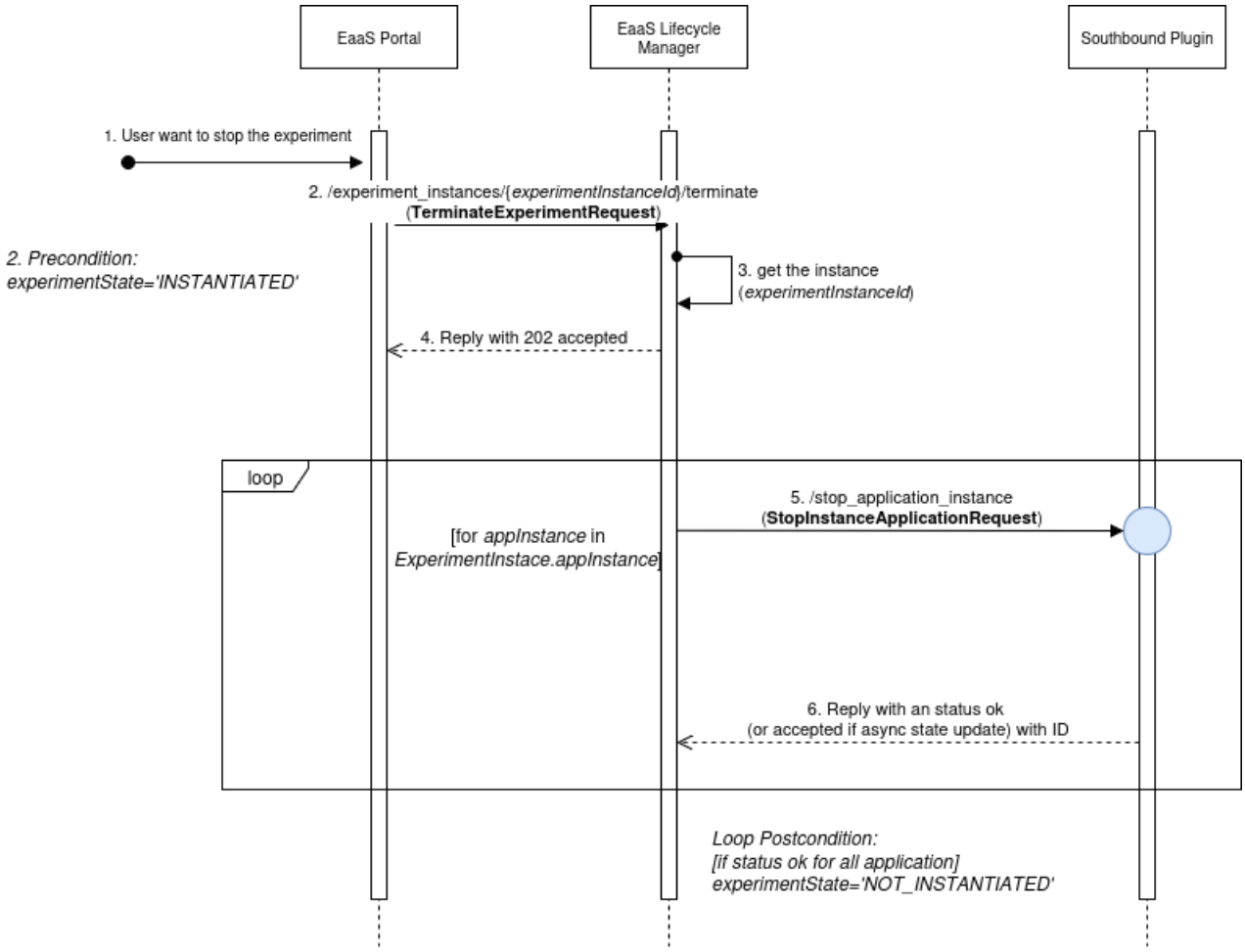

Figure 22 – Stop Experiment sequence diagram

4.3 ENVELOPE Portal

4.3.1 Overview

The ENVELOPE Portal, as the front-end of the EaaS module system, aims to provide a single point of access for the experimenter towards the ENVELOPE Platform that is deployed on each trial site.

After authenticating as a user of the platform, the experimenter will be given access to a web application which offers graphical user interfaces (UIs) to perform the following actions:

- The onboarding of applications.
- The onboarding, orchestration, and monitoring of experiments.
- The lookup of trial sites capabilities.

In the EaaS User Manual[18], that is made available as knowledge base to the participants of the ENVELOPE Open Calls, a detailed, user-friendly explanation for the interaction with the Portal is provided, along with an exhaustive set of screenshots of the webpages. In the present document, focus will shift towards the architecture, interface, and dependencies of the application. Nevertheless, to provide the reader with a glimpse of the look and feel of the user interface presented to the experimenter upon entering the Portal, this section includes some illustrative images.

Figure 23 showcases the *Application Component* webpage. The upper part of the interface features a file upload area dedicated to application onboarding, while the lower part displays a list of all previously created application packages stored in the *Application* Repository whose details are introduced in Section 4.4.

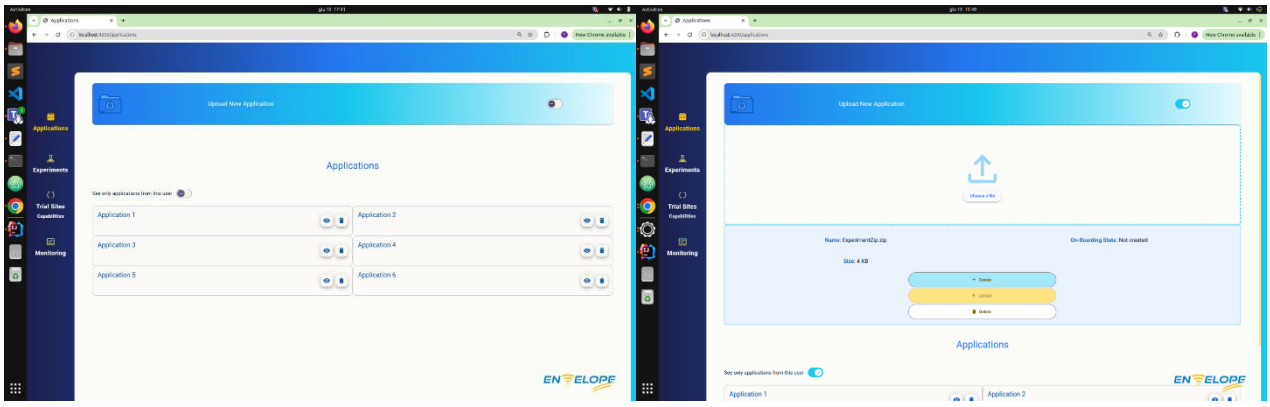

Figure 23 – Examples of Application Component webpage

Figure 24 showcases the *Experimentation Component* webpage. The upper part of the interface features a file upload area dedicated to experiments onboarding, while the central part displays a list of previously created experiment descriptors stored in the *Experiment Descriptor Manager* (Section 4.5). The lower area is dedicated to lifecycle management of onboarded descriptors and the interaction with the *Experiment Lifecycle Manager* (Section 4.6).

[18] EaaS User Manual, https://envelope-project.eu/wp-content/uploads/2025/06/EaaS-User-manual-OC-1-20250602.pdf

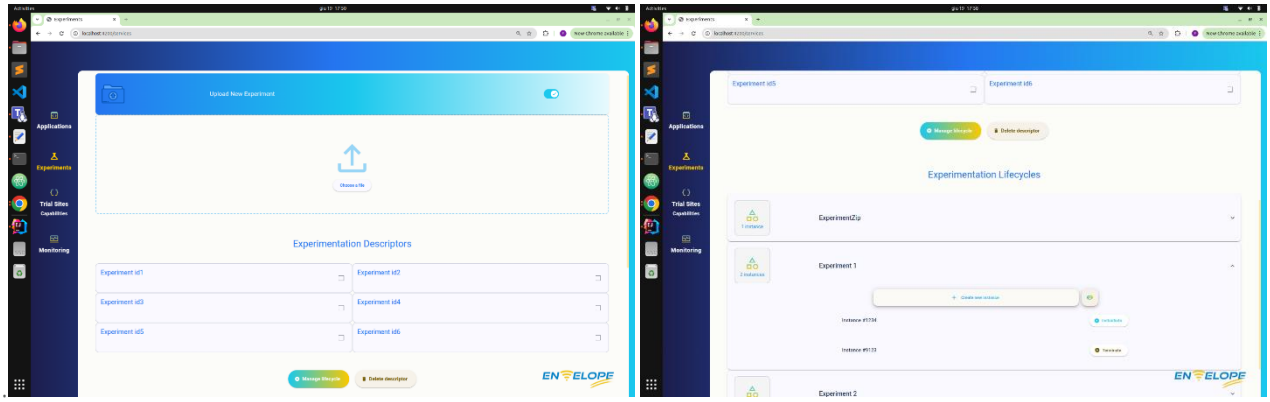

Figure 24 – Examples of Experimentation Component webpage

4.3.2 Architecture & Design

The framework chosen for the development of the portal is Angular[19] version 19 along with Typescript[20] language 5.4.x, for several reasons:

- The availability of many User Interface (UI) components - e.g. forms, buttons, dialogs - that are pre-made and ready to use. Such components are compliant with the *Material 3* gn system[21].
- The extensive community support.
- The easy backward compatibility for future developments.
- The modular architecture feasible for the requirements.

Angular architectural elements (Components, Services, Dependency Injection) have been applied to the ENVELOPE Portal's project structure, enabling the compliance with the *Model-View-ViewModel*[22] (MVVM) design pattern, a software engineering methodology typically used for front-end based applications.

4.3.2.1 Components

An Angular **Component** is a self-contained software unit that manages an independent UI unit (like a full web page, or an independent section of it) and its related business logic.

A Component can encapsulate other Components as children of itself - e.g. a webpage Component can have some framing (top bar, bottom bar, lateral bar), a pop-up window, various sub-sections, etc, and these can be Components as well.

The use of Component presents the following advantages:

- Separation of Concerns: required features for the ENVELOPE Portal involving different UI components and/or different business logic are mapped onto different Components.

---

19 https://angular.dev/
20 https://www.typescriptlang.org/
21 https://m3.material.io/
22 https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

- Scalability and reusability: Components are modular and can be reused across the app.
- Testability: it's easier to test a modularized app.
- Encapsulation: each component manages its own business logic state and communicates to other components only through the explicit parenting of Components.

ENVELOPE Portal's project structure is divided into the following components:

- Application Component, which manages the onboarding and deleting of applications.
- Experimentation Component, for experiments onboarding, deletions, and for managing descriptors and lifecycles.
- A Common Component which is a child component for both the Application and Experimentation, which handles the file uploading. This component is the delegate for the management of the zip local file loaded by the user for the onboarding, whether it be for application or for experimentation.
- Trial Sites Capabilities Component to inspect the trial sites capabilities.
- Monitoring Component to visualize Grafana dashboards for the experiments.

Each component, except for the Common Component, acts as a routing endpoint for the application, or, in simpler words, constitutes a section of the website.

4.3.2.2 Services

While Components hold the view, on the other hand, Services process the data.

An Angular Service is a typescript class that manages data model and business logic for the web application. In particular:

- It creates the network requests to the back-end service and handles its incoming responses (REST APIs services).
- It handles the data model i.e. the structures and types of the data manipulated by the application (data model).
- While getting network responses from the backend, it transforms Json payloads into the data model (deserialization).
- While creating network requests to the backend, it transforms instances of data model into Json payloads (serialization).

Note that Components and Services can both handle business logic, for different purposes: Components for transforming user input into more feasible view data structures, Services for serialization, deserialization, or other data model's related tasks.

Components delegate to Services the interaction with the backend, through REST API calls. A Component holds a reference to a Service through *dependency injection*, which creates a *singleton* instance (i.e., a singleton is a one-time instantiated, shared object) of the Service and provides that to any Component that needs it.

In the ENVELOPE Portal, following services have been implemented:

- EaaS Application Repository Service, which manages the http calls to the Application Repository Manager back-end. This service is injected into and used by the Application Component.
- EaaS Experimentations Service, which manages the http calls for the Experiment Descriptor Manager and Experiment Lifecycle Manager back-end. This service is injected into and used by the Experimentation Component.
- Computing Nodes and Edge Zones services to read the edge cloud zones APIs and to provide information to the Trial Sites Capabilities Component, which is injected into.

4.3.2.3 Model-View-ViewModel

MVVM (Model-View-ViewModel) is a UI architectural pattern that belongs to the broader family of Model-View separation patterns, together with MVC (Model-View-Controller) and MVP (Model-View-Presenter). MVVM shares with MVC similar goals and architectural principles focused on modularity and testability.

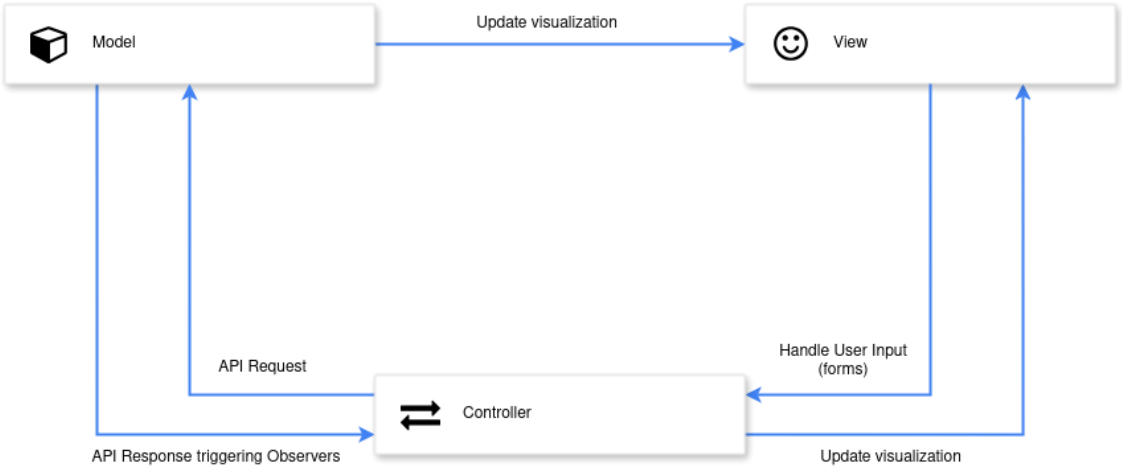

Figure 25 – Model-View-Controller schema

Model-View-Controller (MVC) is a family of design patterns used to organize software applications featured by a user interface. Its main task is to separate different application responsibilities into different modules:

- Model: manages the data, logic, and rules (business logic) of the application. It responds to requests for information and updates the data when necessary. It usually interacts with some remote back-end services.
- View: for the user interface. It displays the data from the model to the user and sends user commands (like clicks or input) to the controller.
- Controller: handles the input from the user, processes it (possibly changing the model), sends the update to the model, and eventually updates the view if model changes.

MVC makes code easier to manage, update, and test, as each part has a clear and specific role (separation of concerns).

Model-View-Presenter (MVP) is a restricted variant of MVC not allowing the Model to directly update the view, as every view update should pass through the Controller (called now Presenter).

MVVM is a further evolution of MVP, as it doesn't let for manual updates of the View by the Controller (called now ViewModel), neither for the Controller to hold any reference of the View object. Instead, View automatically updates when the ViewModel data changes, and vice versa, through a mechanism called *view binding*. Under the hood, view binding is an Observer/Observable pattern, where the ViewModel listens for changes of the view (user input) and vice versa (model changes).

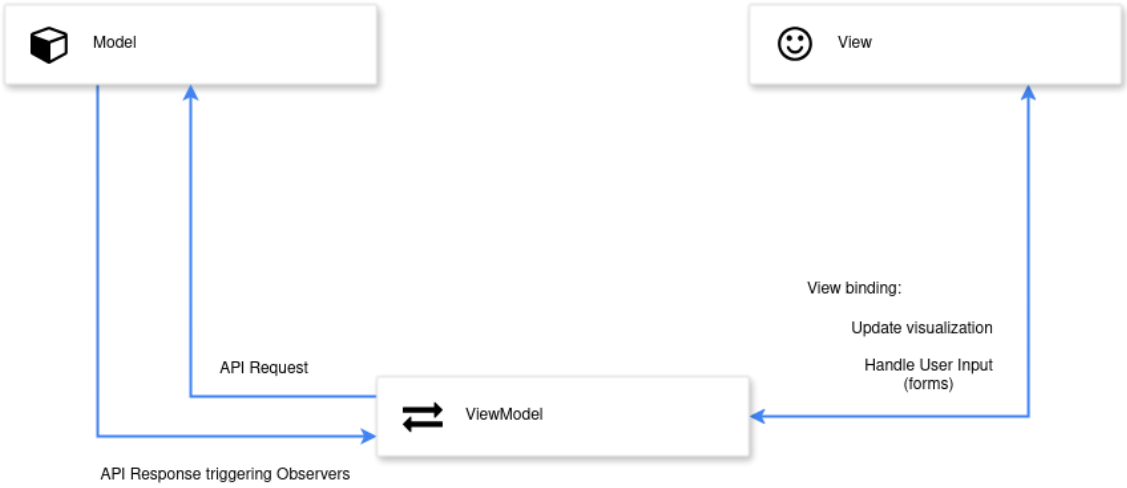

Figure 26 – Model-View-ViewModel schema

In more classic web development frameworks, manual updates of the view (MVC, MVP) are implemented using DOM (Document Object Model)/HTML explicit pointers. This old method is mostly deprecated in current Angular, where a lot of tools are made available to implement the MVVM view bindings: reactive forms (ngModel), directives, signals. Component-Service Angular architecture can fit MVC/MVVM pattern.

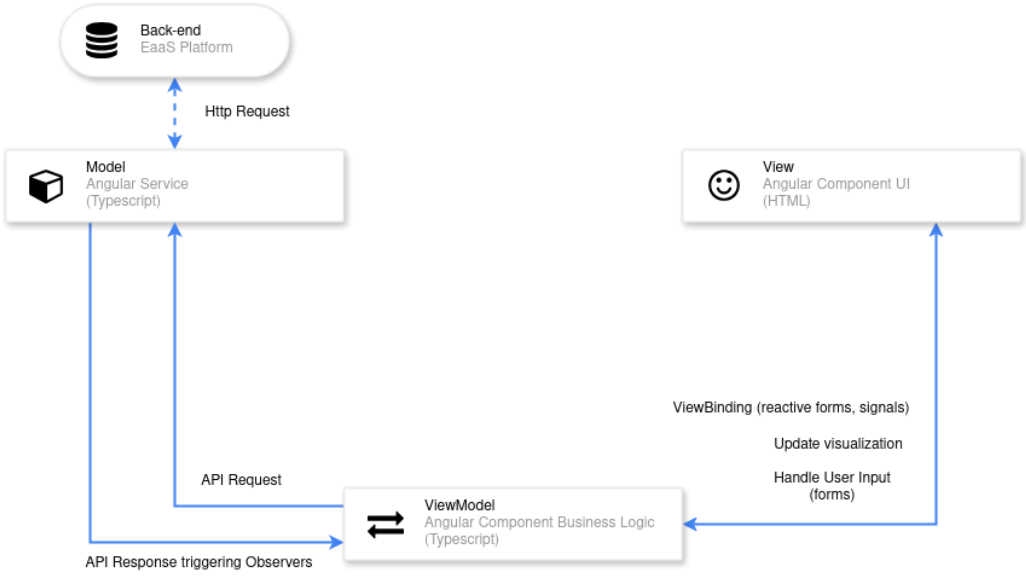

Figure 27 – Model-View-ViewModel with Angular components and services

4.3.2.4 MVVM in the ENVELOPE Portal

The ENVELOPE Portal has been designed to accomplish both MVVM and Angular Components/Services architecture.

The Application Component interacts with the EaaS Application Repository Service to onboard an application zip file to the Application Repository Platform.

In a similar way, the Experimentation Component interacts with the EaaS Experimentations Descriptor Service, to onboard the local zip file of an Experiment to the Experiment Descriptor Manager.

Experimentation Component also interacts with another Service, the EaaS Experimentations Lifecycle Service, to handle the lifecycle of an onboarded Experiment descriptor, i.e. to create an instance, to start and to stop an experiment instance, or to delete a descriptor.

The same structure holds for Trial Sites Component and the services it will rely upon.

In Section 4.3.2.5, the process of a user interacting with the ENVELOPE Portal for onboarding an application local zip file is analysed in its deep behaviour and underlying elements, which are Services, Components, API calls and http calls.

The other use cases (Experiments, Trial Sites) are analogous in structure.

4.3.2.5 Application onboarding case

The steps for onboarding an application, shown in Figure 28, are the following:

1. The first action from the user is to upload a local zip file from their local filesystem to the front-end.
2. Once done, the View of the Component prompts an action to the user with a “Create” button. When the user clicks, the first phase of the onboarding process starts. The outcome of this

first phase is that an application package will be created in the Application Repository (back-end), with onboarding state CREATED and empty content.

3. The ViewModel of the Component listens for button click with a callback function. Once clicked, the callback function invokes the function of the Service responsible for the API call for the package creation. The ViewModel also instantiates an Observer (a class called Subscriber in Typescript) that will trigger once the Service communicates that the operation is completed.
4. The Service starts a POST http request to the back-end with the data provided by the ViewModel. It also instantiates a Subscriber to the back-end response.
5. Once the back-end response comes back, the Service Subscriber will trigger, then also the ViewModel Subscriber will trigger (as technically the last one is chained to the first one).
6. Some variables in the ViewModel will be changed to indicate that the operation is completed. Then, some View component, bound to those variables, will be automatically turned on to display the user the success or failure of the operation.
7. If the creation has been successful, ViewModel will call the Service again, this time to invoke the API to get the list of created applications in the Application Repository back-end, which now will contain the newly created one. ViewModel will subscribe to the Service for its response, which will consist of a list of created application packages.
8. The service will start a GET http request to the back end and subscribe to the response. When results are available, subscriptions will trigger again in the ViewModel, which will re-update the View with the new list of created application packages.

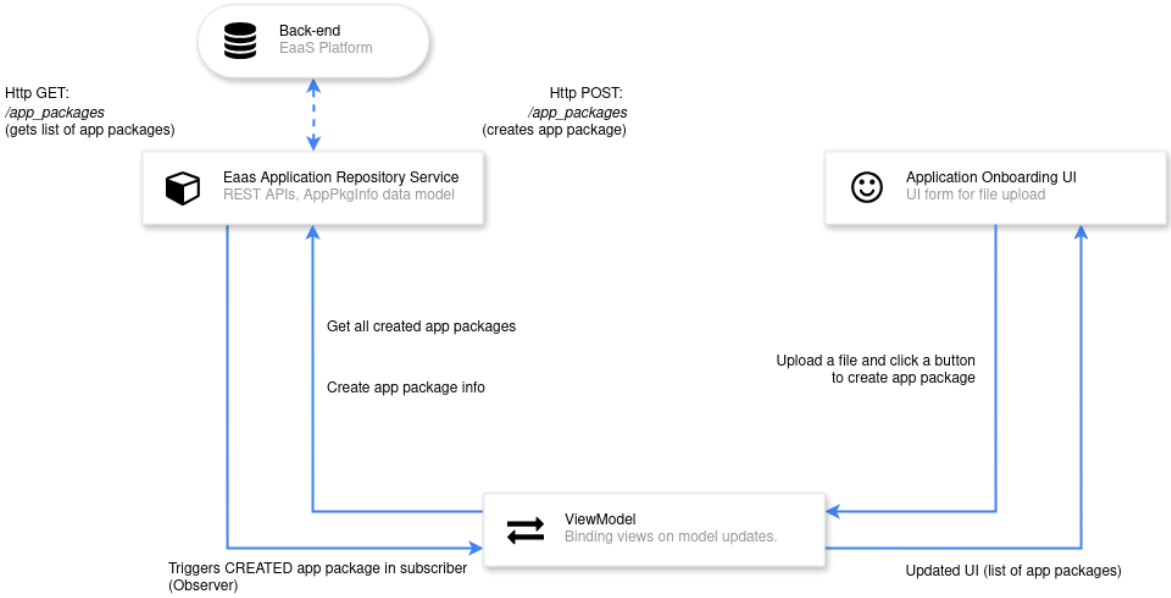

Figure 28 – Onboarding an application – first phase: create app package

9. The second phase (Figure 29) of on-boarding can now start. A button in the View will become active to prompt the user to upload the local zip file into the Application Repository back-end.
10. When the new button is clicked, ViewModel triggers a callback which invokes the Service API responsible for file upload, passing the zip file raw bytes as parameter and subscribing to Service results.

11. The Service API calls the http PUT method, passing the zip file bytes as message payload and subscribes to the back-end response.
12. As soon as back-end responds, Subscriptions will retrigger and entail a recompositing of the View, which will show the updated results (the completion of the onboarding process).

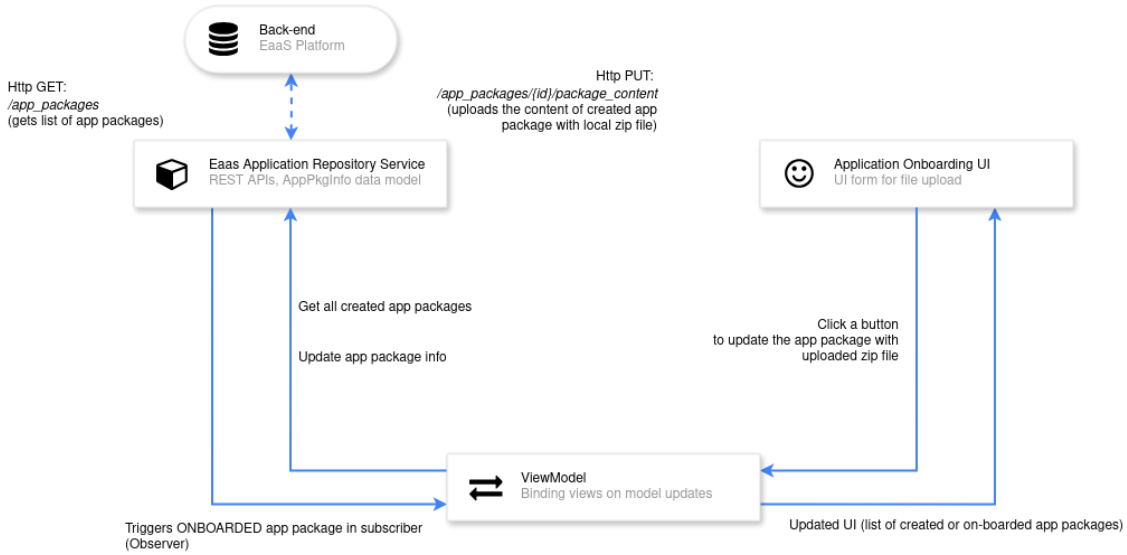

Figure 29 – Onboarding an application – second phase: upload app package

4.3.3 Interfaces

The ENVELOPE Portal offers graphical user interfaces to access the service endpoints exposed by the EaaS Platform that are the most relevant for the user, between those described in 5.4.3, 5.5.3, 5.6.3.

- Application Repository Interface: Enables experimenters to manage application onboarding, including uploading, updating, and deleting applications. This facilitates the preparation of vertical services to be deployed during experimentation.
- Experimentation Descriptor and Lifecycle Interface: Provides endpoints to initiate and terminate experiments. It ensures that experiments are launched correctly and terminated gracefully, in coordination with other portal components.

When users want to access any API endpoint, in read or write mode, they prompt an action to the UI, then this action propagates to the deep data layers of the application, which will finally trigger an http request. The complete process is shown in 5.3.2.2.4 and 5.3.2.2.5.

4.3.3.1 Application Repository Interface

- Get /app_packages: this call is performed to retrieve all the created app packages and display them to the bottom of the page, when the user lands on the Application Component route (i.e. the Applications webpage), or when the user modifies the list of created app packages, i.e. when they create or upload an application package.

- Post /app_packages: this is called when the user clicks the button “Create” after they’ve locally uploaded the zip application file. This will create an empty appPkgInfo with an appPkgId.
- Put /app_packages/{appPkgId}/package_content: this is called to finish the onboarding process by uploading the raw bytes of the local zip file to the Application Repository.

4.3.3.2 Experimentation Descriptor Interface

- Get /experiment_descriptors: this call is performed to retrieve all the created experiment descriptor ids and display them to the central section of the page, when the user lands on the Experimentation Component route (i.e. the Experimentations webpage), or when the user modifies the list of created experimentation descriptors, i.e. when they create or upload an experimentation package.
- Post /experiment_descriptors: this is called when the user clicks the button “Create” after they’ve locally uploaded the zip experiment file. This will create an empty experimentDescriptorInfo with an experimentDescriptorId.
- Put /experiment_descriptors/{experimentDescriptorId}/experimentDescriptor_archive_content: this is called to finish the onboarding process, by uploading the raw bytes of the local zip file to the archive of the descriptor with specified experimentDescriptorId in the Experiment Descriptor Manager.

4.3.3.3 Experimentation Lifecycle Interface

- Post /experiment_instances: The Post method creates a new Experiment instance resource. This call is performed when the user inspects the lifecycle of a selected experimentDescriptorId, then clicks the button “Instantiate” to create a new instance of the experiment descriptor having such id.
- Post /experiment_instances/{experimentInstanceId}/instantiate. The Post method starts an existing Experiment instance resource. This call is performed when the user inspects the lifecycle of a selected experimentDescriptorId and the list of existing Experiment instances for that descriptor that are not started yet. Then clicks the “Start” button aside of one of those instances to make it start.
- Post /experiment_instances/{experimentInstanceId}/terminate. The Post method terminates an existing Experiment instance resource. This call is performed when the user inspects the lifecycle of a selected experimentDescriptorId and the list of existing Experiment instances for that descriptor that are already started. Then clicks the “Stop” button aside of one of those instances to make it terminate.

4.3.4 Dependencies

Angular version 19 has been used for the project. Angular material version 19 provides for all the UI components present in the web application: buttons, forms, widgets, pop-ups, dialogs.

Google Material Symbols have been used for material icons. Colour palette used in webpages is compliant with the document “ENVELOPE Visual Identity”.

Rxjs library in TypeScript has been used for reactive programming such as the implementation of Observer patterns (Subscription in TypeScript) in Angular Services. See 5.3.2. for more about this architectural choice.

4.4 Application repository

4.4.1 Overview

The applications are used to compose a vertical service (i.e., a chained set of applications) to experiment in the ENVELOPE Platform. The Application repository stores the artifacts needed to deploy the application. The Management and Orchestration Layer retrieves these artefacts to deploy the vertical service.

Three main concepts characterize an Application:

- The Application Package represents an atomic component, i.e., the smallest object able to be onboarded. It contains all the information related to a single application in terms of artifacts and descriptor.
- Application Descriptor: A file which describes the application in terms of deployment needs
- Artifact: arbitrary elements that compose an Application (i.e. helm chart, licences, scripts file, ...)

Following the functional requirement described in the Deliverable D2.2, the application repository must provide the functionalities summarized in Table 4.

Table 4 – Functionalities mapped to requirements

Functionality	Requirement
The experimenter shall have the possibility to load applications in the Application repository	REQ-F-Task3.1-Eaas-6
The experimenter shall have the possibility to check which applications have been loaded	REQ-F-Task3.1-Eaas-7
The experimenter shall have the possibility to update applications in the Application repository	REQ-F-Task3.1-Eaas-8
The experimenter shall have the possibility to delete the applications that have been loaded	REQ-F-Task3.1-Eaas-9
The experimenter shall have visible only its own applications if these have not set as public during the loading phase	REQ-F-Task3.1-Eaas-10

4.4.2 Architecture & Design

The primary function of the Application Repository is to manage the user's packages, both the artefacts and the packages themselves, in an as-a-service manner. It should not only handle the onboarding of user packages but also address non-functional requirements such as reliability and scalability. For doing so the application service has been designed to address several challenges following the microarchitecture design principle emphasizing modularity, autonomy, and resilience.

4.4.2.1 Architecture

The architecture of this module is depicted in Figure 30 and it is composed of the following technologies:

- Application Repository: this a microservice that provides all the functionality through a synchronous protocol. It also manages the non-functional requirements with the help of a broker using an asynchronous protocol.
- MongoDB[23]: NoSQL database that store the application repository's state
- Redis[24]: cache layer between the application repository and the MongoDB
- RabbitMQ[25]: reliable and mature messaging and streaming broker.

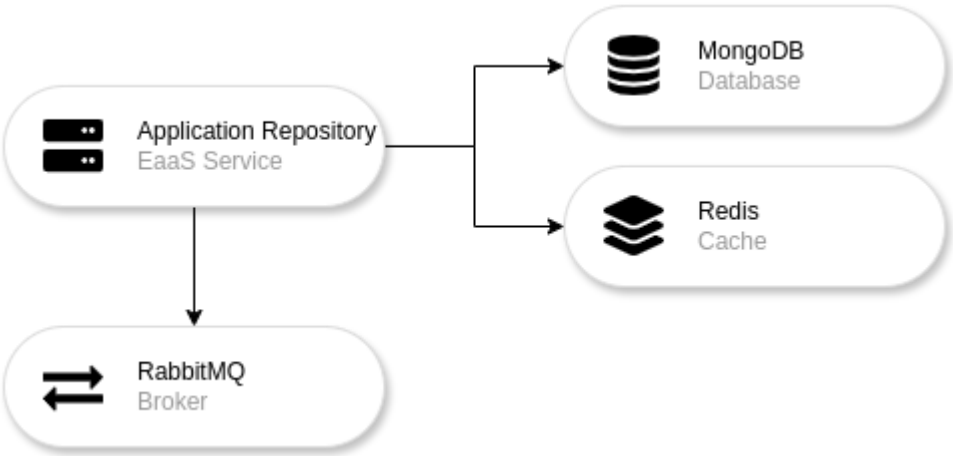

Figure 30 – Application Repository Module Architecture

4.4.2.2 Software module

The Application Repository (Figure 31) is composed of 4 main modules:

- Web: this module exposes APIs and manage the http server; it will delegate the logic behind each API calling methods exposed by the repository module
- Repository: It is responsible for the logical business, managing descriptors, validate them and managing operations, it interacts with the broker and the persistence module.

[23] https://www.mongodb.com/
[24] https://redis.io/
[25] https://www.rabbitmq.com/

- Persistence: is responsible for storing the Application Repository state through two sub module, one which use the MongoDB driver, providing database functionality and the second one uses the Redis driver, providing caching functionality. It also interacts with the broker to push updates about the application package model.
- Broker: this module exposes two independent set of function, the first set is called by the Repository, where the repository manages the usage of an application package and the read of the application package's state update when requested by the portals through the AMQP protocol. The second set of the functions is used by the Persistence where the persistence module pushes (acting as producer) each application package update in a Stream.

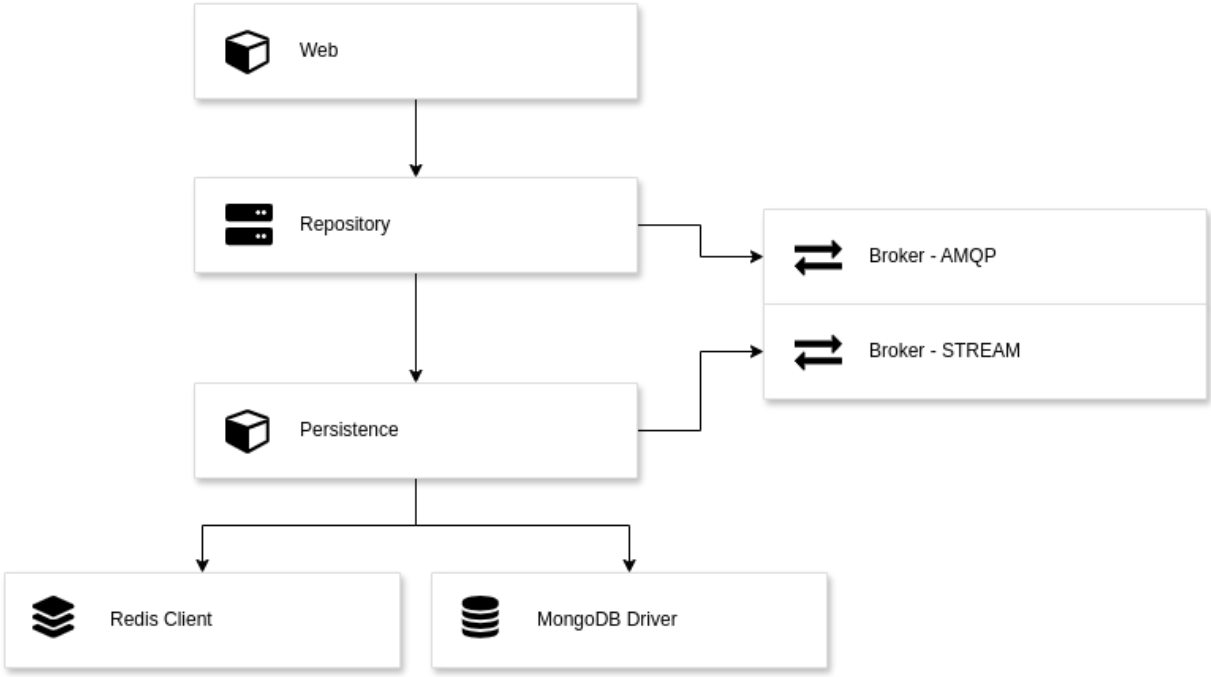

Figure 31 – Application Repository Modules

4.4.2.3 The onboarding process

The onboarding process begins when the information for a single application package is created (5.2) and concludes when the user uploads the package content. A single ZIP file representing the application and all related information. The contents of this ZIP file are referred to as the Application Package. The user can include an arbitrary number of files, which may ultimately be used by the orchestrator to deploy the application.

During this process, the application package onboarding state transitions through different values depending on the phase being handled by the Application Repository. At the end of the process, the state reflects the outcome, either moving to the onboarded state if successful, or to an error state if a failure occurs. In the latter case, the user may repeat the onboarding process by providing a new or corrected package.

4.4.2.3.1 Application Package Onboarding States

The onboarding process for an application package involves five distinct states. Each state is described below, along with its corresponding post-condition:

- *CREATED*: This is the initial state, indicating that the application package virtual resource has been created.
- *UPLOADING*: The user has uploaded the application package, and the Application Repository has stored the uploaded ZIP file, making it persistently available for further processing.
- *PROCESSING*: The Application Repository ensures that the artifacts of each package are made persistent and remain accessible. It also performs validation of both the manifest and descriptor files included in the package.
- *ONBOARDED*: The user has successfully onboarded the application, and the package available for use. This marks the completion of the application onboarding process.
- *ERROR*: Something went wrong during the process. The user may repeat the onboarding procedure to try again.

4.4.2.3.2 Application Package Size

The package contains all the information related to a single application. However, there are no strict boundaries on the size of the package. This is because the Application Repository is designed to be agnostic and transparent with respect to the specific requirements or technologies used by the user or the trial site. As a result, a package can be quite large, for example, if virtual machines (VMs) are included in the package.

This means that the Application Repository should carefully manage the onboarding process of the full package, both when the ZIP file is stored in MongoDB and when its contents are extracted. These operations should avoid using primary memory to hold the entire uploaded package or any individual artefacts provided by the user.

The only two files in the package that are fully loaded into memory are the manifest and the application descriptor.

4.4.2.3.2.1 Streaming the ZIP Upload directly in the MongoDB

When a client submits a ZIP file to the Application Repository, the HTTP request body is exposed internally as an io.Reader[26]. Instead of reading the entire payload into memory, the repository hands this reader over the persistence module, which underlying use the GridFS[27] MongoDB's streaming API.

Internally, the Web module reads from the network socket (incoming TCP packets) in manageable chunks and presents those bytes through the reader interface, while the MongoDB driver offers a writer interface that accepts data in similarly sized segments. By connecting the reader and writer, the application repository creates a continuous pipeline: the web module pulls each segment from

[26] https://go.dev/tour/methods/21
[27] https://www.mongodb.com/docs/manual/core/gridfs/

the client and pushes it directly into MongoDB without ever buffering the full archive in RAM. This process is illustrated in Figure 32.

Figure 32 – Upload the package (shown on the left) by using a buffer in the Application Repository, which then writes the data to MongoDB for persistent storage.

Upload the package (shown on the left) by using a buffer in the Application Repository, which then writes the data to MongoDB for persistent storage.

4.4.2.3.2.2 Io.Reader

In Go[28], an io.Reader is any object that implements the Read method, which fills a provided byte slice with data and reports how many bytes were returned. A buffer in Go is simply a byte slice, usually allocated with a fixed capacity that temporarily holds each chunk as it flows from reader to writer. By reusing the same buffer slice for each read/write cycle, repeated memory allocations are avoided, keeping memory usage constant regardless of the total file size.

4.4.2.3.2.3 Streaming the ZIP Retrieval and Temporary Storage

To extract the contents of the ZIP archive stored in MongoDB, the repository opens a streaming download from GridFS that exposes the stored archive as an io.Reader. Rather than fetching the entire archive into memory, the repository directs this reader into a temporary file on disk by piping each incoming chunk into a file-backed writer. As GridFS yields bytes, the temporary file grows incrementally, and only a small, fixed-size buffer passes through memory at any moment. This guarantees that even multi-gigabyte archives never overwhelm RAM, since each read-and-write cycle handles only a slice of the data before moving on.

After extraction, each file within the archive is uploaded back into MongoDB's GridFS, using a similar streaming process. The only difference is that, instead of reading from an HTTP stream, the repository reads the file contents from disk using a FileReader.

Underneath the hood, the Go runtime schedules the GridFS download and the file-write operations concurrently. The GridFS driver polls the database cursor, reads a block of data into the buffer slice, then immediately returns control to the repository's code. The repository then synchronously writes that slice into the temporary ZIP file on disk. As soon as the write is completed, the next reading begins. Once the reader signals EOF, indicating no more data, the repository closes the temporary file handle. At that point, the repository invokes the standard ZIP decompression routines against the on-disk file: they, in turn, treat the file itself as another reader, pulling out individual file entries and writing each one to its destination directory. Throughout this process, each component, GridFS, the OS file system, and the decompression library, manages its own

[28] https://go.dev/

buffer, but no one ever attempts to load more than a tiny window of data into memory at once. This orchestrated hand-off of byte slices keeps the repository's memory footprint both predictable and minimal. Figure 33, Figure 34, and Figure 35 illustrate these steps.

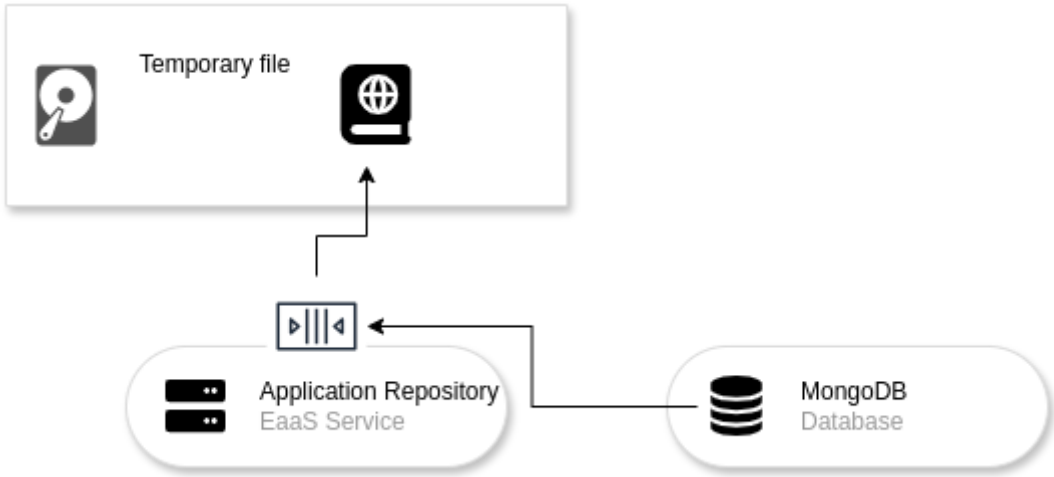

Figure 33 – Download the package (shown on top) by using a buffer in the Application Repository, which then writes the data to a temporary file on disk for further processing.

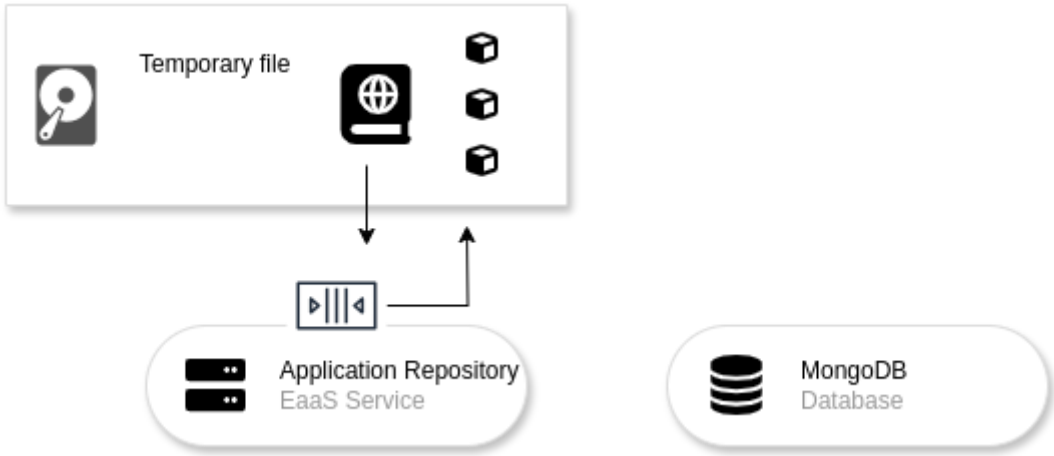

Figure 34 – Unzip the package content (shown on top) by using a buffer in the Application Repository for each file, extracting them one by one without loading the entire archive into memory.

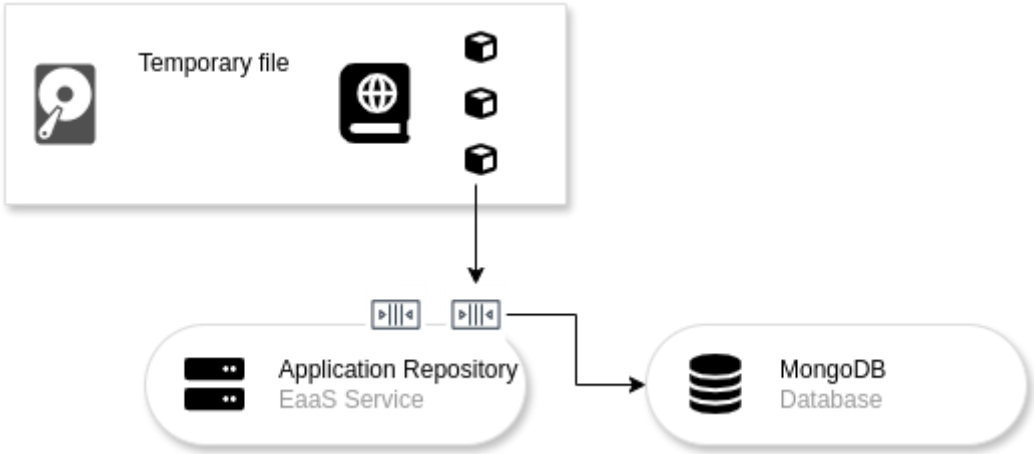

Figure 35 – Upload the package (shown on top) by using a buffer in the Application Repository, saving each extracted file individually into MongoDB.

4.4.2.3.3 Application Package Information Stream

In a microservice architecture, each service owns its own slice of the data model, which keeps responsibilities clear and reduces the risk of unintended side-effects when one component evolves. Yet, in a real-world system, it's common for one service to need up-to-date information about objects managed elsewhere: for example, knowing when an Application Package's information has changed can be essential for services responsible for experiment. By emitting an "update" event every time the Application Package Information model is modified, The Application Repository makes that change discoverable without creating tight coupling or shared databases.

Using RabbitMQ Streams to carry those update messages gives this module several key advantages. Unlike point-to-point calls, a stream lets to broadcast changes to any number of interested consumers, and its built-in persistence and replay capabilities mean that new or recovering services can catch up on missed events simply by resuming the stream. Services can maintain their own local cache of the package data, updating only when it actually changes, which slashes unnecessary database queries and removes synchronous dependencies that could slow down end-user operations.

Moreover, the event-driven pattern implemented gracefully supports eventual consistency: each downstream service applies updates at its own pace, and if one falls behind or temporarily disconnects, it can reprocess the history rather than losing critical state changes. That resiliency is crucial for high-availability systems, where transient failures must not lead to permanently inconsistent views. In short, by isolating the canonical data store yet propagating updates via RabbitMQ Streams, the Application Repository achieve a loosely coupled, scalable and fault-tolerant design in which every service can react to, and record, changes exactly when they occur.

As will be discussed in the following sections, scalability and reliability are critical requirements for this service. Therefore, when linking a database update related to an Application Package Information object with a corresponding stream update message, it's important to acknowledge that these two operations are not atomic. If a disruptive event occurs between the database update and the message publication, it could lead to inconsistencies across consumers relying on the stream.

To mitigate this risk and recognizing that a disruption can occur at any point, the system follows a design where update messages are not pushed to the stream directly during the main execution flow. Instead, the database is updated first, and a separate execution flow monitors changes in the database. This dedicated flow listens for updates and publishes them to the stream in order, ensuring consistency and reliable propagation of changes to other consumers.

4.4.2.3.3.1 Listen update message flow

MongoDB provides a mechanism to listen for document updates, allowing the Application Repository to receive and process Application Package Information documents along with a specific point in time (typically using the change stream resume token). The repository then pushes these updates to the RabbitMQ stream.

However, since a disruption can occur at any point, the Application Repository must persist the last known point in time at which a message was successfully pushed to the stream. If the process is restarted, the repository will use this saved timestamp or resume token to resume listening from that exact point forward by requesting MongoDB to return only the updates after that moment.

This design ensures that no updates are lost, even if the listener stops or crashes for any reason, thereby providing reliability and continuity in the event-driven onboarding workflow.

However, there are two potential issues that must be considered and will be described in the following sections.

4.4.2.3.3.2 Message Replays

The first potential issue is the replay of an update. This can occur if a disruption happens after a message has been pushed to the stream but before the system saves the point in time at which the update was received. If the process restarts under these conditions, it may attempt to push the same message again, resulting in unintended duplicates.

This problem is mitigated by leveraging the publisher ID in each Stream Message. The publishing ID is a strictly increasing sequence that must be incremented by the publishing application for each outbound message. The stream broker maintains a "limit", the highest publishing ID received from a given named producer for a specific stream. It uses this limit to filter out any incoming messages with a publishing ID less than or equal to this value.

In our implementation, the publishing ID is derived from the Unix timestamp corresponding to the exact moment the database document was updated. Only two identical updates would share the same timestamp, and in such cases, only one will be pushed to the stream, ensuring idempotency and consistency.

4.4.2.3.3.3 Scalability Issue

The second consideration relates to scalability. Since each replica of the Application Repository operates independently, they must each listen to database update events to maintain consistency, just as previously described. Thanks to the no-replay guarantee enforced by the use of publishing

IDs, multiple replicas can safely listen to the same events without causing inconsistencies, as only one message will be processed and accepted.

However, while this approach is safe, it is also inefficient. Allowing every replica to listen and compete on every update introduces redundant processing, increases system load, and complicates coordination, especially considering that replicas unable to perform the update must still process and then safely discard the database events they receive.

To mitigate this inefficiency, a leader election mechanism is introduced among the replicas. Each replica periodically attempts to acquire a token, retrying at regular intervals. The replica that wins the election holds the token and becomes responsible for listening to database updates and publishing them to the stream. To maintain leadership, the elected replica must periodically refresh the token. If the leader fails or becomes unresponsive, the token will expire, allowing another replica to detect this and assume leadership. This mechanism ensures high availability while preserving efficiency by delegating stream publishing to a single active replica at any given time.

4.4.2.4 Extendibility

The Application Repository is designed to be extensible without introducing disruptive changes to the overall architecture. It follows a modular design, where each module exposes well-defined functions and communicates with others through clear interfaces. This approach minimizes internal coupling and allows each module to evolve independently.

However, as described in the Application Package section, the Application Descriptor is inherently complex, containing numerous fields to support diverse deployment configurations. The Application Repository maintains an internal representation of the descriptor as structured data and is responsible for validating it. This introduces two key challenges:

1. Validation Complexity – Ensuring that every possible configuration is accurately validated becomes difficult as the descriptor grows in scope.
2. Slow Adaptability – Any change in the descriptor definition may require manual updates to the data structures and validation logic, potentially slowing down the delivery cycle.

To address these issues, the Application Repository uses a YANG model to define the Application Descriptor. Tools like ygot are used to auto-generate both the data structures and their associated validation logic directly from the YANG schema. This means the YANG file acts as the single source of truth, and any changes to the descriptor definition require only an update to the YANG model, without manual intervention in the codebase. While a regeneration step is still required (i.e., re-running the tool to generate the corresponding classes), the process remains automated and consistent, significantly reducing maintenance effort and the risk of human error.

This approach offers several benefits:

- Faster delivery cycles, as structural and validation changes are automated.
- Consistency and maintainability, through a centralized and declarative schema.
- Language agnosticism, allowing services implemented in different languages to generate compatible data structures and validations from the same YANG file.

4.4.2.5 Scalability

The Application Repository is a key service within the EaaS Module, responsible for managing applications and their associated information. Given the resource-intensive nature of application onboarding, which involves both primary and secondary memory usage, as well as multiple I/O operations, this service has been designed to be stateless. Each request is handled independently, allowing the service to scale efficiently. It can be scaled vertically by adding more resources, or horizontally by deploying additional replicas.

The stateless nature of this service enables easier scaling. However, it also introduces additional complexity when addressing reliability, as we will describe in the following section.

When working with data structures, the Application Repository treats all information as transient, acknowledging that data may become stale, especially in a distributed environment. This is true even when only a single replica is deployed, as the Portal may issue multiple concurrent requests acting on the same application package.

To maintain consistency, all conditions are evaluated at the moment each database query is executed, leveraging the atomicity guarantees provided by the underlying database.

For instance, when updating the application package state (e.g., to UPLOADING), the database operation follows a test-and-set approach: the update is applied only if specific conditions are met at the time of execution. This ensures safe state transitions and supports idempotency. The conditions for a valid update to the UPLOADING state are:

- An application package with the specified ID must exist.
- The application package must currently be in the CREATED or ERROR state
    - OR The application package is already in the UPLOADING state, but it must have the same file ID as the ZIP being processed.

This ensures that state transitions and validations reflect the most current and valid data, even in the presence of concurrent operations or race conditions.

4.4.2.6 Reliability

The concept of reliability in this context aligns with the following definition:

The quality of being able to be trusted or believed because of working or behaving well

As mentioned in the previous section, the Application Repository plays a critical role within the EaaS module. Therefore, a high level of reliability must be ensured, especially given its operation under an as-a-service model.

To ensure reliability, the Application Repository must be capable of handling service disruptions and making operations recoverable, without losing messages and always maintaining a consistent state in accordance with the application package model.

Most of the APIs specified in the interface section are synchronous and idempotent, meaning they can be safely retried by the user in the event of a disruption without leading to inconsistent states or unreliable behaviour.

However, there are two operations that require special attention:

- **Application Package Creation** – This operation is synchronous but not idempotent. A disruption during this step can lead to ambiguity about whether the package was successfully created, potentially resulting in duplicate packages.
- **Onboarding Process** – As described here, this operation is asynchronous and spans multiple states and internal steps. Reliability must be ensured throughout the process to prevent partial or inconsistent onboarding results.

4.4.2.6.1 Application Package Creation

This operation creates a new application package, into which the user can eventually upload their application content. In the event of a disruption during this operation, there are two possible outcomes.

The first outcome occurs in the case of a post-insertion disruption, where the new application package is successfully inserted into the database, but a disruption occurs before the success response is returned to the user. As a result, the user may believe that the operation has failed. If the user retries the creation and it succeeds, the result will be two distinct application packages. While this may not be the user's intention, they can review the list of created packages and manually delete any unintended duplicates.

The second outcome occurs when the disruption happens before the new application package is inserted into the database. In this case, it is as if the operation never took place, and no further action is required from the user.

4.4.2.6.2 Onboarding Process

Even though this PUT operation is idempotent, if a disruption occurs at any point during the operation, particularly when the package is in an intermediate state, the user could be not allowed to manually restart the process. In such cases, a reliable system must ensure that the application package eventually transitions to a terminal state, either ONBOARDED or ERROR. This guarantees consistency and enables both the user and the system to confidently determine the outcome of the onboarding process.

Given the stateless nature of the Application Repository, which enhances reliability by ensuring that no internal state is lost in the event of a disruption, two key components support the reliability of the onboarding process:

- **MongoDB** – Acts as the persistent store, maintaining the state of each application package throughout the onboarding process.
- **Message Broker** – Coordinates operations across service replicas. If an instance goes down, the broker ensures that the onboarding workflow continues seamlessly by allowing other replicas to take over the pending tasks.

Together, these components help ensure that the onboarding process remains consistent, recoverable, and fault-tolerant.

As stated in Section 4.4.2.3, there are five distinct states that an application package transitions through during the onboarding process. For the sake of clarity, we assume that no logic errors

(such as invalid zip or invalid descriptor format) occur during the process. Under this assumption, the onboarding process can be summarized with the following multi-stage transition sequence.

4.4.2.6.2.1 Multi-Stage Onboarding Process

The onboarding of an application package is designed as a multi-stage process, ensuring scalability, reliability, and clear state management throughout its lifecycle.

This multi-stage process begins when the user issues an HTTP request to upload the ZIP package. The repository stores the package in MongoDB. Before updating the application package state to UPLOADING or returning an ACCEPTED response, the following steps are executed:

An Application Package Job is created; this is a message containing all the necessary information to process the onboarding task. Specifically, it includes the application package ID, the file ID of the uploaded ZIP archive, and the state to be handled (i.e. UPLOADING). This message is then published to an AMQP queue, where the Application Repository acts as both publisher and consumer. If multiple replicas are deployed, messages are distributed among them using a round-robin policy.

Only after the message is successfully published to the queue, the repository returns the ACCEPTED status to the Portal. From this point onward, the repository is responsible for guaranteeing the onboarding process, ensuring that the package eventually reaches a terminal state, as outlined at the beginning of this section.

The consumer, which may be the same instance that published the message, or another Application Repository replica, reads the Application Package Job message from the queue. Upon receiving the message, this consumer updates the Application Package state to UPLOADING. This marks the beginning of another stage of the onboarding process.

After the Application Package Job is successfully processed, the repository sends an acknowledgement (ACK) to the broker. Once acknowledged, the broker marks the message as consumed and removes it from the queue.

However, if the repository crashes before sending the ACK, the broker will re-enqueue the message. This ensures that the job will eventually be re-processed by the same or another replica of the Application Repository, maintaining the reliability and fault tolerance of the onboarding workflow.

This guarantees that the onboarding process initiated by consuming the Application Package Job can be safely repeated. If a job is retried (e.g., due to a crash), it will only succeed if it refers to the same ZIP file, ensuring that no conflicting operations are performed concurrently. In essence, if two identical flows begin from a CREATED or ERROR state, only one will succeed, and the other will fail cleanly preserving idempotency.

4.4.2.6.2.2 Disruptive Events

In the Application Onboarding Process, a disruptive event can occur at three distinct stages:

Before Queue Publishing
After Queue Publishing, Before Response
After Queue Publishing, After Response

If a disruption occurs before the application package job is published to the AMQP queue (1), the operation is effectively aborted. The user receives a failure response and can retry the upload with either the same or a new ZIP file. In this case, the onboarding phase restarts from the beginning.

If a disruption happens after the message is successfully published to the queue but before the ACCEPTED response is returned to the user (2), the onboarding process will proceed in the background. The message will still be consumed by the repository, and the next stage of onboarding will begin. However, from the user's perspective, the request may appear to have failed.

The actual user experience in this case depends on the Portal's implementation: If the portal listens for application package updates, it will observe the onboarding progress despite the failed response. If it does not, the user may retry the upload. Thanks to the idempotency of the onboarding flow (as long as the file ID remains the same), the retry will be harmless.

Once the ACCEPTED response has been returned to the user and the Application Package Job has been published to the queue, any disruption that occurs thereafter (3) will not affect the user experience, aside from a possible delay in processing.

From this point on, the onboarding process proceeds through a series of background stages. For example, once the uploading operation completes (i.e., the ZIP file has been successfully processed), a new Application Package Job is enqueued. This job follows a similar pattern to the one used for transitioning to the UPLOADING state.
The consumer, again, potentially any replica of the Application Repository, will read this new job, update the application package state accordingly, and initiate the next operation in the sequence. Before successfully processing the job and sending an acknowledgment, the consumer may publish another Application Package Job to advance the onboarding workflow.

This staged, message-driven approach continues until the application package reaches a terminal state, such as ONBOARDED or ERROR. By decoupling each phase into discrete, recoverable jobs, this design ensures fault tolerance and scalability, enabling the onboarding process to recover gracefully from transient failures while maintaining a consistent and reliable state.

4.4.3 Interfaces

4.4.3.1 Application Repository's Interface (Northbound)

This interface of the Application Repository exposes functionality to the ENVELOPE Portal, allowing experimenters to perform application package management operations such as uploading, updating, and deleting them.

4.4.3.1.1 User's Application Packages

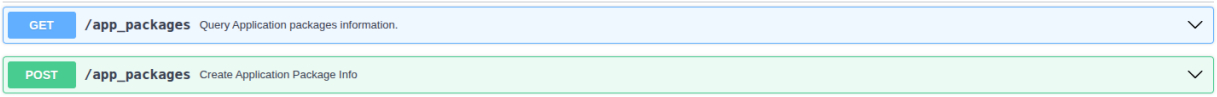

Figure 36 – User's Application Packages swagger

This set of APIs (Figure 36) let the user to retrieve packages and eventually creating new ones:

- Get /app_packages: Fetches information about the application packages owned by the caller. This API supports a query parameter showPublic. When set to true, it also includes all public application packages onboarded by other users in the response. It returns a list of AppPkgInfo
- Post /app_packages: Creates a new Application Package Info resource. The request payload is minimal and includes only userDefinedData.

4.4.3.1.2 Single applications package

Figure 37 – Single applications package swagger

These APIs (Figure 37) manage a single package, retrieve information, updating some information and eventually deleting it:

- GET /app_packages/{appPkgId}: Retrieves information about a specific Application package. It returns a single AppPkgInfo
- PATCH /app_packages/{appPkgId}: Patch the userDefinedData about a specific Application package.
- DELETE /app_packages/{appPkgId}: This endpoint allows deletion of a specific application package, but only if its packageUsageState is NOT_IN_USE. Packages currently in use cannot be deleted.

Parameters:

- appPkgId: the application package's id

4.4.3.1.3 Single application's package content

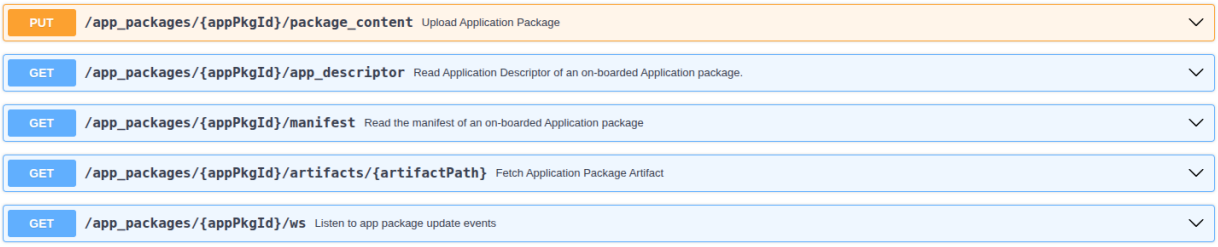

Figure 38 – Single application's package content

This set of APIs (Figure 38) retrieves specific information about an application package that is onboarded, the put API let the user to upload the zip file with all artifacts of the packages starting the application onboarding process:

- PUT /app_packages/{appPkgId}/package_content: This endpoint allows the user to upload the contents of an application package. The request must include a ZIP file as the body, with the Content-Type header set to application/zip.
- GET /app_packages/{appPkgId}/app_descriptor: the GET method reads the content of the Application Descriptor within an Application package. It returns the JSON representation of an Application Descriptor
- GET /app_packages/{appPkgId}/manifest: the GET method reads the content of the manifest within an Application package. It returns an application/text type content of the manifest
- GET /app_packages/{appPkgId}/artifacts/{artifactPath}: Retrieves a specific artifact from an Application package.
- GET /app_packages/{appPkgId}/ws: Open a WebSocket to retrieve application update events (AppPkgEvent).

Parameters:

- appPkgId: the application package's id
- artifactPath: the artifact path within the package

4.4.3.1.4 Schema - UserDefinedData

A simple object composed by arbitrary key:value,

```
- {
        "userDefinedData": {
            "isPublic": false,
            "additionalProp1": {}
        }
  }
```

4.4.3.1.5 Schema - AppPkgInfo

contains information about an application package

```
- {
        "id": "59f31114-7e51-42d2-a498-e55681396df4",
        "appDescriptorId": "nginx-appd-001",
        "appProvider": "123e4567-e89b-12d3-a456-426614174000",
        "appProductName": "NginxApplication",
        "appSoftwareVersion": "1.0.0",
        "packageOnboardingState": "PROCESSING",
        "packageOperationalState": "DISABLED",
        "packageUsageState": "NOT_IN_USE",
        "additionalArtifacts": [
            { "artifactPath": "Artifacts/MCIOPs/app-nginx-0.1.0.tgz" },
            { "artifactPath": "NginxApplication.mf" },
```

```
                { "artifactPath": "NginxApplication.yaml" }
        ],
        "appmInfo": [ "etsiappm:v2.3.1", "0:myGreatAppm-1" ],
        "userDefinedData": {
                "ad_a9c": 19148490.941475943,
                "elit2": 40068639.7627148,
                "isPublic": false,
                "ut2a6": -83890892
        }
}

-
```

4.4.4 Dependencies

Table 5 introduces the key software libraries.

Table 5 – Key Software Libraries

Name	Description
Gofiber	Web Framework
RabbitMQ	Both AMQP and stream client libraries
MongoDB	Mongo driver
GoYANG	Yang library
Redis	Redis library

Other Dependencies

The Application Repository depends on the southbound plugin for what concern the onboarding of an application package.

4.4.5 Application Package

An Application Package is the smallest indivisible object able to be onboarded. It contains all the information related to a single application in terms of artifacts and descriptor. Two specific files are mandatory and must be named <application-name>.mf and <application-name>.yaml. These represent, respectively, the manifest of the package and the descriptor of the application. The <application-name> is an arbitrary name decided by the user.

4.4.5.1 Application Manifest

An Application Manifest (<application-name>.mf) provides metadata about the application, including its version, compatibility information, and a list of files that make up the package. It is the first file to be parsed when processing the package and is considered the entry point that enables the system to interpret the contents of the package.

It consists of two blocks, one is the metadata fields, introduced in Section 4.4.5.1.1, where the application package metadata are listed, the second block consists of a list of all constituent file you want to be onboarded.

4.4.5.1.1 Metadata

The Metadata of the Application Manifest are introduced in Table 6.

Table 6 – list of Metadata for application manifest

Name	Description	Example
appd_id	A unique identifier for this application descriptor.	nginx-appd-001
app_product_name	A human-readable name for your application. <application-name>	NginxApplication
app_provider_id	The UUID of the organization or provider publishing this descriptor.	123e4567-e89b-12d3-a456-426614174000
app_software_version	The version tag of the application image or code itself.	1.0.0
app_package_version	The version of this descriptor package (so you can bump it independently of the software).	1.0
app_release_date_time	The exact release timestamp of this package, in ISO-8601 format with timezone.	2017-01-01T10:00:00+03:00
appm_info	Application-Management (APPM) metadata string(s), often including the APPM schema version and a custom tag.	etsiappm:v2.3.1,0:myGreatAppm-1
compatible_specification_versions	A comma-separated list of Application-Descriptor specification versions that this package supports.	2.7.1,3.1.1

4.4.5.1.2 Sources

Sources (Table 7) list every file that must accompany this manifest. Order is not strict, but including the manifest first is conventional.

Table 7 – list of sources

Sources	Description	Example
Source: <application-name>.mf	The manifest itself.	Source: NginxApplication.mf
Source: <application-name>.yaml	The main Application Descriptor.	Source: NginxApplication.yaml
Source: <file-location>	Any artifacts or other bundles in this package.	Source: Artifacts/MCIOPs/app-nginx-0.1.0.tgz

4.4.5.2 Application Descriptor

An Application Descriptor is a YAML template that lets you define all the metadata, deployment parameters, and interfaces for an application you want to deploy.

4.4.5.2.1 Metadata

Similar to the manifest, the descriptor must include metadata fields (Table 8) that identifies the application uniquely and provides versioning info.

Table 8 – list of Metadata for application descriptor

Name	Description	Example
appdId	A unique identifier for this application descriptor.	nginx-appd-001
appdExtInvariantId		nginx-appd-extinv-001
appProvider	The UUID of the organization or provider publishing this descriptor.	123e4567-e89b-12d3-a456-426614174000
appProductName	A human-readable name for your application. <application-name>	NginxApplication
appSoftwareVersion	The version tag of the application image or code itself.	1.0.0
appdVersion	The version of this descriptor package (so you can bump it independently of the software).	1.0
appmInfo	Application-Management (APPM) metadata string(s), often including the APPM	Generic-ApplicationM

	schema version and a custom tag.	

4.4.5.2.2 OS Container Descriptor(s)

An OS Container Descriptor (Table 9) specifies resource requirements and links to software images, and it describes the resources needed for containers or VMs.

Table 9 – OS Container Descriptor fields

Name	Description	Example
osContainerDescId	Unique identifier of this OsContainerDesc in this descriptor.	nginx_container_desc_id
name	Human readable name of this OS container.	Nginx Desc
description	Human readable description of this OS container.	Nginx Container Descriptor
requestedCpuResource	Number of CPU resources requested for the container	1
requestedMemoryResource	Amount of memory resources requested for the container	586
cpuResourceLimit	Number of CPU resources the container can maximally use	1
memoryResourceLimit	Amount of memory resources the container can maximally use	1024
swImageDesc	Describes the software image realizing this OS container.	nginx_image_id

4.4.5.2.3 Virtual Deployment Unit(s) (VDU)

Table 10 reports the VDU that groups containers/VMs into logical units.

Table 10 – Virtual Deployment Unit fields

Name	Description	Example
vduId	Unique identifier of this VDU in this descriptor.	VDU_nginx
name	Human readable name of the VDU.	Nginx Desc

description	Human readable description of the VDU.	Nginx Container Descriptor
mcioIdentificationData	Name and type of the MCIO that realizes this VDU.	helm
osContainerDesc	Describes CPU, memory requirements and limits, and software images of the OS Containers realizing this VDU.	nginx_container_desc_id

4.4.5.2.4 Deployment Flavours

A deployment flavour (Table 11) describes a specific deployment version of an application in terms of sizing/performance profiles and lifecycle parameters. It defines “flavours” (e.g., small, medium, large) with scaling and lifecycle settings.

Table 11 – Deployment Flavours fields

Name	Description	Example
flavourId	Identifier of this DF within this descriptor.	df_nginx
description	Human readable description of the DF.	Default Deployment flavour for Nginx
vduProfile	Describes additional instantiation data for the VDUs used in this flavour.	See below
instantiationLevel	Describes the various levels of resources that can be used to instantiate the Application using this flavour	See below

4.4.5.2.4.1 VDU Profile

VDU profile fields are reported in Table 12.

Table 12 – VDU Profile fields

Name	Description	Example
flavourId	Identifier of this DF within this descriptor.	df_nginx
description	Human readable description of the DF.	Default Deployment flavour for Nginx

vduProfile	Describes additional instantiation data for the VDUs used in this flavour.	See below
instantiationLevel	Describes the various levels of resources that can be used to instantiate the Application using this flavour	See below

4.4.5.2.4.2 Instantiation Level

Table 13 introduces the instantiation level fields.

Table 13 – Instantiation Level fields

Name	Description	Example
levelId	Uniquely identifies a level within the DF.	default_level
description	Human readable description of the level.	Default instanton level
vduLevel	Indicates the number of instances of this VDU to deploy for this level.	See below

4.4.5.2.4.3 MciopProfile

The MciopProfile fields are reported in Table 14.

Table 14 – MciopProfile fields

Name	Description	Example
mciopId	Identifies the MCIOP in the Application package.	Artifacts/MCIOPs/app-nginx-0.1.0.tgz
associatedVdu	List of VDUs which are associated to this MCIOP and which are deployed using this MCIOP.	VDU_nginx

4.4.5.2.5 External Interfaces

The external interfaces (Table 15) lists the connection points that are exposed externally.

Table 15 – External Interfaces fields

Name	Description	Example

cpdId	Identifier of this Cpd information element.	Ext
virtualCpd		Nginx80_connection_point

4.4.5.2.6 Virtual Connection Points

The Virtual Connection Points fields (Table 16) define the network interfaces, within the descriptor, that map to containers/VDUs.

Table 16 – Virtual Connection Points fields

Name	Description	Example
cpdId	Identifier of this Cpd information element.	nginx80_connection_point
layerProtocol	Specifies which protocol the CP uses for connectivity purposes.	IPV4 IPV6
description	Provides human-readable information on the purpose of the CP	Connection point to nginx service on port 80
vdu	References the VDU(s) which implement this service.	VDU_nginx
additionalServiceData	Additional service identification data of the VirtualCp exposed	portData (see Table 17)

4.4.5.2.6.1 Additional Service Data

Additional Service Data fields are reported in Table 17.

Table 17 – Additional Service Data fields

Name	Description	Example
name	The name of the port exposed by the CP.	nginx_port
protocol	The L4 protocol for this port exposed by the CP.	TCP
port	The L4 port number exposed by the CP.	80

portConfiguration	Specifies whether the port attribute value is allowed to be configurable.	false

4.5 Experiment descriptor manager

4.5.1 Overview

This component handles user requests for querying, creating, and storing experiment descriptor information. The user can access the functionality of the Experiment descriptor manager via the exposed API endpoints, using the EaaS portal frontend. The experiment descriptors collect all information about a specific experiment, e.g., an identifier, the applications required, including deployment-specific parameters, etc. The manager stores these descriptors to be fetched later before an experiment is instantiated.

Specified by the functional requirements in Deliverable D2.2, the experiment descriptor and/or the descriptor repository must provide the functionalities that are reported in Table 18.

Table 18 – mapping functionalities to requirements

Functionality	Requirement
The experimentation descriptor shall allow to indicate the applications chain making up the experimentation	REQ-F-Task3.1-Eaas-11
The experimentation descriptor shall allow to indicate the amount of computing resources (i.e., storage, RAM, CPU) needed by each application in the experimentation	REQ-F-Task3.1-Eaas-12
The experimentation descriptor shall allow to indicate the amount of network resources (e.g., type of network slice or 5QI) needed by each application in the experimentation	REQ-F-Task3.1-Eaas-12
The Experiment lifecycle manager shall allow to store the descriptor of a given experimentation for later (re)use by the experimenter	REQ-F-Task3.1-Eaas-15

4.5.2 Architecture & Design

As the Experiment Descriptor Manager module manages incoming requests concerning a user's experiment descriptors, it consists of two main modules:

- an HTTP server implementation handling the REST API endpoints, and

- a database storing the experiment descriptor resources and other related data.

The HTTP server implements all functionality regarding the experiment descriptors, including the creation of new resource stubs, the processing and validation of uploaded experiment descriptor resources, and the retrieval of saved resources to be used when running experiments. The database stores the state of the manager, i.e., the experiment descriptor resources.

4.5.2.1 Creating an Experiment Descriptor resource

To create and store an Experiment Descriptor resource, the user must go through the following steps:

1. Issue a call to the appropriate API endpoint to receive a resource stub (*ExperimentDescriptorInfo*) with a unique ID
2. Fill in the stub with the required descriptor metadata (e.g., name, designer, version), the application package IDs required for the experiment, and any other relevant user-defined key-value pairs (*userDefinedData*)
3. Create a manifest file with additional metadata and a list of files uploaded
4. Create the uploadable descriptor archive package, including a YAML file with the ExperimentDescriptorInfo as its content, the manifest file, and any other relevant files
5. Upload the descriptor archive package using the appropriate API endpoint

4.5.3 Interfaces

4.5.3.1 Experiment Descriptor Manager's Northbound Interface

This interface exposes functionality of the Experiment Descriptor Manager to the ENVELOPE Portal, allowing experimenters to perform experiment descriptor management operations such as creating, uploading, and fetching them.

4.5.3.1.1 User's Experiment Descriptors

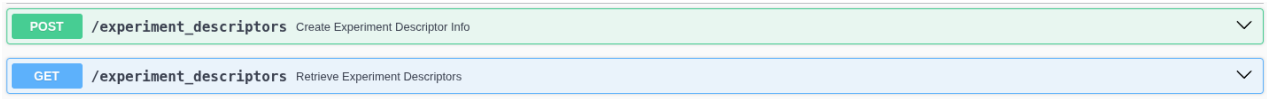

Figure 39 – User's Experiment Descriptors swagger

The family of APIs, depicted in Figure 39, that let the user to create new experiment descriptors and retrieve existing ones are:

- GET /experiment_descriptors: This method lets the user of the EaaS platform fetch the Experiment Descriptor IDs created by the user. The method is invoked through the EaaS portal, and a successful response returns an array of the IDs of all descriptors created by the invoking user.
- POST /experiment_descriptors: This method creates a new Experiment Descriptor information resource with a generated unique ID. The request payload is minimal and includes only userDefinedData. The method is invoked through the EaaS portal, and a successful response returns a partially filled ExperimentDescriptorInfo to be filled in completely by the invoking user.

4.5.3.1.2 Single experiment descriptor

Figure 40 – Single experiment descriptor swagger

These APIs (Figure 40) manage a single experiment descriptor resource, allowing the user to fetch or archive (store) a specific descriptor resource:

- GET /experiment_descriptors/{experimentDescriptorId}: This method reads information about an individual Experiment Descriptor previously archived. The method is invoked through the EaaS portal, and a successful response returns the descriptor information resource (ExperimentDescriptorInfo) matching the ID in the request to the invoking user.
- DELETE /experiment_descriptors/{experimentDescriptorId}: An individual experiment descriptor resource can be deleted using this method.
- PUT /experiment_descriptors/{experimentDescriptorID}/archive_content: This method lets the user upload Experiment Descriptor archive content to the manager. The archive contains the experiment descriptor information and other required metadata. The method is invoked through the EaaS portal.

4.5.3.1.3 Schema - UserDefinedData

A simple object of arbitrary key:value pairs,

```
{ "userDefinedData":   {
          "additionalProp1": {}
          }
}
```

4.5.3.1.4 Schema - ExperimentDescriptorInfo

Contains information about an experiment descriptor resource

```
{
 "appPkgIds": [
  "nginx-appd-001",
  "busybox-appd-001"
 ],
 "experimentDescriptorDesigner": "ExampleDesigner",
 "experimentDescriptorExtInvariantId": "",
 "experimentDescriptorId": "my_experiment_1",
 "experimentDescriptorInvariantId": "my_experiment",
 "experimentDescriptorName": "My Experiment",
 "experimentDescriptorOnboardingState": "CREATED",
 "experimentDescriptorOperationalState": "ENABLED",
 "experimentDescriptorUsageState": "NOT_IN_USE",
 "experimentDescriptorVersion": "1.0",
```

```
  "id": "my_experiment_1"
}
```

4.5.4 Dependencies

4.5.5 Experiment Package

An Experiment Package contains all the information related to a single experiment in terms of artifacts and descriptor.

4.5.5.1 Experiment Manifest

The Experiment Descriptor Manifest (.mf) is a lightweight YAML-style file that describes:

- Who created the experiment descriptor.
- Which version of the descriptor schema it follows.
- When it was released.
- Which other files (e.g., the main YAML descriptor) are part of this experiment package.
- Which specification versions it's compatible with.

This manifest makes it easy for tooling and users to check compatibility before attempting to deploy or process the descriptor and it is the file used by the module to parse the package. The fields of the experiment descriptor are summarized in Table 19.

Table 19 – Experiment descriptor fields

Name	Description	Example
experimentd_designer	Who authored or maintains this descriptor (e.g., a person, team, or organization).	ExampleDesigner
experimentd_invariant_id	A stable machine-friendly ID that never changes, even if you bump the manifest version.	my_experiment
experimentd_name	A human-readable title shown in UIs or logs	MyExperiment
experimentd_file_structure_version	The version of the manifest schema itself. Increment this if you ever change the manifest format.	1.0
experimentd_release_date_time	When this descriptor version was released, in ISO-8601 format (including timezone).	2018-04-08T10:00+08:00

compatible_specification_versioexperiment	Which experiment-descriptor specification versions this manifest supports, comma-separated.	2.7.1,3.1.1,4.3.1

Tip: Always update experimentd_release_date_time when you publish a new descriptor version, and add any new compatible spec versions here.

4.5.5.1.1 *Sources*

Table 20 introduces the metadata of the sources files, listing each file that constitutes this experiment package. In your example:

Table 20 – sources files

Sources	Description	Example
Source: <experiment-name>.mf	The manifest itself.	Source: MyExperiment.mf
Source: <experiment-name>.yaml	The main Experiment Descriptor.	Source: MyExperiment.yaml

Each line names a file that must accompany the manifest.

Ordering is not critical, but including the manifest itself first helps humans and tools verify integrity before loading the descriptor.

Experiment Descriptor

An Experiment Descriptor is a YAML template that lets you define a multi-application experiment, including which application descriptors to include, how many instances to run, and security settings. This manual will help you:

- Understand each top-level section.
- Identify mandatory vs. optional fields.
- Create your own descriptor by modifying the provided example.

Use your own naming convention for experimentdIdentifier and experimentdInvariantId to ensure uniqueness.

4.5.5.1.2 *Identity, Designer and included Applications (appdid)*

List the Application Descriptor IDs that this experiment will deploy: each entry must match a valid appdId from your Application list. It is possible to include as many applications as needed. Fields are introduced in Table 21.

Table 21 – application describer fields

Name	Description	Example
experimentdIdentifier	Identifier of this descriptor information element.	my_experiment_1
designer	Specifies the designer of this experiment.	ExampleDesigner
experimentdName	Provides the human readable name of the experiment	My Experiment
experimentInvariantId	Identifies a descriptor in a version independent manner.	my_experiment
appdid	References the application descriptors of this experiment.	- nginx-appd-001 - busybox-appd-001

4.5.5.1.3 *Experiment Deployment Flavours (experimentDf)*

It defines one or more Deployment Flavours, similar to “profiles”, that group application profiles under a flavour key. Table 22 introduces Deployment flavours fields.

Table 22 – Deployment flavours fields

Name	Description	Example
experimentDfId	unique within this descriptor.	df_my_experiment
flavourKey	arbitrary tag (e.g., “small”, “test”).	string
appProfile	List of app profile	See below

Fields of AppProfile are introduced in Table 23.

Table 23 – AppProfile

Name	Description	Example
appProfileId	local reference for the profile.	experiment_nginx_profile_1
appdId	link to the original application descriptor.	nginx-appd-001
appdExtInvariantId	link to the original application descriptor.	nginx-appd-extinv-001
flavourId	refer to the Application Descriptor own flavours.	df_nginx
instantiationLevel	refer to the Application Descriptor own level.	default_level

minNumberOfInstances	control scaling bounds.	1
maxNumberOfInstance	control scaling bounds.	1

4.5.5.1.4 *Experiment Instantiation Levels (experimentInstantionLevel)*

It maps profiles to actual instance counts for a given experiment level as detailed in Table 24.

Table 24 – Experiment Instantiation Levels

Name	Description	Example
nsLevelId	Identifier of this NsLevel information element. It uniquely identifies an NS level within the DF.	default_experiment_level
description	Human readable description of the Experiment level	Simple experiment level
appToLevelMapping	Specifies the profile of the Applications involved in this Experiment level and, for each of them, the required number of instances.	See below

nsLevelId: e.g., “default”, “scale-out”, etc.

4.5.5.1.5 *appToLevelMapping*

It ties each appProfileId to an exact replica count (Table 25).

Table 25 – appProfileId fields

Name	Description	Example
appProfileId	References the profile to be used for a VNF involved in an NS level.	experiment_nginx_profile_1
numberOfInstance	Specifies the number of VNF instances required for an NS level.	1

4.5.5.1.6 *Security*

It provides a signature section to prevent tampering (Table 26).

Table 26 – signature fields

Name	Description	Example
signature	cryptographic signature of the descriptor.	<string>

algorithm		SHA256
certificate	reference to a certificate (optional).	

4.6 Experiment lifecycle manager

4.6.1 Overview

This component manages the lifecycle of specific experiment instances from creation, through instantiation, to termination. The user can create individual experiment instances based on the experiment descriptors and manipulate them using the API endpoints exposed by the lifecycle manager. The lifecycle manager lets the user monitor the experiment lifecycle status. The lifecycle manager also works as an interface between the user (using the EaaS portal frontend) and site-specific southbound plugins, handling the site-specific cloud infrastructure and orchestrators. The lifecycle manager receives API requests from the user and manages the experiment lifecycle events by invoking the exposed API endpoints of the southbound plugins.

Specified by the functional requirements in Deliverable D2.2, the experiment lifecycle manager must provide the functionalities introduced in Table 27.

Table 27 – mapping functionalities to requirements

Functionality	Requirement
The Experiment lifecycle manager shall request to the compute orchestrator the computing resources for each application of the experimentation as requested by the experimenter in the descriptor of the experimentation	REQ-F-Task3.1-Eaas-17
The Experiment lifecycle manager shall request to the network orchestrator the networking resources for each application of the experimentation as requested by the experimenter in the descriptor of the experimentation	REQ-F-Task3.1-Eaas-18

4.6.2 Architecture & Design

As the Experiment Lifecycle Manager module consists of two main modules:

- an HTTP server implementation handling the REST API endpoints, and
- a database storing the lifecycle state of the experiment instances.

The HTTP server implements all functionality regarding the experiment instances, including the creation, fetching, instantiation, termination, and deletion of experiment instances. The Lifecycle

Manager calls APIs exposed by the Experiment Descriptor manager and the Southbound plugins to achieve its functionality.

4.6.3 Interfaces

4.6.3.1 Experiment Lifecycle Manager Northbound Interface

This interface exposes functionality of the Experiment Lifecycle Manager to the ENVELOPE Portal, allowing experimenters to perform interactions with individual experiment instances such as creating, fetching, instantiating, terminating, and deleting them.

4.6.3.1.1 User's Experiment Instances

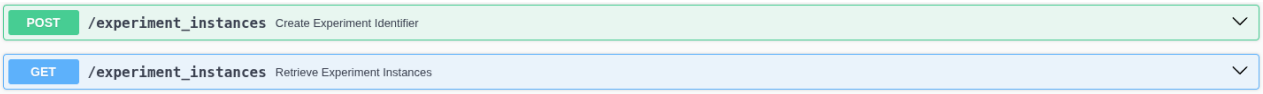

Figure 41 – User's Experiment Instances swagger

The family of APIs (Figure 41) that lets the user to create new experiment instance resources and retrieve existing ones.

- GET /experiment_instances: This method lets the user of the EaaS platform fetch the Experiment Instances created by the user. The method is invoked through the EaaS portal, and a successful response returns an array of the ExperimentInstance resources of all experiments created by the invoking user.
- POST /experiment_instances: This method creates a new Experiment Instance resource. The request payload is minimal and includes only CreateExperimentRequest, which consists of an experiment descriptor ID, the experiment name, and a description. The method is invoked through the EaaS portal, and a successful response returns the created ExperimentInstance resource to the invoking user.

4.6.3.1.2 Single experiment instance

Figure 42 – Single experiment instance

These APIs (Figure 42) manage a single experiment instance, allowing the user to fetch, delete, instantiate, or terminate a specific experiment instance:

- GET /experiment_instances/{experimentInstanceId}: This method reads information about an individual Experiment Instance previously created. The method is invoked through the EaaS portal, and a successful response returns the experiment instance resource (ExperimentInstance) matching the ID in the request to the invoking user.
- DELETE /experiment_instances/{experimentInstanceId}: An individual experiment instance resource can be deleted using this method, but only if the state of this resource is NOT_INSTANTIATED.

- POST /experiment_instances/{experimentInstanceId}/instantiate: The user can instantiate an existing experiment instance resource, specified by the request parameter, using this method. The request body is an InstantiateExperimentRequest object which includes other request parameters for the infrastructure orchestrating the applications required for the experiment.
- POST /experiment_instances/{experimentInstanceId}/terminate: The method terminates a running experiment instance identified by the request parameter. The request body is a TerminateExperimentRequest object with additional parameters for the termination process.

4.6.3.1.3 Schema - ExperimentInstance

Contains information about an experiment instance resource


```
{
  "experimentDescriptorId": "my_experiment_1",
  "experimentDescriptorInfoId": "my_experiment_1",
  "experimentInstanceName": "ExampleExperiment",
  "experimentState": "NOT_INSTANTIATED",
  "id": "51eb5ff7-c36f-4b16-b118-283156034759"
}
```


4.6.4 Dependencies

None

4.7 Southbound plugins

4.7.1 Overview

The southbound plugin component is responsible for abstracting and exposing capabilities of the underlying MANO layer to other EaaS components. Two main EaaS components interact with the southbound plugin layer:

- **Application repository**: It invokes the southbound plugin layer for onboarding a particular application.
- **Experiment Lifecycle Manager**: It manages the experiment lifecycle events by invoking the exposed API endpoints of the southbound plugins, e.g., by triggering the start/stop of an application instance and to monitor the status of an application instance.

4.7.2 Architecture & Design

The southbound layer integrates the EaaS module with the underlying MANO layer by defining southbound APIs that abstract plugin-specific APIs. This modular approach allows various plugins to be deployed, making the architecture flexible.

Figure 43 shows the architecture overview of the southbound layer and other EaaS components. Application onboarding is triggered by the application repository when a new user application is uploaded.

Typically, the experimentation lifecycle manager uses the southbound layer to interact with the MANO layer for application lifecycle control. In some cases, the underlying layer includes its own EaaS modules and lifecycle manager, allowing direct invocation from the EaaS portal.

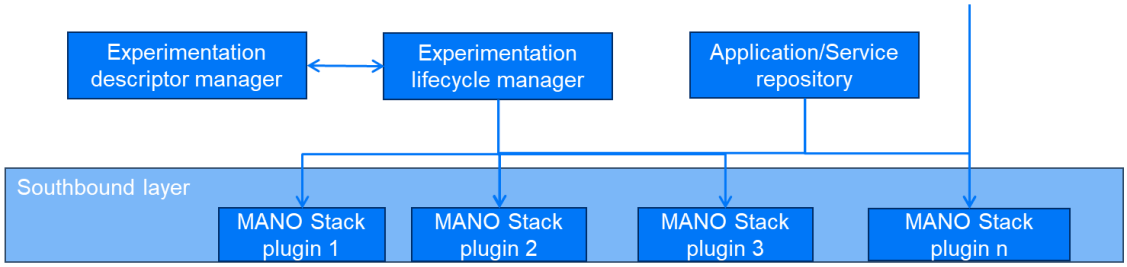

Figure 43 – Architecture overview of the southbound layer

4.7.3 Interfaces

4.7.3.1 Southbound plugin's Interface

These APIs are the southbound interfaces of the plugins, and they are northbound interfaces from the MANO layer perspective.

4.7.3.1.1 Application Onboarding

Figure 44 – Application Onboarding swagger

This API (Figure 44) triggers the onboarding of an application in the underlying Management and Orchestration (MANO) solution:

- POST /application_onboarding: Onboards the application in the underlying MANO layer. The request payload requires the AppPkgInfo structure, including the extra field, i.e., the onboarding id of this Application package, that is allocated by the Southbound Plugin. This extra onboarding id shall be present after the application package content has been on-boarded and absent otherwise. A successful response returns the onboarding id.

4.7.3.1.2 Application Instance

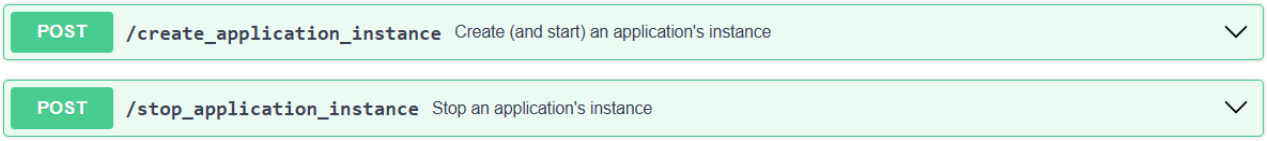

Figure 45 – Application Instance

These APIs (Figure 45) manage the application instance creation and deletion in the underlying MANO layer:

- POST /create_application_instance: Creates and starts the application instance in the underlying MANO layer. The CreateInstanceApplicationRequest structure is required as request payload. A successful response returns the application instance id.
- POST /stop_application_instance: Stops and deletes the application instance in the underlying MANO layer. The StopInstanceApplicationRequest structure is required as

request payload, which includes the application id returned during the creation of the application instance.

4.7.3.1.3 Single application's instance state

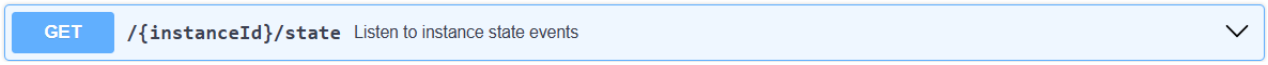

Figure 46 – Single application's instance state

This API (Figure 46) retrieves the state information of a single application instance:

- GET /{instanceId}/state: Retrieves the state information of a single application instance. It requires the *instance Id* string as query parameter. It returns the current status of the application instance in string format: NOT_INSTANTIATED, INSTANTIATED, INSTANTIATING, FAILED.

4.7.3.1.4 Schema - AppPkgInfo

This schema contains information about an application package, including the extra onboarding id field.


```
-   {
        "id": "59f31114-7e51-42d2-a498-e55681396df4",
        "_onboardingId": "c10407ac-31f3-4ee9-a51a-ba1171cd4ca1",
        "appDescriptorId": "nginx-appd-001",
        "appProvider": "123e4567-e89b-12d3-a456-426614174000",
        "appProductName": "NginxApplication",
        "appSoftwareVersion": "1.0.0",
        "packageOnboardingState": "PROCESSING",
        "packageOperationalState": "DISABLED",
        "packageUsageState": "NOT_IN_USE",
        "additionalArtifacts": [
            { "artifactPath": "Artifacts/MCIOPs/app-nginx-0.1.0.tgz" },
            { "artifactPath": "NginxApplication.mf" },
            { "artifactPath": "NginxApplication.yaml" }
        ],
        "appmInfo": [ "etsiappm:v2.3.1", "0:myGreatAppm-1" ],
        "userDefinedData": {
            "ad_a9c": 19148490.941475943,
            "elit2": 40068639.7627148,
            "isPublic": false,
            "ut2a6": -83890892
        }
    }
```


4.7.3.1.5 Schema - CreateInstanceApplicationRequest

This schema contains the data structure for requesting the creation of an application Instance.

```
-   {
            "appOnboardingId": " c10407ac-31f3-4ee9-a51a-ba1171cd4ca1",
             "additionalParams": {
                    "additionalProp1": {}
    }
```

4.7.3.1.6 Schema - StopInstanceApplicationRequest

This schema contains the data structure for requesting the stop/deletion of an application Instance.

```
-   {
            " appInstanceId": " d90907ac-31f4-4ae9-a51a-ba1171cd4df4",
    }
```

4.7.4 Dependencies

The southbound layer interacts with the application repository for onboarding applications and with the experimentation lifecycle manager for controlling the application lifecycle, including instantiation and deletion. Other components are specific to the implementation of the underlying MANO layer.

4.7.5 Southbound plugin integration

4.7.5.1 Italian site

In the Italian site, the southbound plugin integration is achieved by mapping the southbound plugin APIs to the underlying Nextworks Service Orchestrator APIs (Table 28).

Application onboarding is done by mapping the Application Description with the Vertical Service Blueprint of the Nextworks Orchestrator, while application lifecycle management is translated into the Networks Service Lifecycle Manager APIs.

Table 28 – mapping from southbound plugin APIs to the underlying Nextworks Service Orchestrator APIs

Southbound plugin API	Nextworks Service Orchestrator API
POST /application_onboarding	POST /api/v1/vsb-catalogue/
POST /create_application_instance	POST /api/v1/service-lcm/create-instance
POST /stop_application_instance	POST /api/v1/service-lcm/{instanceId}/terminate
GET /{instanceId}/state	GET /api/v1/service-lcm/{instanceId}

4.7.5.2 Dutch site

In the Dutch site, the southbound plugin integration is done by mapping the southbound plugin APIs to the underlying Liqo-based Kubernetes multi-cluster enabler that exposes CAMARA Edge

Cloud (Edge Application Management) APIs, as described previously in section 4.4.3. The API mappings is summarized in Table 29.

Table 29 – mapping from Southbound plugin API to CAMARE edge Cloud APIs

Southbound plugin API	CAMARA Edge Cloud (Edge Application Management) API
POST /application_onboarding	POST /apps
POST /create_application_instance	POST /appinstances
POST /stop_application_instance	DELETE /appinstances/{appInstanceId}
GET /{instanceId}/state	GET /appinstances (with {appInstanceId} as query parameter)

The CAMARA Edge Cloud APIs transformation functions rely on FastAPI libraries as web framework for building HTTP-based service APIs in Python 3.8+. The program is containerized with Docker[29] v3 and exposes port 8000.

4.7.5.3 Greek site

In the Greek site, the southbound plugin integration is done by mapping the southbound plugin APIs to the underlying aerOS-based enabler which serves as an abstraction layer for the aerOS continuum. The API mappings are the ones introduced in Table 30.

Table 30 – mapping from Southbound plugin APIs to aerOS APIs endpoints

Southbound plugin API	aerOS API endpoints
POST /application_onboarding	POST /hlo_fe/services/{service_id}
POST /create_application_instance	PUT /hlo_fe/services/{service_id}
POST /stop_application_instance	DELETE /hlo_fe/services/{service_id}
GET /{instanceId}/state	GET /hlo_fe/services/{service_id}

4.8 Trial site capabilities

4.8.1 Overview

The term capabilities in this context refers to multiple categories of information. It encompasses the ENVELOPE Enablers supported by the Trial site, including the available ENVELOPE APIs. Additionally, it includes details about the computing resources (such as far-edge devices, edge servers, and cloud infrastructure) and the network resources (including supported Camara QoS

[29] https://www.docker.com/

Profiles).
The Trial Site capabilities consist of three different parts. The first is the set of available APIs that users can utilize when running their experiments. The Portal retrieves this list from a static file, which contains an OpenAPI (Swagger[30]) definition that users can view directly within the Portal. The second part concerns network capabilities, which include a list of Quality of Service (QoS) profiles available at the trial site. The third part involves computational capabilities, which are exposed through the CAMARA Edge Cloud APIs provided by the trial site. These APIs describe the available computational resources and deployment zones.

4.8.1.1 ENVELOPE API Enablers

Each trial site provides a list of ENVELOPE APIs available at that location, including each API's name, description, and a link to its OpenAPI (Swagger) specification, so that users can view this information directly through the portal

4.8.1.2 Network Resource

Each trial site also provides a list of CAMARA QoS profiles that a user's application can request.

4.8.1.3 Computation Resource

The Portal requests computational resources directly from the Trial Site by calling the CAMARA Edge Cloud APIs.

4.9 Monitoring

4.9.1 Overview

The Monitoring Platform serves two primary functions within the system architecture:

1. Support for Experimentation and Evaluation: The platform collects and exposes real-time metric data essential for experimenters. This data enables continuous monitoring of ongoing experiments, facilitating immediate feedback during execution, and providing a reliable basis for post-experiment evaluation and analysis. Through intuitive dashboards and programmatic access, experimenters gain insights into service behaviour, system load, and other key performance indicators relevant to their use cases.
2. Integration with Management and Orchestration Functions: The Monitoring Platform plays a critical role in the broader system management by supplying metrics to the Management and Orchestration layer. These metrics offer visibility into infrastructure resource usage, network health, and service performance, supporting timely and informed operational decisions. More specifically, the platform continuously streams telemetry data to Closed-Loop Analysis components, enabling proactive, data-driven adaptation and optimization of services.

[30] https://swagger.io/

Therefore, the Monitoring Platform is designed to cater to the specific data needs of two primary user groups: the human experimenters conducting research and analysis, and the automated Closed-Loop Analysis functions responsible for intelligent system management.

4.9.2 Architecture & Design

This monitoring system is designed as a modular, containerized platform for ingesting, processing, and visualizing metrics data, primarily for network, infrastructure, and application metrics. It is implemented as a **microservice architecture** using Docker Compose, ensuring scalability, portability, and ease of deployment.

At the core of the system is **Prometheus**[31], which serves as the time-series database and rule evaluation engine. Prometheus aggregates metrics from multiple sources, enabling centralized monitoring and analysis.

Data ingestion is achieved through two primary pipelines:

- The first pipeline utilizes an **MQTT broker** (Eclipse Mosquitto[32]) as the central hub for metrics data. Devices or services publish metrics to MQTT topics, which are then scraped by dedicated exporters: *mosquitto-exporter* exposes broker-level statistics such as client counts and topic metrics, while mqtt-exporter subscribes to specific MQTT topics and translates incoming messages into Prometheus-compatible metrics.
- The second pipeline uses an **HTTP REST API** built with **FastAPI**[33]. This API exposes a secure endpoint for external services to submit metrics in JSON format. Internally, the API updates Prometheus-compatible metrics, which Prometheus scrapes at regular intervals.

In addition to these main ingestion methods, the architecture is designed to be **extensible**, allowing the integration of **additional HTTP endpoints** as internal data ingestion pipelines. These endpoints are not intended for external experimenters but rather to monitor internal services, processes, or components within the system itself.

For visualization and analysis, the system integrates **Grafana** as the frontend. Grafana connects directly to Prometheus, enabling us to build dashboards that provide insights into system health, resource utilization, and network activity. Configurations such as dashboards, data sources, and visualizations are managed through mounted configuration files, allowing for easy customization and efficient deployment

The system is optimized for near-real-time monitoring, with Prometheus scraping targets every second. All components are containerized for modularity and ease of deployment, with configuration files organized in a structured directory layout.

This design enables a flexible and extensible monitoring platform capable of integrating both push-based (MQTT) and pull-based (HTTP API) data sources, delivering a robust solution for monitoring complex systems. The overview design diagram can be seen in the Figure 47.

---

[31] https://prometheus.io/
[32] https://mosquitto.org/
[33] https://fastapi.tiangolo.com/

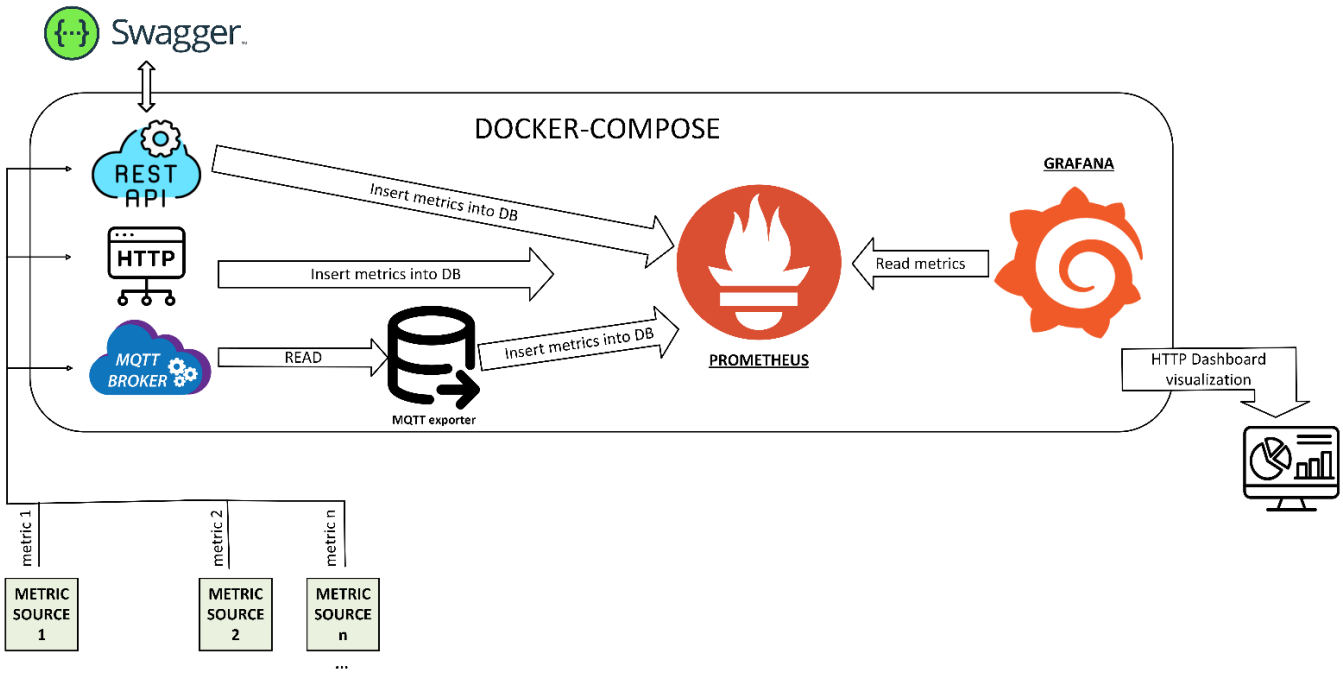

Figure 47 – Monitoring system design diagram

4.9.3 Interfaces

4.9.3.1 Interfaces to ingest data

Prometheus supports several **metric types**, each designed for specific use cases in monitoring and time-series analysis. The two metric types, introduced in Table 31, are considered the most relevant in ENVELOPE:

Table 31 – ENVELOPE most relevant metrics

Type	Description	Example Use Cases
Gauge	Represents a single numerical value that can go up or down. Suitable for current states.	Number of active sessions, Allocated and total IP count.
Counter	A cumulative metric that increases monotonically, suitable for events that accumulate over time.	Dropped packages, forwarded bytes, total packets processed.

Prometheus scrapes data from multiple sources, configured in the *prometheus.yml*.

MQTT Ingestion Pipeline

The system integrates an **MQTT broker** (Eclipse Mosquitto) as the central hub for data acquisition. Devices and services publish messages to MQTT topics, which are then consumed by two exporters:

- **Mosquitto Exporter** (mosquitto-exporter) exposes internal broker metrics such as client connections, topics, and message throughput.
- **MQTT Exporter** (mqtt-exporter) subscribes to all MQTT topics (e.g., data) and translates incoming payloads into Prometheus-compatible metrics for scraping.

This design enables decoupled, real-time metric ingestion via a publish/subscribe model.

REST API Ingestion (FastAPI + Prometheus Client)

The **REST API** (Figure 48) is a custom service built with FastAPI that exposes an endpoint for external systems to push metrics directly into the monitoring system.

- **POST /metrics-data/**:
- Accepts JSON payloads with a metric name and value:

```
{ "metric": "upf_sessions", "value": 123 }
```

  It validates and updates the internal metric state using the Prometheus Python client.

- **GET /metrics**:
  Exposes the current metrics in Prometheus format for scraping.
- **Swagger UI /docs**:
  Provides an interactive interface for exploring and testing the API.

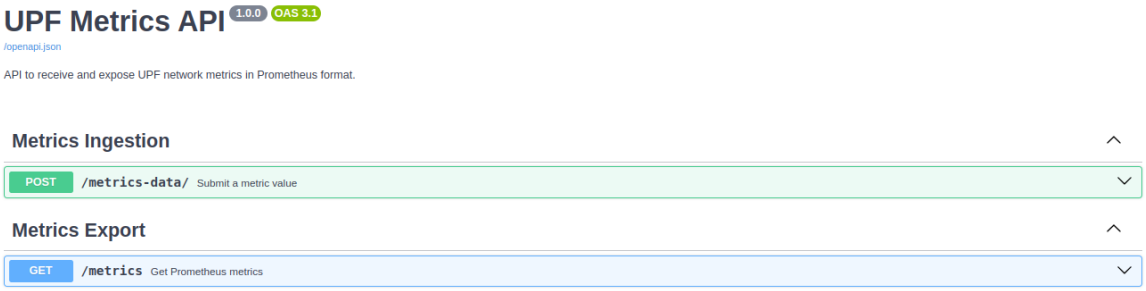

Figure 48 – Swagger FastAPI documentation

This REST API allows systems that cannot be published via MQTT to still participate in the monitoring architecture by sending metrics directly over HTTP.

Here's the active configuration in Table 32.

Table 32 – Scrapped Targets

Target	Port	Exposed Metrics
Prometheus itself	9090	Internal Prometheus metrics
Mosquitto Exporter	9234	MQTT broker metrics (clients, topics, etc.)
MQTT Exporter	9000	Data published to MQTT topics, converted to Prometheus
REST API	8000	UPF metrics via POSTed JSON (FastAPI service)

Prometheus scrapes metrics **every second** (scrape interval: 1s), supporting near-real-time monitoring. Also, more **HTTP endpoints** could be added for the data ingestion if requested.

4.9.3.2 Interface to visualize data

Grafana[34] serves as the visualization layer of the monitoring system, providing interactive dashboards that display real-time metrics, historical trends, and key performance indicators derived from the data ingested by Prometheus. These dashboards enable users to monitor system health, analyse traffic patterns, and identify potential issues through graphical representations. Grafana dashboards translate raw metrics into meaningful visual insights, such as line charts, gauges, tables, and alerts.

An example monitoring dashboard is shown in Figure 49. In this case, the visualization focuses on network metrics, providing panels for packet forwarding and drop rates, as well as session and IP pool utilisation, or system resource usage.

Figure 49 – Example Monitoring Dashboard

The monitoring system includes an initial set of **Prometheus alerting rules** focused on key metrics for early detection of anomalies and critical issues within the 5G network's user plane. These alerts are defined in the *alert.rules* file and are designed to cover potential problems such as high packet

[34] https://grafana.com/

loss, traffic spikes, IP pool exhaustion, and exporter downtime. The current alert rules are: HighPacketDropRate, ForwardTrafficSpike, IPPoolExhaustion.

These alerts provide an initial layer of observability and serve as an early warning system for potential issues with UPFs (Figure 50). However, they are not the only alerts that will be configured. As more metrics are integrated into the system and additional panels are created in Grafana, the set of alert rules will expand to cover a wider range of operational and performance indicators. This will enable more comprehensive monitoring across different layers of the system, including new metrics exposed by the REST API and MQTT data streams or other HTTP Endpoints.

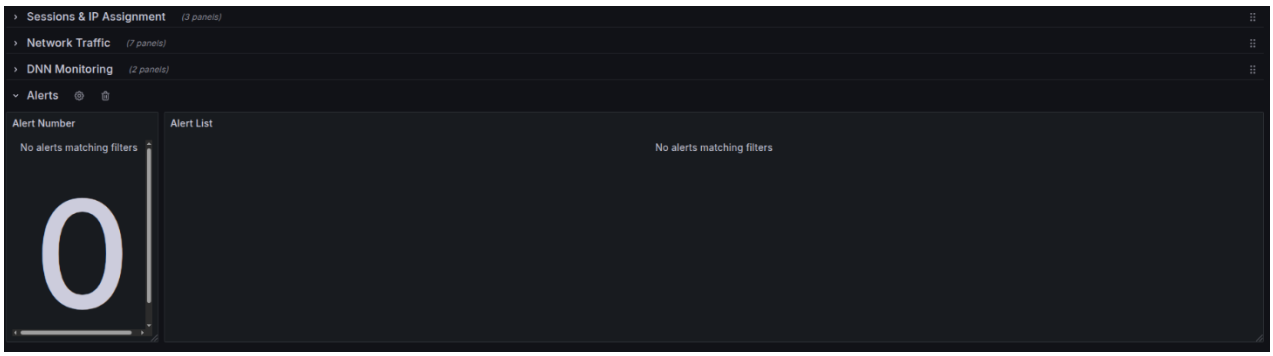

Figure 50 – Alerting Dashboard Panel

4.9.4 Dependencies

This monitoring system is built on a collection of containerized services, tools, and libraries that work together to enable data ingestion, storage, and visualization. The platform is orchestrated using **Docker** and **Docker Compose v3**, ensuring that each service runs in its own isolated container while simplifying deployment, scaling, and maintenance. Configuration files, such as those for Prometheus and Grafana, are mapped into containers through Docker volumes to ensure consistency, version control, and easy updates.

The system relies on the key components introduced in Table 33.

Table 33 – Docker dependencies

Component	Technology / Image	Purpose	Ports / Interfaces
MQTT Broker	eclipse-mosquitto	Message broker for telemetry data. Devices/services publish metrics to topics here.	1884 (MQTT), 9001 (WebSocket)
Mosquitto Exporter	sapcc/mosquitto-exporter	Exposes MQTT broker metrics (clients, topics, throughput) in Prometheus format.	9234 (scraped by Prometheus)
MQTT Exporter	kpetrem/mqtt-exporter	Subscribes to MQTT topics, converts payloads to Prometheus metrics for ingestion.	9000 (scraped by Prometheus)

REST API	Custom FastAPI application	Receives external metrics via JSON (/metrics-data/), exposes metrics in Prometheus format.	8000 (HTTP API + Prometheus scrape endpoint /metrics, Swagger UI /docs)
Prometheus	prom/prometheus	Time-series database for storing metrics. Scrapes metrics from MQTT exporters and REST API.	9090 (Prometheus web UI and API)
Grafana	grafana/grafana	Visualization and dashboarding frontend. Connects to Prometheus as the data source.	3001 (Grafana web UI)

The custom REST API uses several key Python libraries (see Table 34) to support API development, metrics management, and HTTP servings:

Table 34 – Monitoring platform Python dependencies

Library	Purpose
fastapi	Web framework for building the REST API, automatic Swagger documentation.
uvicorn	ASGI server for running the FastAPI application.
prometheus_client	Library for creating and exposing Prometheus-compatible metrics.
pydantic	Data validation and serialization for FastAPI payloads.

5 Conclusions and Next Steps

This deliverable provided the results of the activities of Task 3.1 in which the ENVELOPE platform has been detailed by defining the common technical aspects across the trial sites. The common ENVELOPE platform architecture has been refined compared to the one introduced in D2.2 by providing information about the ENVELOPE Enablers at a finer level of granularity. Furthermore, the realization of the different layers of the ENVELOPE platform has been discussed by introducing the common functionalities and the common denominator of each layer starting from the requirements that have been introduced in WP2.

Further reported results in this deliverable concern the implementation activities of the EaaS module. The functionalities of the EaaS module, the operations, and the internal interfaces have been detailed as well as the implementation details of the different components comprising the EaaS module.

The content of this deliverable serves as a basis for the other implementation activities in WP3 and for achieving an interoperable implementation across the three trial sites. Next steps will be to implement the innovations introduced in this deliverable to make the ENVELOPE Platform ready for experimentation activities.

This deliverable is also useful for disseminating the information about the ENVELOPE platform and the EaaS module external to the project, in particular to external experimenters (e.g., open calls participants) who will test their use cases on the ENVELOPE platform through the EaaS module.