This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No. 101069732

aerOS

EUROPEAN IOT-EDGE-CLOUD

D3.2 – Intermediate distributed compute infrastructure implementation

Deliverable No.	**D3.2**	**Due Date**	**29-FEB-2024**
Type	**Other**	**Dissemination Level**	**Public**
Version	**1.0**	**WP**	**WP3**
Description	*Intermediate implementation of infrastructure components, and update in the implementation.*		

Copyright

The aerOS consortium consists of the following 27 partners:

UNIVERSITAT POLITÈCNICA DE VALÈNCIA	ES
NATIONAL CENTER FOR SCIENTIFIC RESEARCH "DEMOKRITOS"	EL
ASOCIACION DE EMPRESAS TECNOLOGICAS INNOVALIA	ES
TTCONTROL GMBH	AT
TTTECH COMPUTERTECHNIK AG (*third linked party*)	AT
SIEMENS AKTIENGESELLSCHAFT	DE
FIWARE FOUNDATION EV	DE
TELEFONICA INVESTIGACION Y DESARROLLO SA	ES
ORGANISMOS TILEPIKOINONION TIS ELLADOS OTE AE - HELLENIC TELECOMMUNICATIONS ORGANIZATION SA	EL
EIGHT BELLS LTD	CY
INQBIT INNOVATIONS SRL	RO
FOGUS INNOVATIONS & SERVICES P.C.	EL
L.M. ERICSSON LIMITED	IE
SYSTEMS RESEARCH INSTITUTE OF THE POLISH ACADEMY OF SCIENCES IBS PAN	PL
ICTFICIAL OY	FI
INFOLYSIS P.C.	EL
PRODEVELOP SL	ES
EUROGATE CONTAINER TERMINAL LIMASSOL LIMITED	CY
TECHNOLOGIKO PANEPISTIMIO KYPROU	CY
DS TECH SRL	IT
GRUPO S 21SEC GESTION SA	ES
JOHN DEERE GMBH & CO. KG*JD	DE
CLOUDFERRO S.A.	PL
ELECTRUM SP ZOO	PL
POLITECNICO DI MILANO	IT
MADE SCARL	IT
NAVARRA DE SERVICIOS Y TECNOLOGIAS SA	ES
SWITZERLAND INNOVATION PARK BIEL/BIENNE AG	CH

Disclaimer

Authors

Name	Partner	e-mail
Raúl San Julián	P01 UPV	rausanga@upv.es
Rafael Vaño	P01 UPV	ravagar2@upv.es
Salvador Cuñat	P01 UPV	salcuane@upv.es
Dr. Harilaos Koumaras	P02 NCSRD	koumaras@iit.demokritos.gr
Vasilis Pitsilis	P02 NCSRD	vpitsilis@iit.demokritos.gr
George Makropoulos	P02 NCSRD	gmakropoulos@iit.demokritos.gr
Thanos Papakyriakou	P02 NCSRD	thpap@iit.demokritos.gr
Eleni Charou	P02 NCSRD	exarou@iit.demokritos.gr
Andreas Sakellaropoulos	P02 NCSRD	asakellaropoulos@iit.demokritos.gr
Renzo Bazan	P05 SIEMENS	renzo.bazan.ext@siemens.com
Florian Gramß	P05 SIEMENS	florian.gramss@siemens.com
Philippe Buschmann	P05 SIEMENS	philippe.buschmann@siemens.com
Korbinian Pfab	P05 SIEMENS	korbinian.pfab@siemens.com
Ioannis Chouchoulis	P10 IQB	giannis.chouchoulis@inqbit.io
Konstantinos Kefalas	P10 IQB	konstantinos.kefalas@inqbit.io
Ioannis Makropodis	P10 IQB	giannis.makropodis@inqbit.io
Vasiliki Maria Sampazioti	P10 IQB	vasiliki.maria.sampazioti@inqbit.io
Aris Farao	P10 IQB	aris.farao@inqbit.io
Panagiotis Bountakas	P10 IQB	panagiotis.bountakas@inqbit.io
Tarik Taleb	P14 ICTFI	tarik.taleb@ictficial.com
Tarik Zakaria Benmerar	P14 ICTFI	tarik.benmerar@ictficial.com
Amine Taleb	P14 ICTFI	amine.taleb@ictficial.com
Nikolaos Gkatzios	P15 INF	ngkatzios@infolysis.gr
Vaios Koumaras	P15 INF	vkoumaras@infolysis.gr
Eugenia Vergi	P15 INF	evergis@infolysis.gr
Alvaro Martinez Romero	P16 PRO	amromero@prodevelop.es
Eduardo Garro	P16 PRO	egarro@prodevelop.es
Francesco De Angelis	P19 DST	f.deangelis@dstech.it
Riccardo Leoni	P19 DST	r.leoni@dstech.it
Oscar Lopez	P20 S21SEC	olopez@s21sec.com
Ramiro Torres	P20 S21SEC	rtorres@s21sec.com
Jon Egaña	P20 S21SEC	jegana@s21sec.com

History

Date	Version	Change
30-11-2023	0.1	Final ToC
22-12-2023	0.2	Start first round of contributions
28-12-2024	0.5	Merged document with first round of contributions. Start second round of contributions
15-02-2024	0.6	Collection of second and final round of contributions.
16-02-2024	0.7	Merged document with the final round of contributions.
22-02-2024	0.9	Receive comments from IR and start addressing them.
29-02-2024	1.0	Final version.

Key Data

Keywords	Decentralized orchestration, smart networking, security, edge-cloud continuum, self-*, Monitoring, Common API, and Identity and Access Managements, Minimum Valuable Product.
Lead Editor	IQB
Internal Reviewer(s)	Joseph McNamara (LMI) , Álvaro Martinez Romero (PRO)

Executive Summary

The document is contextualized to the works in aerOS' WP3: *aerOS secure, scalable and decentralized compute infrastructure*. The present deliverable is the second and intermediate version of three WP3 deliverables planned for M12, M18 and M30. The deliverable is based on the aerOS module definitions presented in D3.1 (initial distributed compute infrastructure specification and implementation) and D2.6 (aerOS architecture definition) and depicts an advanced version of WP3 activities presenting the relevant components of the aerOS architecture composed from the following tasks:

- T2.5 architecture.
- T3.1: Smart networking for infrastructure element connectivity.
- T3.2: Communication services and APIs.
- T3.3: aerOS service and resource orchestration.
- T3.4: Cybersecurity components.
- T3.5: Node's self-* and monitoring tools.

D3.2 is structured in a manner that will clearly provide for every task the methodological and technological advances that achieved in the context of the aerOS decentralized infrastructure and performed since D3.1.

IMPORTANT: This deliverable is of type OTHER. This means that D3.2 is mostly a software deliverable. While this document reports the advances of tasks T3.1-T3.5 in the period M12-M8, it must be understood together with the software release that is uploaded alongside it.

From a smart-networking perspective, the primary focus is on developing a highly integrated and sophisticated network architecture for aerOS. The achievement of a service mesh in aerOS is realized through a combination of intra- and inter-domain network service mesh strategies. Technologically, this aspect of the project is concentrated on several key areas. The use of eBPF technology, through Cilium, underpins the network's capability to efficiently manage packet routing, security policies, and load balancing. The dynamic adaptation of network services and topologies is facilitated by the interaction with physical or virtual network devices through OpenFlow commands.

In the realm of modern cloud-to-edge computing, communication services and APIs stand as pivotal elements, orchestrating the seamless flow of data and interactions across a multifaceted network of devices and platforms. Through the adoption of RESTful services, HTTP, Apache Kafka, and the integration of Fiware IoT agents, these APIs ensure not only interoperability but also adaptability to the constantly evolving technological landscape.

Regarding the implementation of multi-domain smart service and resource orchestration systems presents a host of challenges but also offers numerous solutions. Building a scalable and robust orchestration system requires a thoughtful architectural approach. The architecture must be designed to handle a high volume of requests, manage resources efficiently, and ensure fault tolerance. In the context of orchestration systems, Kopf plays a vital role as an extensible Kubernetes operator framework. Kafka is particularly relevant in orchestration systems for inter-service communication. For machine-learning operations in high-level orchestration systems, open-source tools like Kubeflow, MLFlow, and Alibi/Seldon are critical. These tools are chosen for their robustness, scalability, and flexibility.

From cybersecurity perspective, for the secure API gateway, KrakenD has been introduced to secure the aerOS APIs (e.g., OrionLD, HLO, etc.) from unauthorized access. KrakenD is integrated with the aerOS Identity and Access Management (IAM) framework, namely Keycloak and provides an extra security layer by validating the deployed Role-based Access Control (RBAC) rules. For the implementation of the aerOS authentication and authorization services, managed by IAM, the OpenID Connect standard has been used as the main technological enabler.

aerOS node self and monitoring tools have been advanced significantly to empower the system with unparalleled autonomy and efficiency. With the implementation of open-source solutions like PowerTOP and psutil, the nodes gain the ability to continuously monitor their health and workload, ensuring optimal performance and

rapid response to potential issues. The json-rules-engine further enhances this capability by enabling sophisticated self-orchestration, allowing the system to dynamically adapt to changing conditions and manage its own workloads with minimal human intervention. Moreover, the integration of KubeEdge into this ecosystem facilitates robust self-healing mechanisms, particularly crucial for edge computing scenarios, thereby bolstering the overall resilience and reliability of the aerOS nodes.

The software implementation of aerOS is on-going where each of the aforementioned parts are progressing with the development of their own components. Additional refinements will be presented in the forthcoming deliverable in D3.3 planned for M30, concluding the activities of all WP3 tasks.

Table of contents

List of tables

List of figures

List of acronyms

Acronym	Explanation
AAA	Authentication, Authorization and Access
ACME	Automatic Certificate Management Environment
AI	Artificial Intelligence
AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machines
ARP	Address Resolution Protocol
BLE	Bluetooth Low Energy
CA	Certificate Authority
CB	Context Broker
CBAC	Context-Based Access Control
CF	CloudFerro
CIC	Constructing Composite Indicator
CIDR	Classless Inter-Domain Routing
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CPU	Central Processing Unit
CR	Custom Resource
CRD	Custom Resource Definition
CRUD	Create, Read, Update and Delete
DDS	Data Distribution Service
DevOps	Development and Operations
DevPrivSecOps	Development, Privacy, Security and Operations
DPM	Data Product Management
eBPF	Berkeley Packet Filter
EP	EndPoint
ETL	Extraction, Transformation and Load
ETSI	European Telecommunications Standards Institute
FaaS	Function-as-a-Service
FE	FrontEnd
FOM	Federated Orchestration Module
FQDN	Fully Qualified Domain Name
GNU	GNU's Not Unix

GW	GateWay
HATEOAS	Hypermedia As The Engine Of Application State
HLO	High-Level Orchestrator
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAM	Identity and Access Management
IdM	Identity Management
ID	IDentifier
IE	Infrastructure Element
IT	Information Technology
IoT	Internet of Things
IP	Internet Protocol
ISRG	Internet Security Research Group
JSON	JavaScript Object Notation
K8s	Kubernetes
LCM	Lightweight Communications and Marshalling
LDAP	Lightweight Directory Access Protocol
LLO	Low-Level Orchestrator
MAC	Media Access Control
MANO	Management and Orchestration
MEC	Mobile Edge Computing
ML	Machine Learning
MQTT	Message Queue Telemetry Transport
MVP	Minimum Viable Product
N/A	Not Available or Not Assessed
NAT	Network Address Translation
NFV	Network Function Virtualization
NGSI-LD	Next Generation Service Interface – Linked Data
NS	Network Service
NSD	Network Service Descriptor
NSM	Network Service Manager
OAS	OpenAPI Specifications
OIDC	OpenID Connect
OMG	Object Management Group
OSM	Open Source MANO
PID	Process IDentifier

QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
RBAC	Role-Based Access Control
REST	REpresentational State Transfer
RFC	Request For Comments
RISC	Reduced Instruction Set Computing
ROS	Robot Operating System
RPC	Remote Procedure Call
SDK	Software Development Kit
SDN	Software-Defined Networking
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol
SPARC	Scalable Processor ARChitecture
SSL	Secure Socket Layer
TLS	Transport Layer Security
TSN	Time-Sensitive Networking
TU	Time Utility
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VIM	Virtual Infrastructure Manager
VNF	Virtual Network Function
VNFD	Virtual Network Function Descriptor
VPN	Virtual Private Network
VPP	Vector Packet Processor
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

1. About this document

Deliverable D3.2 presents a concrete view of the intermediate methodological specification and technological implementation of the components that constitute the aerOS decentralised infrastructure, which is an essential part of the aerOS Meta-OS. It builds up on the candidate technologies that thoroughly described in D3.1 and elaborates on the progress of the composing components and their interactions.

Another deliverable will follow in M30 to finalize the development of the aerOS components their interconections with the activities of other related work, such as WP2 and the final aerOS architecture. Finally, this deliverable is the intermediate blueprint of the aerOS infrastructure and the components that developed in WP3 and will be integrated in aerOS use cases as detailed in WP5.

1.1. Deliverable context

Item	Description
Objectives	O1 (Design, implementation and validation of aerOS for optimal orchestration): Development and intermediate implementation of the components related to aerOS orchestration capabilities. O2 (Intelligent realisation of smart network functions for aerOS): Development and intermediate implementation of the smart-networking components. O3 (Definition and implementation of decentralised security, privacy and trust): Development and intermediate implementation of the aerOS cybersecurity components related to authentication, authorization, and secure access to aerOS APIs. O5 (Specification and implementation of a Data Autonomy strategy for the IoT edge-cloud continuum): Development and intermediate implementation of the NGSI-LD module and its integration with other aerOS communication services and APIs.
Work plan	D3.2 content is based on the definitions and technologies specified in tasks: • T2.1 state of the art. The development of the aerOS components that presented in this deliverable are based on the recorded state of the art. • T2.2 use cases and requirements. The development of the aerOS components that presented in this deliverable consider the requirements for the different use cases. • T2.4 DevPrivSecOps. The development of the aerOS components that presented in this deliverable take into account the DevPrivSecOps methodology. • T2.5 aerOS architecture. The components that developed and presented in D3.2 are defined in the aerOS architecture. The content of D3.2 is the result of the following tasks activities: • T3.1 Smart networking for infrastructure element connectivity. • T3.2 Communication services and APIs. • T3.3 aerOS service and resource orchestration. • T3.4 Cybersecurity components. • T3.5 Node's self-* and monitoring tools. D3.2 presents the intermediate integration of components defined by WP4 tasks within the decentralized infrastructure. The development of the D3.2 components are contributing to WP5 integration and use case deployments tasks.

Milestones	This deliverable is an intermediate base towards the achievement of the milestone MS7 – Final software release planned for M30.
Deliverables	D3.2 is based on the state-of-the-art that defined in D2.1 (State-of-the-Art and market analysis report), the requirements presented in D2.2 (Use cases manual, requirements, legal and regulatory analysis 1), the components that described in D2.6, and the candidate technologies analysed in D3.1 (Initial distributed compute infrastructure specification and implementation).

1.2. The rationale behind the structure

D3.2 details the intermediate development and integration phase of the functional components that performed in the context of the five WP3 tasks and formalize the work package's activities as well as elaborates on the actions that performed in the context of WP3 to finalize the Minimum Viable Product (MVP). Hence, the deliverable unfolds in five sections. Section 1 provides basic information about the deliverable. Section 2 contains a brief introduction of the context and current status of aerOS. Section 3 presents an overview of the MVP, while Section 4 elaborates on the advancements, during M12 and M18, of the five WP3 tasks, detailed in separate subsections that follow the same formal structure. More specifically, the subsections of Section 4 (4.1-4.5) begin with an updated description of the main functionalities. Later it provides the updated structure diagrams along with a description of each component and concludes with the technologies and standards that employed in the MVP. Finally, Section 5 concludes the deliverable and describes the future work expected for the last WP3 deliverable (D3.3).

1.3. Outcomes of the deliverable

This deliverable aims at providing the intermediate version of the aerOS infrastructure components and their progress during the period of M12-M18 along with the work done in WP3 to accomplish the MVP. As in D3.1, the components description will be approached both from methodological and technological point of view, considering the five different domains that each of the WP3's tasks focus.

The aerOS smart-networking represents the functional components responsible for attaining networking efficiency, agility and performance across the aerOS infrastructure elements.

The aerOS communication services and APIs produces the functional components responsible for effortless, efficient, and continuous communication of the aerOS services across the whole IoT edge-cloud continuum.

The aerOS service and resource orchestration develops the functional components aiming to deploy, manage, and federate services responsible for delivering the aerOS functionalities. Moreover, it prepares the functional components essential to properly allocate and evenly deploy various resources to meet the requirements of vertical IoT services employed on top of aerOS.

The aerOS cybersecurity components provides Identity and Access Management (IAM) services focusing on registering and authenticating users in aerOS, managing their access to aerOS elements as well as providing secure access to computerized resources (APIs, infrastructure elements or domains) by linking users' roles and restrictions with registered identities.

The aerOS node's self and monitoring tools develop the functional self-* components to enhance Infrastructure Elements (IEs) employing automated procedures that will minimize the human interaction during all the operations of IEs. To accomplish this, several functional and runtime parameters, such as health and security status, will be provided.

1.4. Version-specific notes

As mentioned above, This deliverable is of type OTHER. This means that D3.2 is mostly a software deliverable. While this document reports the advances of tasks T3.1-T3.5 in the period M12-M8, it must be understood together with the software release that is uploaded alongside it.

In the compressed file that is downloaded when accessing this deliverable, the reader will be able to find two main artefacts: (i) this very document, that reflect in a narrative way the progresses achieved, and (ii) a compressed file that is, in turn, composed of several compressed GitLab repositories corresponding to the code development progress by M18.

In particular, and in order to facilitate the readability of the technical delivery, here below there is an indication of the repositories that have been included in the submission. They are structured following the task reporting that is used in this document (D3.2). This schema is also used in the submitted file. The directories contain the current advances, alongside an explanatory README.MD in each of them in order to describe their purpose and content. Another, equivalent, release will be done by the end of the WP3 (in deliverable D3.3, due in M30).

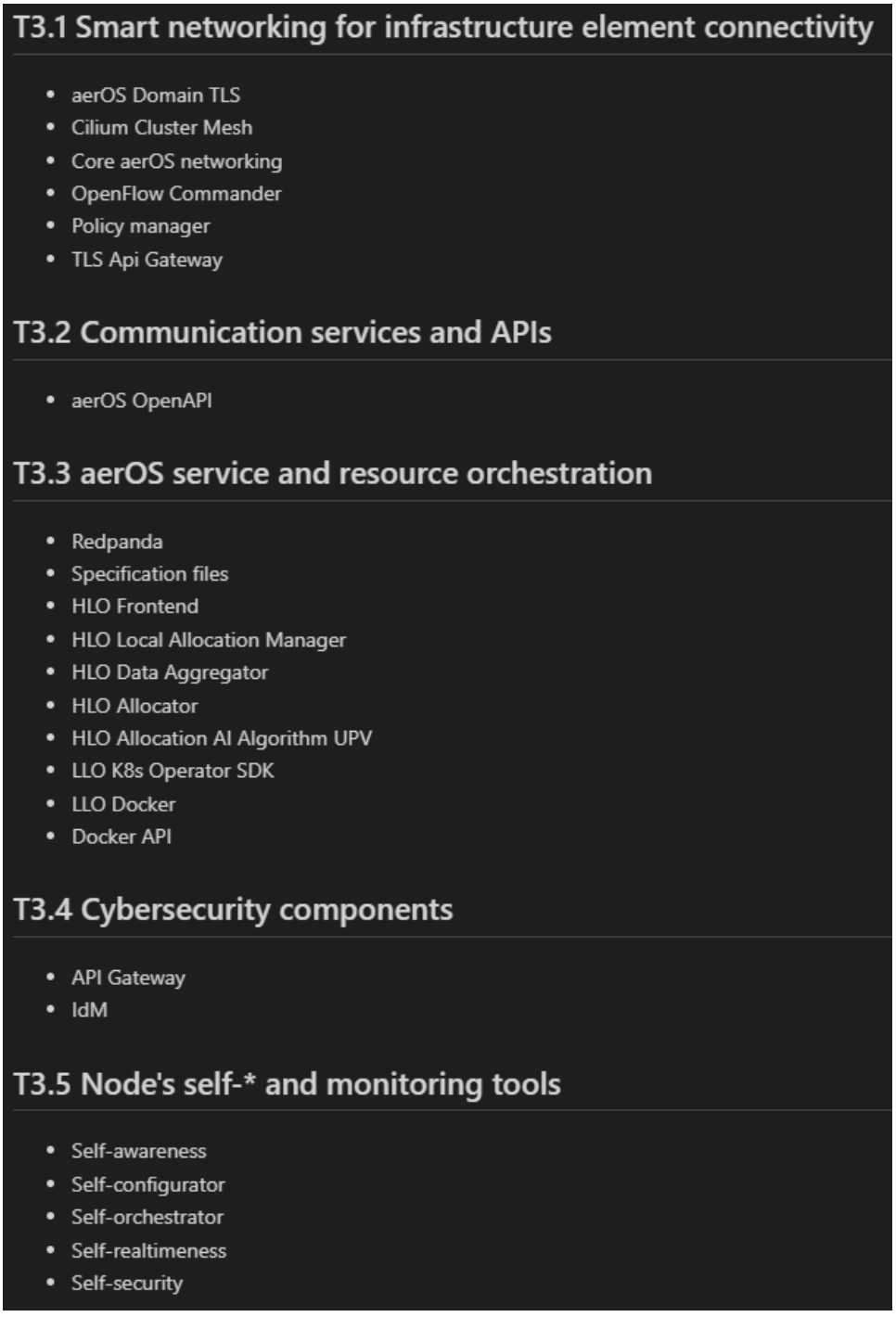


Figure 1. Software release of D3.2

2. Introduction

The aerOS technical work, namely the design, development and implementation of the components that constitute the aerOS architecture is performed in WP3 and WP4, which are the main technical work packages of the project.

The present document (D3.2) provides a thorough analysis of the software components, relationships and building blocks that defined in WP3 and align with the aerOS architecture. D3.2 presents the intermediate state of the components development and clearly communicates the advancements since D3.1 that was focused on the discussion of the candidate technologies to be used in the various WP3 tasks. Moreover, it elaborates on the components of the MVP, which developed in the context of WP3, to present the intermediate status of the aerOS components in a concrete and easy-to-understand manner. Considering the aerOS architecture that discussed in D2.6, the system developed based on the development of several smaller building blocks and fundamental and innovative concepts.

D3.2 is an essential deliverable as it provides an initial view of the development of the aerOS modules that through the MVP outputs the capabilities of the aerOS system in a close to real-life scenario. More specifically, this deliverable discusses the development of the network and compute fabric (T3.1), the service fabric and common aerOS APIs (T3.2), the decentralized orchestration and distributed state repository (T3.3), the authentication, authorization, and access control elements (T3.4), and the self-* and monitoring tools (T3.5).

3. MVP Overview

aerOS project in the first year of its realization has carefully designed an architecture which is intended to provide to IoT developers a coherent environment over which they should be able to take advantage of distributed capabilities all along the continuum and provide to them a common execution environment which can support IoT services deployment and reuse.

With the vision to functionally unify a multitude of diverse computing and network resources from cloud to edge even to IoT devices, aerOS has employed and combined many state-of-the-art concepts and technologies.

In parallel (and after) the design of the architecture, actual advance beyond the state of the art is being achieved till M18 by progressing in the various technical fields of the project. Research and implementation beyond current state of the art in the fields of compute and network fabric, service fabric and data fabric have introduced development and integration complexities, which are reflected to a great variety of technologies and tools.

More specifically, a wide area of technologies in the field of programmable networks for enhanced connectivity, resources and service management and orchestration, resilient and self-adapting runtime layers need to be employed to provide the minimum for the execution environment that aerOS requires. Additionally, Cybersecurity tools and trust management need to ensure private and secure communications and access to services over all the aerOS continuum. All IEs and aerOS domains should seamlessly expose APIs for fully defined communication among components and services. Respectively, data fabric technologies and integrated components should support the transition from heterogeneous IoT data to a unified data fabric, and while monitoring capabilities should extract all information produced and needed for the self-adaptation of the ecosystem, analytics are foreseen to support events recognition and healing processes' triggering. Finally, AI tasks are designed to run over different Infrastructure Elements (IEs) in the continuum with optional use of frugality techniques and inclusion of explainability and interpretability.

Above mentioned technologies represent the primary aerOS technologies and tools needed to realize the continuum and all these are envisioned to be implemented encompassing assimilable cloud native practices to enable stakeholders to design, deploy, and operate scalable and resilient applications over the aerOS Meta-OS. The goal is to encompass cloud-native techniques naturally in continuum deployments, where infrastructure (physical and virtualized) ranges from IoT devices all the way up to cloud data centers (and not only the latter, which is the usual cloud-native case). This fact implies that the great complexity emerging from the research, development and integration of all components suggests an iterative development which should consider and integrate early implementation evaluations, and which should optimize functionalities based on feedback emerging both from development teams and from targeted audience, i.e. IoT developers.

It is worthwhile mentioning that addressing all those complexities and successfully achieving the results cannot be tackled in a single stage. Thus, following the agile schema of the project, it was decided to define a clear strategy that is hereby documented. aerOS team defined a Minimum Viable Product (MVP) that should be completed by M18 to bring together all the aforementioned technologies and tools as an integrated product. In this framework, MVP serves as a tangible aerOS prototype that can be quickly deployed and tested, realizing all the basic functionalities of the continuum. While developing the MVP, aerOS team gained invaluable insights into the feasibility and viability of architecture concepts and it is thus possible to refine intended approach in real-time. Additionally, MVP supports resources' efficient use, optimizing their usage in an effort to stay as light a system as possible without compromising the core essence of its existence. Finally, MVP supports safe guarded mitigation to pilot locations, allowing the team to test aerOS scenaria in a controlled environment without hinging validated core concepts and their efficiency.

The aerOS MVP encompasses the most compelling aerOS functionalities (although more will come in future deliveries) and integrates two aerOS domains, which are deployed in two distinct locations, in geographic and administration terms, so as to demonstrate its functionality over the public cloud. One domain is designed to be the entrypoint domain, while the other represents a plain aerOS domain, which could be deployed anywhere across the continuum. The entrypoint domain is located at the common development and integration infrastructure of the project (a space provided by the partner, cloud provider, CloudFerro), while the plain one resides in the premises of the Technical Coordinator - NCSRD. This diverse topology of the MVP allows the evaluation of aerOS federation mechanisms for expanding in an agile way the aerOS continuum domains with additional/new ones.

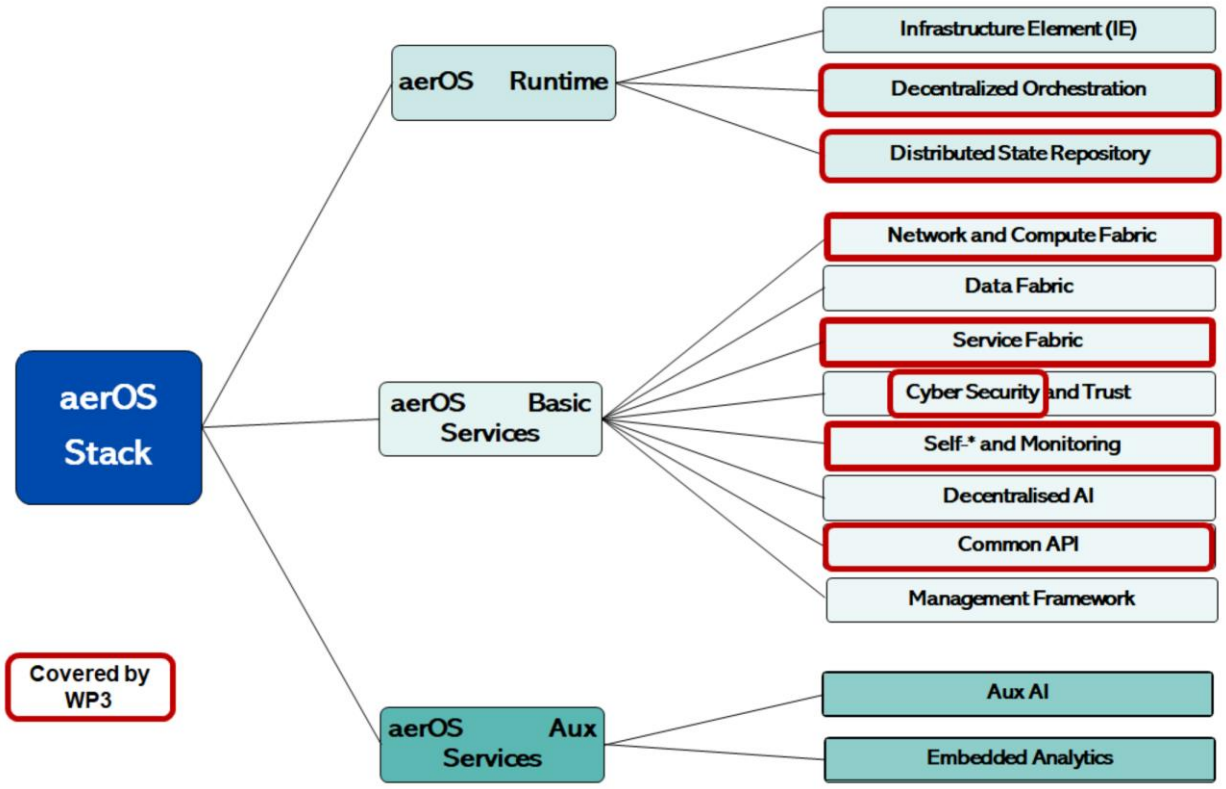


Figure 2. Building Blocks of WP3

MVP builds upon outcomes from both WP3 and WP4 which constitute the two technical work packages of the aerOS project. WP3 works on providing the required infrastructure components, based on the aerOS architecture, needed to enable scalable and secure IoT edge-cloud continuum aiming to support the resources and services orchestration across the continuum. WP3 encompasses several technologies and is related to several components in the aerOS stack. As already presented in D3.1, figure above represents the building blocks which WP3 addresses.

Distributed over 5 tasks, many diverse technologies are addressed within WP3. Each task further breaks down to a set of relevant technologies related to its domain of interest. In order to build an early MVP release, **priority**

has been given to components which are estimated to play a prominent role in establishing aerOS continuum with core functionalities enabled, over components that can be integrated in a second round as they will be based on core aerOS features and which are not crucial to provide a prototype capable to demonstrate overall aerOS functionality.

Having as a target to support, in the grounds of MVP deployment, an early integrated environment which can fully demonstrate federated orchestration -of a variety of services over a multitude of heterogeneous resources-capabilities, WP3 focused on providing the underlying mechanisms which support this process and which can, once validated, be easily integrated in existing isolated computing infrastructure and smoothly turn them into aerOS capable domains, as this will be the case for aerOS pilot sites. Efforts have been directed towards delivering:

- Tools which can set up a minimum aerOS runtime and consequently monitor and extract computing and network resources capabilities and runtime availabilities information, thus exposing the status of all participating domains.
- Mechanisms establishing required connectivity, from edge to cloud and IoT devices, among registered aerOS domains, exposing underlying aerOS services which enable information federation and resources orchestration management.
- APIs design and standardization, both for inter and intra aerOS domain services connectivity, taking into account both synchronous and asynchronous modes. This made it possible to develop services that are in advance equipped with the all the information needed to access or share data and request services either from other domains or internally.
- Components able to orchestrate based on aerOS-modelled data regarding real time aerOS continuum status, and able to access underlying, diverse, computing resources (IEs) and propagate orchestration decisions to them.
- Security and privacy mechanisms which vertically cover all aspects of secure and controlled access to data, exposed services, and resources management capabilities.

In the following paragraphs a summary of the outcomes of each WP3 task, which have been highlighted to be, in priority, delivered and included in MVP realization, and their relevant significance to aerOS continuum establishment, are presented.

In the realm of **networking research**, provision of secure connectivity between aerOS domains was one of the most important concerns with the other being the connectivity of IE within each aerOS domain. The initial focus was on developing programmable mechanisms to expose project domain services through externally accessible endpoints. This step is vital for enabling the interconnection of federation services based on the NGSI-LD standard. Such connectivity lays the groundwork for exchanging data, which model the continuum state, among all aerOS domains. This ensures that every decision engine can “on request” acquire, and thus consider, the current state of all Infrastructure Elements (IEs) from edge to cloud, when it comes to orchestrating services and resources across the continuum. To support cloud-native service deployments, several key features are implemented: MetalLB for load balancing, KrakenD integration with MetalLB for service exposure, enhanced security through TLS and Cert-Manager, and seamless programming of traditional networking functions like 1:1 NAT. These components work together to ensure robust and secure connectivity. On the other hand, equally important was the efficient networking of all computing resources (IEs) within each domain over a framework with monitoring and software-defined capabilities. With the project's federated orchestration layer operating on a Kubernetes framework, Container Network Interface (CNI) capabilities are providing the means to develop and integrate programmability, security, and observability. A topic of research has also been the interconnection of aerOS domains either in a networking level, e.g. VPN, or even in a service level when seamless connectivity, security policy enforcement, and load balancing between services running in different clusters will be needed. The goal is to achieve these in a cloud-native fashion, to enable seamless connectivity, to enforce security policies, and to manage load balancing between services across different clusters, all while adhering to cloud-native principles and practices.

In terms of service intertwining and API establishment the foundational aspects of API development are shaped within the aerOS ecosystem, focusing on API guidelines and best Practices, and API specifications and tooling. As this task has been instrumental in establishing comprehensive guidelines for **API design**, ensuring

consistency, scalability, and security across the system it has greatly contributed to MVP since by embracing industry-standard specifications like OpenAPI, T3.2 has not only streamlined API documentation but also facilitated their integration across diverse tools and platforms. The adoption of code generators under this task has further expedited the development process, enhancing the ecosystem's versatility and interoperability. These efforts collectively form a crucial part of the aerOS infrastructure, setting a robust framework for efficient communication within aerOS MVP.

One of the most relevant innovations of aerOS, and the flagship component starring in the MVP, both high level and low-level **orchestration** layers (which were defined and described in D2.6) have been addressed and included in the M18 delivery. aerOS low level orchestration is designed to be a component-based extendable layer which includes components capable of addressing all existing resources included in an aerOS domain, but not more than these so as to avoid layer overloading. Thus, in the MVP, aerOS team has proceeded in developing the initial LLOs which can orchestrate K8s, containerd and docker based workload resources, based on MVP integrated resources. LLO enforces service deployments to actual resources, but HLO is the component which supports all the process of receiving IoT service deployment requests (and required topologies), decoding the requirements, selecting the most suitable IEs among all IE operating across aerOS continuum, and finally addressing the proper LLO which is responsible for the most suitable IEs. Within the orchestration system, the HLO is further broken down into several components handling different aspects of HLO work including API models transformation, data fabric access to retrieve candidate IE across continuum, smart and AI decision engine to inference most suitable IEs, deployment capabilities to address and provide suitable deployment templates to LLOs. These components have been developed to fully support HLO integration in MVP.

Secure and controlled access to resources is critical and thus most of the processes related to aerOS **Cybersecurity** are included in the MVP so as to demonstrate their integration with the whole aerOS Meta-OS. As a result, authentication, authorization, and access control capabilities of aerOS are deployed and presented in the MVP, showcasing that users with different access rights can be effectively managed by aerOS Identity and Access Management (IAM) system using the Role-based Access Control (RBAC) mechanism and KrakenD secure gateway. The combination of previous mentioned tools makes it possible to block or allow access to aerOS APIs (e.g. ngsi-ld endpoints). Additionally, IAM enables authorized access to aerOS Management Portal so as to support restricted access to different Management Portal domains based on different user roles and groups. Thus, access to registered resources and functionalities, within management portal, are controlled and permitted only to authorized users. While previous mentioned features are related with aerOS runtime environment security, a basic toolset which supports secure components development and integration is built around GitLab platform providing pipelines for secure code development and continuous integration. Although this is not directly integrated in the MVP it provides facilities for securely developing and deploying aerOS components for the MVP.

Also, in the MVP, the most relevant self-* modules (those that live and act within the scope of a single IE) have been addressed, developed, integrated, and tested. Some self-modules were casted more prioritized than others, being the former absolutely crucial for supporting federated orchestration process. **Self-awareness** component has been developed and is included in MVP as it automates the process of publishing IE capabilities and updating, on real time, running IE availability. This information is modelled using aerOS data model for the continuum (WP4), and IE status is propagated all across the continuum consisting thus a candidate for IoT service components deployment. Additionally **self-orchestration** component has also been developed as is the first chain in the loop of announcing resources alerts and thus trigger, when and as needed, service mitigation. Self-realtimeness (the one covering the cases where real-time sensitive containers come into play) and self-security (protecting and mitigating intrusion events into the IE) have been developed and included in the MVP of M18, redounding then in a robust result that can now be deployed in aerOS-compliant continuums.

The above-mentioned outcomes are varied and originate from distinct domains of expertise. It is the integration of these diverse components within the MVP which reflects the project's progress and clarifies identification of tasks that need to be addressed next. The MVP's role within the project lifecycle is dynamic, adjusting according to timeline requirements, achieved milestones, and the objectives targeted at each phase.

In the initial, design, phase it served as a guide to the prioritization of developments by supporting the definition of what is the minimal set of features that will make aerOS viable to its first set of, internal, users and capable for an initial demonstration of its visions and functionalities.

Accordingly, MVP is the field of validating architectural concepts and evaluating components viability and synergy. aerOS development team consists of many technical partners working, each one of them, on defined assignments with the responsibility to deliver components that should be integrated and work seamlessly one with the other. Although this development is based on specifications, development contracts, APIs and data model definitions it would not be possible to verify the interaction and the interworking of these components as an integrated system without a deployment environment, an environment where all things come together.

MVP development, and its current progress status, is what makes feasible to plan for the demonstrator that will be showcased in the mid-term review. aerOS team has exhaustingly worked on designing the architecture, providing blueprints that guide the development of all components, their functionalities, and interactions. The MVP has been specifically designed as the platform which enables the timely and efficient demonstration of architectural concepts and aerOS functionalities. Scenarios planed for the demonstrator showcase are based on a realistic environment that is underpinned by the MVP implementation.

Accordingly, MVP will support an iterative development phase. Vertical stakeholders, partners of aerOS project pilots, will provide their feedback which will lead development towards enhancing existing features, fixing issues, and, perhaps, introducing new functionality guided by user demand. Additionally, aerOS open calls will contribute components which will be validated in the MVP.

At the final stage MVP will act as a guide for deploying aerOS stack to project pilot use cases. The validation of core functionalities stability will pave the way for replicating aerOS deployment in the 5 use case pilot locations. Not all of them have the exact same needs and they do not have to deploy the full stack, just these services that make them aerOS compatible and anything more suitable for their vertical domain purposes. MVP facilitates the understanding of all functionalities provided and enables the selection of the ones that provide what is needed for each one of them.

As a last word it should be mentioned that MVP development success is directly connected with the use of the aerOS DevOps platform. aerOS development lifecycle management is based on an, in premises (UPV hosted), GitLab platform deployment (https://gitlab.aeros-project.eu). aerOS Gitlab does not only foster a unified development environment but also ensures that every iteration, enhancement, and refinement made to the MVP is systematically documented and version controlled. aerOS Gitlab enables workflows streamline, from coding and testing to deployment, allowing for real-time tracking of the MVP's evolution. Internally, the structure of aerOS GitLab groups, subgroups and projects is in accordance with the tasks and research domains of development work packages (WP3 and WP4). Each group is built around a research domain concept, so for WP3 there are the Network, APIs, Orchestration, Cybersecurity and Self-* groups and within each group a project for every component that must be developed. For example, in Orchestration group there are projects which reflect all components that should be developed to support HLO and LLO and messaging among these components; so internally to this group there are the HLO Data Aggregator, HLO Local Allocation Manager, various LLOs, Specification files for internal messaging data modeling and even more projects. All projects are documented with content that can support both development and deployment process. This documentation supports the transfer of all aerOS development to the MVP. Permissions to these groups and projects are configured based on the internal assignments. Beyond the source code versioning workspace, aerOS GitLab also hosts the project image and package repository where development teams push their final products which are ready to be deployed on MVP and pilots and which in the future will be used for deploying aerOS to candidate computing resources. Thus, it is obvious that the whole development process and accordingly the MVP setup is robustly supported with DevOps platform integration and processes enforcement and that there is a strong correspondence of what is deployed in MVP and what is versioned in GitLab; each GitLab, tagged, version corresponds to an MVP version. As a result, software accompanying this deliverable, exported directly from GitLab, can be demonstrated in aerOS MVP.

4. Intermediate Implementation

4.1. Advancements in Smart networking for Infrastructure Element connectivity

4.1.1. Updated description and main functionalities

aerOS is designed, and MVP is structured accordingly, in such a way that the continuum is a network of aerOS domains, where each one of them provides the same capabilities and is itself a network of IEs. The design avoids components which need to be centrally controlled so that any domain can be self-contained and can also be easily integrated as a peer in the continuum. This is reflected in networking functionalities. All network capabilities are built with the goal to support self-contained functionality and at the same time to flexibly adapt to wide area connectivity requirements once a domain joins aerOS continuum. Towards the goal of establishing network and compute fabric, WP3 provides connectivity for IEs so they can register themselves as part of the continuum and undertake specific workloads execution, during IoT services deployment. As described in D2.6, IEs publish their capabilities and offer their computational resources via their integration in administrative domains, sharing a common set of core functionalities, called aerOS domains. The capability developed to network IEs within each domain and the capability to abstract their cross-domain connectivity, under aerOS domains networking and connectivity, is the required underlay for the continuum establishment. Secure and control-enabled establishment of these networking capabilities is absolutely required, especially when running over public networks. These, along with the network functions needed to provide secure exposure, balancing and access to federated orchestration services of each domain, were the most crucial goal of networking related development towards MVP. The solutions chosen and developed do not just target network connectivity; they pave the way towards integrating technologies which can provide dynamic adaptation of network parameters, programmability of network functionalities, performance monitoring, and they build on tools and technologies which provide separation of control and data planes. Moving a little higher, in terms of abstraction, workloads running over physical or virtual resources, i.e. IEs, may stay agnostic of the already virtual networking of IEs and still have their own connectivity requirements regarding policies, security, load balancing or other aspects. The research and development undertaken this period aim to create a foundational layer of connectivity for IEs, which is essential for expanding aerOS across diverse resources, from cloud to edge environments. This foundational layer incorporates abstractions and automated processes typical of SDN, facilitating the seamless integration of resources.

D3.1 outlines that task efforts are organized across seven key research lines:

- Smart networking within the K8s context
- Intra-domain network service mesh
- Inter-domain network service mesh
- Integration of Network Service Mesh with Service Mesh
- Synergy between Network Service Mesh and SDN
- Combining Networks Service Mesh and NFV
- TSN Support for the aerOS continuum

The primary objective to develop MVP, as an early prototype, makes it possible to assess developed networking functionalities and identify additional requirements. To achieve this, development efforts have been concentrated on tools and technologies that intersect several of the research lines mentioned above. It is important to note that not all research lines have received equal attention in this phase. In the next paragraphs progress carried out so far regarding networking tools and main functionalities and topologies for the MVP are presented.

Network Functions for aerOS services exposure and cross-domain connectivity

aerOS core services, within each domain, on top of which the data, the service and the network and compute fabric across the continuum are established, must be securely exposed, accessed, and inter-connected among domains.

Each aerOS domain is a federation of IEs which, mostly, live inside private networks either in self-hosted premises or in public providers. Furthermore, within each aerOS domain a set of core services are deployed which are not accessible from outside the domain, due to the fact they are deployed within K8s runtime framework. This runtime provides a virtual networking overlay for services and workloads to communicate but can expose these to the outer world. Thus, virtual network functions able to serve as a unified ingress gateway, providing a single, external though in private IP space, endpoint to securely expose selected services for external access were developed. This exposed IP is the connected endpoint where a public IP, of the hosting premises, is forwarded (1-1 NAT). As a result, aerOS services are accessible in public and connectivity through a single and controlled endpoint. Components involved to provide this functionality are Ingress controller, Load Balancer, APIs, TLS capabilities. Further a 1:1 NAT with the premises edge networking infrastructure is employed to expose the discoverable access endpoint for each aerOS domain.

Nginx Ingress controller is the unique access point for all incoming traffic towards aerOS domain. Beyond acting as a reverse proxy, it also undertakes endpoints' encryption (TLS), subsequently it directs all incoming traffic to API gateway. **KrakenD** serves as the **aerOS API Gateway** providing path (or verb) based routing of external REST API requests to underlying aerOS services. The APIs set to be externally routed include NGSI-LD for the data fabric and HLO endpoints for IoT deployment and generally life cycle management requests. KrakenD also integrates seamless authentication and authorization bit this is discussed in section 4.4. To facilitate load balancing and enforce IP advertisement, **MetalLB** is integrated as a cloud-native network function. MetalLB leverages L2/L3 layer address discovery protocols to advertise a range of IP addresses within the local network by responding to ARP requests for advertised address and answering with IE actual IP address. It is dynamically configured to promote an IP address from the private network's range. This IP is then targeted by a 1:1 NAT process, allowing external traffic to be directed to the project's selected services through a public IP address. The 1:1 NAT setup ensures a unique public IP address is directly linked to the corresponding internal IP address advertised by MetalLB and ingested by Ingress controller (and VPN server). This arrangement guarantees redirection of traffic arriving at specified public IP ports to the private aerOS domain IP and the exposed ports. A cloud native deployment of **WireGuard** server, exposed with the support of MetalLB on port 51820/UDP, provides the functionality to establish secure encrypted tunnels between aerOS domains facilitating communication at both IEs and workloads level if service components might need such a provision.

Figure 2 in the diagrams and components section below offers a detailed view of this topology. To enforce security of data exchange among project domains, TLS encryption alongside HTTPS protocol is strictly enforced, ensuring the safeguarding of the aerOS domain exposed services against data breaches and unauthorized access while preserving communication confidentiality. A detailed diagram and description is provided in Figure 3. This approach ensures that the project's exposed services are secure, effectively mitigating the risk of data leaks or unauthorized exposure and maintaining the confidentiality of communication. Cert-Manager tool has been integrated for this purpose and the selected CA to issue certificates is Let's Encrypt. ACME protocol HTTP-01 challenge is implemented to ensure domain ownership. TLS certificates are issued for the MVP domains based on the FQDN that have been provided for them under the aeros-project.eu domain name. In the case of absence of domain name security can be established using self-signed or domain-signed TLS certificates, in order to keep a self-contained trust model. These certificates are either generated with cryptographic suite tools or obtained via the Kubernetes Certificates API. In any case certificates are kept as secrets in the aerOS domain runtime and are seamlessly incorporated into Ingress controller and used for endpoints' TLS termination.

aerOS programable networking

IEs are built on top of physical or virtual processing nodes and although these nodes should already be networked in a private network infrastructure, services that run on top of them to provide the aerOS network and computing fabric and the workloads that are developed by IoT developers should communicate seamlessly as if they were on the same physical network, regardless of the underlying network topology, utilizing virtual overlay networks on top of existing network infrastructure. The technology employed to offer networking capabilities is **Cilium**, well known to provide and enhance networking and security for container-based systems.

Cilium is built on top of **eBPF** technology which allows to dynamically insert and update networking and security functionality without changing the kernel code enabling thus efficiently handling for packet routing, security policies, load balancing, and more. Programmability of network policies, based on Cilium CNI implemented capabilities, is being developed which will offer the possibility to modify (allow or deny) egress and ingress traffic from and towards specific workloads. The objective is to develop strategies that aim to control communication between workloads.

Although the aerOS networking capabilities are purposed to be self-contained, there are cases where an interaction with external infrastructure or network services is required. An aerOS component capable of interacting with either OpenFlow-enabled networking devices (virtual or physical) which offer suitable APIs or with SDN controllers API is being developed for the MVP. This component will be able to send OpenFlow commands that can configure and dynamically adapt the behaviour of physical or virtual network services and consequently network topology. As an example, and as we already mentioned, when an aerOS domain is deployed, a 1:1 NAT between the Private IP address, used by KrakenD to expose aerOS services, and the public IP at the premise's edge router, must be realized to make the aerOS domain visible in the wide and enable its connectivity across the aerOS continuum. This procedure involves a procedure which can be automated if the edge router is OpenFlow-enabled. An application under development, as a K8s pod, supports this automation, through generation and transmission of OpenFlow commands to the edge router. Containerized, application generates the appropriate OpenFlow commands which will be afterwards sent to the edge router, instructing the creation of the 1:1 NAT rule.

Network mesh across aerOS Domains

For the case of advanced networking capabilities, as the seamless connectivity between workloads across aerOS domains the advanced Cilium feature, **Cilium Cluster Mesh** is integrated. By enabling Cilium Cluster Mesh, two or more aerOS domains can interconnect, sharing services, security policies, and networking capabilities as though they were part of a unified domain. To enable network mesh, based on Cilium Cluster Mesh, there are considerations to be foreseen and respected. Unique cluster id, name and CIDRs, for the pod and service subnets, for each domain are needed and advanced shared key management for authorization, and for ensuring trust and secure data exchange. Cilium ports and K8s API should be accessed, and this might include configuring network routes, VPNs, or other networking solutions.

4.1.2. Updated Structure diagram

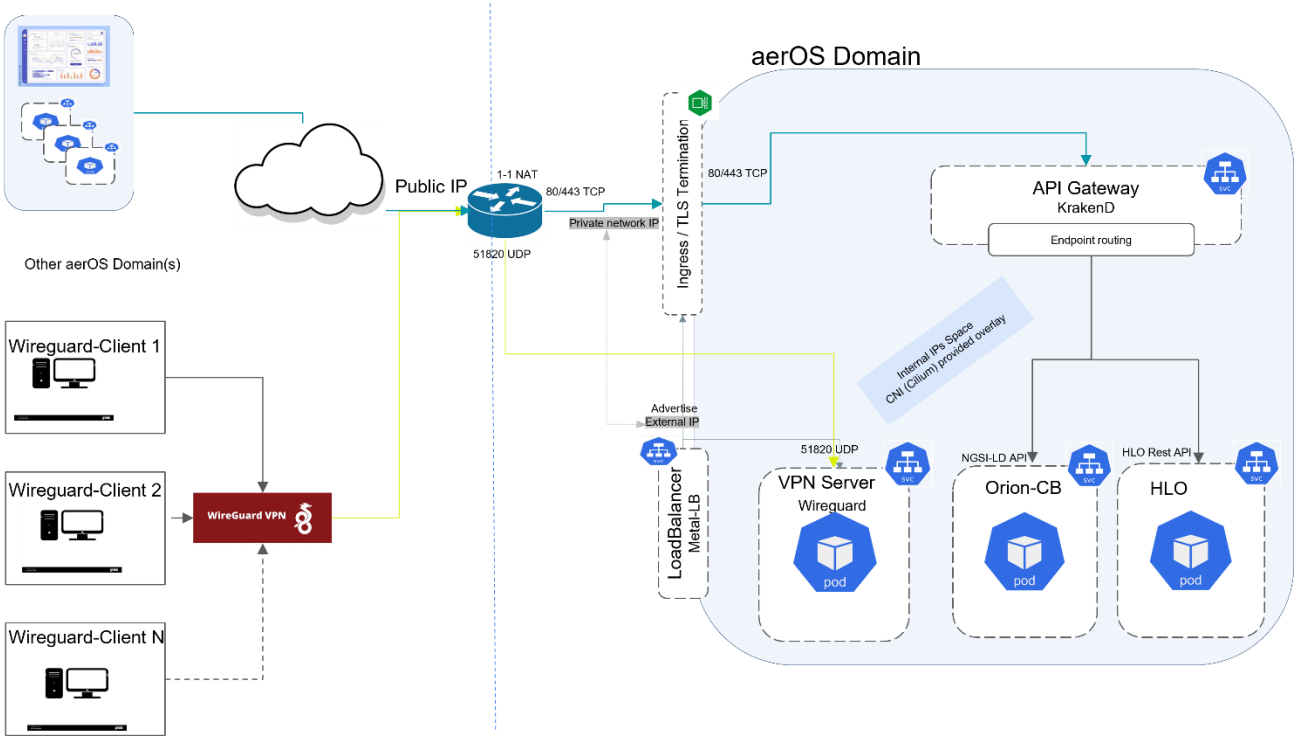


Figure 3. aerOS domain external connectivity

Table 1. Components description, aerOS cross-domain connectivity

Component	Description	Interactions
Load Balancer	Assigns external IP address to services of type LoadBalancer, to be accessible from outside the aerOS domain. Responds to ARP requests (L2), for the service IP address, with the MAC address of a selected IE, effectively making that IE the gateway for traffic to the service.	Advertises, external to domain, address which is bound to **VPN server** for UDP VPN connection requests and to **Ingress Controller** for all other traffic towards aerOS domain services.
Ingress / TLS Termination	Acts as a reverse proxy, providing an entry point for external traffic to the aerOS domain. Terminates SSL/TLS connections, handling the encryption and decryption of traffic between external word and aerOS domain.	Premise's **edge router** (not in aerOS domain), forwards traffic from public IP to aerOS domain private network IP (ingress listens to it). **API gateway** to which it routes external traffic to be routed to appropriate endpoints.
API Gateway	Directs incoming API requests to the correct microservices based on the path, method, and other attributes of the request. Integrates with aerOS IAM and validates credentials and ensures that only authorized agents or users can access specific endpoints.	**Ingress** from which it receives all external traffic. Underlying **aerOS domain services** to which it directs incoming requests based on the incoming path or method. Such services are data fabric **NGSI-LD, HLO REST API** and this list can be expanded.
Endpoint routing	Configuration object of the API Gateway defining the translation and redirection of external requests to underlying aerOS services.	**API Gateway** which is the subject of the configuration object, and which undertakes their enforcement.
ORION-CB / HLO	aerOS core services, implementing REST API exposure. These services run within each domain, implementing federation and orchestration capabilities, and thus making each aerOS domain part of the continuum.	**API GW** is directing external traffic to each of these based on the exact path pf the incoming request or the http method.
Edge Router (1-1 NAT)	Premise's edge router configured to forward (1-1NAT) traffic from arriving public IP to aerOS domain private network IP which is intercepted by Ingress. Public IP address is mapped directly to private network IP address on a one-to-one basis. If device is OpenFlow enabled, it might be programmed from within aerOS	On the one edge component is connected to the **public internet** on the other is connected to private network where aerOS domain IEs are also connected. The aerOS domain **Ingress Controller** is the sole aerOS component acting as a receiving point for all forwarded traffic from edge router.

	domain from networking service.	
VPN Server	Acts as VPN server and implements the WireGuard protocol providing the capability to implement secure tunnels among aerOS domains.	Premise's **edge router** (not in aerOS domain), forwards traffic from public IP, destinated to UDP port 51820 traffic.

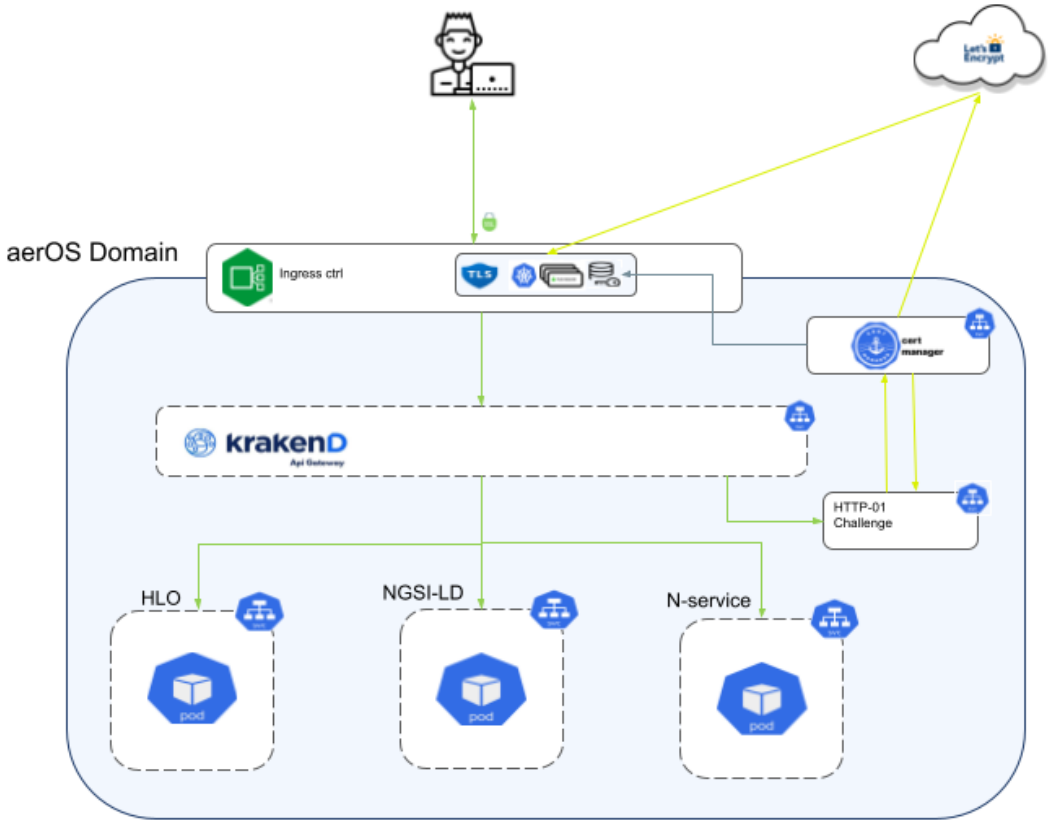


Figure 4. aerOS secure connectivity

Table 2. Components description, aerOS domain TLS

Component	Description	Interactions
Ingress Controller	Terminates SSL/TLS connections, handling the encryption and decryption of traffic between external word and aerOS domain. Acts as a reverse proxy, providing an entry point for external traffic to the aerOS domain.	**API gateway** to which it routes external traffic to be routed to appropriate endpoints. **External agents,** whose requests to access aerOS domain services are routed via Ingress Controller. (Intermediate **edge router** and 1-1 NAT process have been skipped for brevity as they are described above). **Cert-Manager** which provides the secret object which contains the issued certificates.
Certificate Manager	Automate the management, issuance, and renewal of TLS certificates for aerOS domain securing thus communication for services. Integrates with	**Let's Encrypt CA** server, which issues the certificate. **Ingress Controller** which "mounts" the issued certificate.

	certificate authorities (CAs) or self-signed certificates. For MVP Let's Encrypt integration is configured, employing ACME protocol. Creates a temporary Ingress resource, or modifies an existing one, and routes traffic for a specific verification path to a solver pod (nginx). Stores issued certificate to an aerOS secret which is subsequently used by Ingress controller to secure endpoints.	Solver pod (**nginx**) to which an ingress resource is created to verify the challenge (ACME HTTP-01 challenge)
Cloud Let's Encrypt	Cloud service accessible over the internet that automates the process of issuing, renewing, and revoking digital certificates.	**Cert-manager** accesses cloud service to request a certificate. Solver pod (**nginx**) to which an ingress resource has been created and cloud service verifies domain ownership. (ACME HTTP-01 challenge) **Cert-manager,** which retrieves issued certificate and provides it as a secret to aerOS environment.
API Gateway	Directs incoming API requests to the correct microservices based on the path, method, and other attributes of the request. Integrates with aerOS IAM and validates credentials and ensures that only authorized agents or users can access specific endpoints.	**Ingress** from which it receives all external traffic. Underlying **aerOS domain services** to which it directs incoming requests based on the incoming path or method. Such services are data fabric **NGSI-LD, HLO REST API** and this list can be expanded.
ORION-CB / HLO	aerOS core services, implementing REST API exposure. These services run within each domain, implementing federation and orchestration capabilities, and thus making each aerOS domain part of the continuum.	**API GW** is directing external traffic to each of these based on the exact path pf the incoming request or the http method.

4.1.3. Technologies and standards deployed in MVP

Table 3. Technologies and standards for aerOS networking implementation

Technology/Standard	Description	Justification
MetalLB	A load balancer implementation for bare metal K8s clusters. Provides a network load-balancing solution that does not rely on external hardware or cloud	Operates in Layer 2 mode, using standard Ethernet protocols to create a network bridge between the physical network and selected K8s services. Thus, responds to ARP requests for the service IP, effectively

	providers' load balancers.	making the service IP reachable, outside the K8s cluster, within the local network. This enables to bind a private network IP address to aerOS domain Ingress (and VPN).
Nginx Ingress Controller	A reverse proxy which provides unique point of access to aerOS domain and directs external traffic to the underlying configured services.	Acts as sole entrypoint for the aerOS domain and as a TLS termination point which handles the encryption and decryption of HTTPS traffic towards backend services, simplifying thus certificate management and enhancing security.
Cert-manager	Cloud-native tool which can manage and issue TLS certificates. Obtains, renews and stores SSL/TLS certificates from various issuing sources (CAs).	Automates the process of managing certificates for the TLS termination of exposed REST API endpoints. Integrates easily with aerOS components with the capability to handle and provide both self-signed and Let's Encrypt certificates.
Let's Encrypt	Open Certificate Authority (CA) provided by the Internet Security Research Group (ISRG).	Automates and simplifies the process of obtaining, installing, and renewing SSL/TLS certificates. It is open and free.
KrakenD	API gateway which can act as a single point router on external requests towards underlying services. It provides L7 routing towards underlying services based on the content of the messages, such as URL paths, headers, and other request attributes. Additionally, among others, it offers response aggregation, and authN/Z aggregation.	The reasons behind the KrakenD integration were to route incoming traffic requests to different backend services based on the URL path, making available thus a conformant exposure layer, and the seamless integration of authentication and authorisation to underlying exposed services..
WireGuard	High-performance VPN protocol that aims to be faster, simpler, and more secure than existing VPN protocols such as IPSec and OpenVPN	Offer the possibility to connect to aerOS domain securely over the internet.
Cilium (eBPF)	Cilium, as an open-source, cloud-native solution, is employed for providing, securing, and observing network connectivity between workloads. It leverages eBPF to provide and enhance a wide range of networking, security, and observability features in cloud-native environments. eBPF allows for the dynamic injection of bytecode into the kernel at runtime enabling user defined code to run in kernel space and thus program response	It facilitates advanced networking features, including basic and advanced virtualized networking overlays, within aerOS domains. Its ability to leverage eBPF technology sets it apart from traditional networking solutions, allowing it to operate at the kernel level with high efficiency and flexibility. Beyond connectivity for workloads, it provides observability, monitoring, network policies enforcement and can be extended to support multi cluster, cross-domain, networking. It provides extended CNI programmability, expected to be especially useful for policies

	to events such as packet arriving.	definition and network monitoring within aerOS domains.
Cilium cluster mesh	An advanced feature of Cilium that extends its networking, security, and observability capabilities across multiple Kubernetes clusters.	This feature is integrated to enable seamless, secure, and efficient communication between services deployed in different aerOS domains, located in different administrative domains. The goal is to turn separate domains as one from the perspective of network connectivity and security policies.
OpenFlow	Protocol used in software-defined networking (SDN) that allows the path of network packets through the network switches to be determined by software running on a controller.	OpenFlow protocol is integrated to enable aerOS domain interaction with external networking equipment that provide programmability capabilities and thus make feasible a partial control of surrounding networking environment to the benefit of aerOS domain.

4.2. Advancements in communication services and APIs

4.2.1. Updated description and main functionalities

In the dynamic and interconnected world of cloud-to-edge computing, the role of communication services and APIs has become increasingly pivot al. These services and APIs are the linchpins in the aerOS ecosystem, enabling standardized, secure, and efficient interactions among various software entities. At their core, APIs act as facilitators, exchanging information and commands while adhering to predefined protocols and formats.

The primary objectives guiding this task within the realm of aerOS include:

- **Standardized Communication Protocols**: Establishing a common language for interaction, pivotal for the MVP and subsequent releases in the aerOS ecosystem. This encompasses adopting RESTful services over HTTP and Apache Kafka, and ensuring these protocols meet the evolving needs of aerOS services.
- **Interoperability**: This goal strives to enable seamless interaction across different systems, devices, and technologies. Initiatives include the use of code generators and data translators, ensuring smooth communication between diverse technologies and adherence to NGSI-LD standards.
- **Automation and Integration**: A focus on streamlining processes and enhancing the synergy between various components of the aerOS ecosystem. This involves the integration of different IoT devices and the employment of integration services for efficient communication.
- **Security and Access Control**: Ensuring robust security measures like authentication and authorization are in place.
- **Testing and Maintenance**: Although not a primary focus for the MVP, these aspects are vital for the long-term reliability and efficiency of aerOS services. Regular testing and maintenance are essential for ensuring the APIs' continued correct, secure, and efficient functionality.

The upcoming sub-chapters will delve deeper into these objectives, outlining the progress in implementing pre-selected candidate technologies for the aerOS MVP. They include:

1. **API concepts, guidelines and best practices (4.2.1.1)**: Discussing the standards and practices essential for robust API development and maintenance.

2. **API specifications and tooling (4.2.1.2)**: Covering the tools and specifications such as OpenAPI, code generators, and AsyncAPI, crucial for effective API development.
3. **Low code tools (4.2.1.3)**: Investigating the potential of tools like NodeRED and Behavior trees in enhancing the aerOS ecosystem's functionality.

Each of these areas represents a cornerstone in the development and optimization of communication services and APIs, ensuring that aerOS remains at the forefront of technological innovation and efficiency.

4.2.1.1. APIs specifications and tooling

When talking about REST APIs, there is not standardized procedures to create endpoints (a.k.a resources), encode body payloads or define which are the return codes for either successful or erroneous invocations.

Some standardization organizations tried producing such a standard, but normally there is no a "fit-all" solution and they end-up providing use-case/domain specific guidelines. For example, ETSI produced the "Mobile Edge Computing (MEC); general principles for mobile edge services APIs"[1]. Those are the principles used when building both NGSI and NGSI-LD APIs for Orion-LD Context Broker.

Similar guidelines can be found almost in each cloud provider and across the internet. A single search in "medium.com", (an aggregator of blogs and articles published in different technical sites), will provide hundreds of articles for "REST API best practices". Most of them pivot around the same ideas but have slightly different recommendations on how to model behaviours (e.g., interpretation of HTTP verbs, return HTTP codes, etc.).

The best practices that can be extracted from these articles are sketched below:

URI design

As a rule of thumb, endpoints are expressed as a valid URI (following the IETF RFC 3986[2]) and modelled as a combination of a verb and an object. Verbs correspond to the five HTTP methods used to represent CRUD (Create, Read, Update, Delete) operations: GET for read, POST for create, PUT for full updates, PATCH for partial updates, and DELETE for delete.

The object part of the URI must be a noun representing a "collection of objects". Therefore, the practice is to use a plural name in URIs.

For example, if we have some entities representing a broker in modelled in our system, the URL to access that collection would take the form of:

VERB + /brokers

The URI can be extended to point to a single element in the collection by adding a nested level to the url with a token that uniquely identifies the element. For example, if each broker is identified using a numeric index, the URL would look like:

/brokers/<id>

When collections are chainable, it is not a good practice to replicate the structure mentioned above, as it is unclear how to interpret the query and has a poor scalability, since multiple endpoints are needed for different nesting. In this case, the better option is to use query strings for all levels except the first one.

/brokers/<id>?property1=val1&property2=2 … &propertyN=valN

Pagination

Like the filtering concept introduced in the previous section, access to resources may produce large datasets that will require large bandwidth and increase the latency of the communication. In such case, pagination should be used using query string params. The most common technique for pagination is "offset pagination" in which the

[1] https://www.etsi.org/deliver/etsi_gs/MEC/001_099/009/01.01.01_60/gs_MEC009v010101p.pdf
[2] https://datatracker.ietf.org/doc/html/rfc3986

client provides an offset and a limit that the server must consider when querying for entities. There are two conventions to support this strategy:

- Using “limit” and offset: GET /<endpoint>? offset=40& limit=10
- Using “page” and “page_size”: GET /<endpoint>?page=4&page_size=10

Versioning

When multiple versions of the API need to coexist, client must be able to invoke operations on a specific version. There are several techniques to achieve this coexistence: embed the version into the URI, sending a query property with the version or setting a request header with the version.

The last two approaches reuse the URI and delegates the logic of providing version specific contents to the server hosting the API. The URI approach provides different URIs for the same resource when invoking different API versions, so a client using a specific version will not need to add additional properties/headers to each request.

The recommended URI structure for deploying versionable APIs is:

{apiRoot}/{apiName}/{apiVersion}/{apiSpecificSuffixes}

Were “apiSpecificSuffixes” are the resources discussed in the previous section.

Status codes

For every request from a client to the API, the server must respond and provide a return status code. The convention is to use HTTP status codes[3]. These codes are grouped in 5 different ranges, covering the majority of possible situations:

- 1xx: relevant information (not required in APIs)
- 2xx: operation successful
    - 200 OK: response to a successful GET/PUT/PATCH request.
    - 201 Created: response to a successful POST request.
    - 204 No Content: response to a DELETE request.
    - 202 Accepted: indicates that the server received the request but did not process it yet. It is used in asynchronous operations to indicate the operation will be executed in the future.
- 3xx: redirection
    - 301 permanent redirect: indicates that the resource URL has change and the provided one in the response should be used in the future.
    - 302 temporary redirect: a new URL should be used on this request, but in the future the original URL should be used.
    - 303 See other: indicates that the server is redirecting the user agent to a different resource, as indicated by a URI in the Location header field, which is intended to provide an indirect response to the original request. A user agent can perform a retrieval request targeting that URI.
- 4xx: client error
    - 400 Bad request: the request in not understood by the server. Normally this situation happens when the request contains data that is not expected by the server, or the data is expressed in a format not expected.
    - 401 Unauthorized: the client did not provide authentication details to access the resource.
    - 403 Forbidden: the user provided authentication details but is not authorized to access the resource.
    - 404 Not Found: the resource does not exist, or it is unavailable.
    - 405 Method not allowed: the server does not allow the use of the verb in that resource.
    - 406 Not acceptable: the server cannot provide any of the content formats supported by the client.

[3] https://datatracker.ietf.org/doc/html/rfc7231

  - 410 Gone: the resource has been moved from this URI or it is no longer available.
  - 412 Precondition failed: the request has preconditions, like checking an ETag to avoid write conflicts, that were not met.
  - 415 Unsupported Media type: the requested return format is not supported.
  - 422 Unprocessable entity: the attachment uploaded by the client failed to be processed by the server.
  - 429 Too many requests: the server exceeded the max number of concurrent requests and cannot process this one.
- 5xx: server error
  - 500 Internal Server Error: the request is valid, but an unexpected error happened in the server.
  - 503: Service Unavailable: the API is not reachable.

Since each code has a known explanation, clients can determine the outcome of an invocation by checking the status code.

Error details

When a 4xx/5xx status code is returned, the client already has a good overview of what has happened. Therefore, the importance of correctly modelling response codes in the design of an API.

It is good practice to also include in the response additional information describing what happened, providing context on the status code returned. Again, there is no standardized way to do it, but most recommendations suggest using an object containing at least one property "error" which provides an API specific error code. Additionally, other fields such as "description" providing a human-readable string of the error (normally for logging), "parameters" to indicate which specific part of the request was incorrect, can be appended. Nevertheless, the structure of such object can be shaped on each API, but all errors in the same API should be coherent and use the same semantics.

If the API can provide multiple errors (e.g., server-side validation of a form submission), the returned structure can be extended to an array. The resulting structure could be (using JSON media content, but it can be expressed using other media types such as XML):

```
[
  {
    "error": 1,
    "description": "Error 1 in the API",
    "parameters": [
      "param1"
    ]
  },
…
  {
    "error": 34,
    "description": "Another API specific error"
  }
]
```

It is responsibility of the API designers to document all the possible error scenarios so consumers can also implement the logic to correctly handling errors.

Asynchronous operations

Sometimes requests with a POST, PUT, PATCH or DELETE might require processing that takes some time to complete. If the server keeps the request waiting until the operation is completed, it may lead to unacceptable latency.

In this scenario is good practice to immediately return a "202 Accepted" status code together with a status endpoint to check for the status of the asynchronous operation. The process is sketched in the next figure:

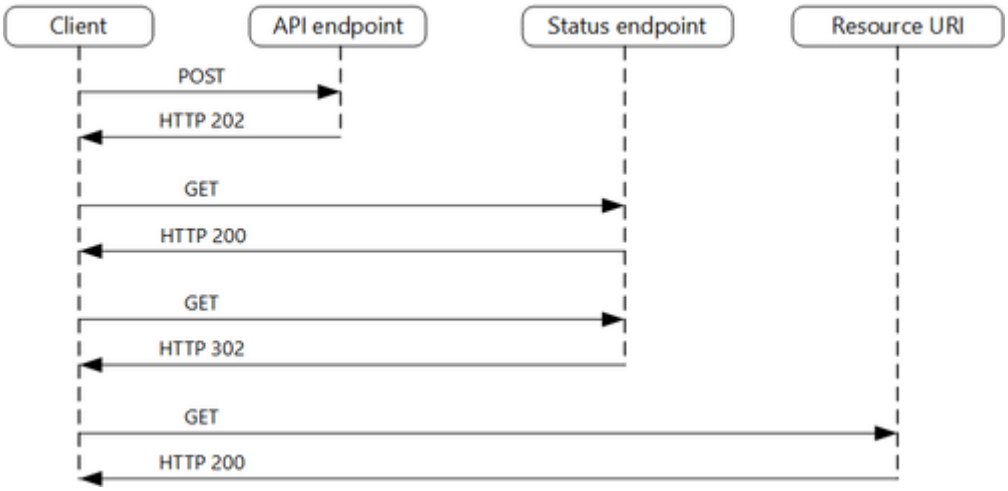


Figure 5. Asynchronous REST operations management

The most common way to return the status endpoint is via the "Location" header. It is also common to include the "Retry-After" header to indicate the client an estimation of how much time the operation may take and prevent continuous polling that may overwhelm the server.

Hypermedia contents or HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) allows the use of hypermedia links in the response contents. These links allow the client to navigate to the appropriate resource by traversing hypermedia links and without further knowledge on how the API is designed.

The main idea is that the hypermedia contents returned from the server drive the application's state and not the other way around. This technique prevents the client from hardcoding all URIs and allows the API to evolve without breaking the client logic.

RFC 5988[4] introduces the concept of media linking and the different techniques that can be used to implement it.

4.2.1.1.1. API maturity

In the world of web services and API design, achieving a high level of maturity and compliance with RESTful principles is often a goal for many developers and architects. The Richardson Maturity Model (conceived by Leonard Richardson) provides a structured way to measure the maturity and adherence of web services to REST principles[5].

The Richardson Maturity Model is a framework for evaluating the maturity of web services in terms of their adherence to RESTful principles. Leonard Richardson analyzed a hundred different web service designs and divided these designs into four categories. These categories are based on how much the web services are REST

[4] https://datatracker.ietf.org/doc/html/rfc5988
[5] https://restfulapi.net/rest-architectural-constraints/

compliant. The model uses 3 elements to assess the maturity of a service: URI design, use of HTTP verbs and implementation of hypermedia links (HATEOAS)[6].

As a result, the model defines levels 0 to 3:

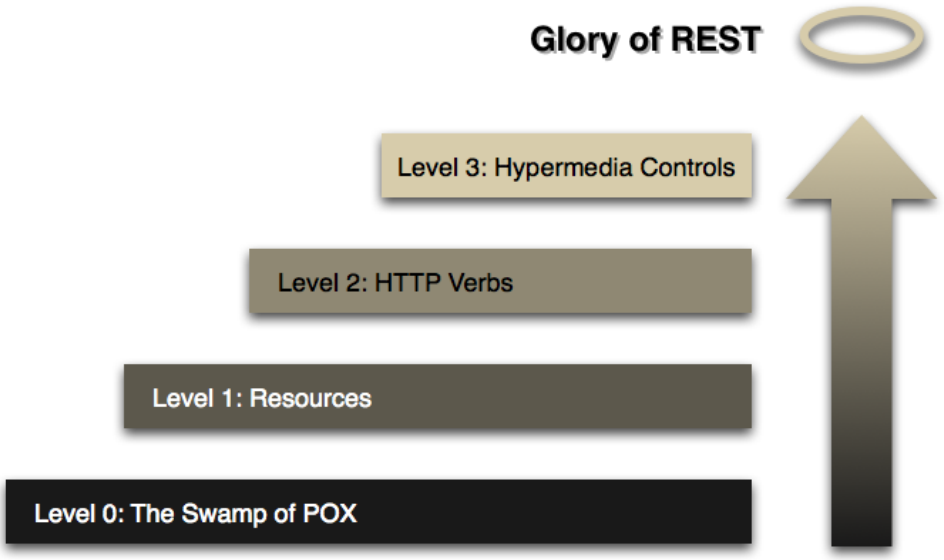


Figure 6. Richardson's maturity model

- Level 0: the most basic approach for implementing a service-oriented application. It does not use the concept of URIs, verbs or HATEOAS, it mainly uses post requests to a single URI. For example, SOAP Webservices and XML-RPC APIs can be categorized as level 0.
- Level 1: it introduces the concept of URI to segregate the access to different resources, but it does not exploit the potential of HTTP verbs and hypermedia links.
- Level 2: this level provides a significant maturity in a restful API implementation, as it uses HTTP verbs and URIs, but it does not implement hypermedia.
- Level 3: APIs at this level embrace all the three aspects. At this level, services prioritize discoverability and self-descriptiveness by using HATEOAS.

Richardson Maturity Model serves as a reference for assessing and improving the design of RESTful web services. It highlights the importance of URI design, HTTP methods, and HATEOAS in achieving different levels of RESTful maturity. By understanding and applying these principles, developers, and architects can create REST web services that are not only well-structured but also highly efficient and accessible.

4.2.1.2. APIs specifications and tooling

In the pursuit of robust and interoperable communication across diverse computing resources within aerOS, the adoption of clear and standardized API specifications emerges as a cornerstone. API specifications play a pivotal role in ensuring clarity, consistency, and ease of implementation, providing a common language for developers and stakeholders involved in the aerOS ecosystem.

Our commitment to fostering a standardized approach led us to embrace OpenAPI[7]. OpenAPI is a specification for HTTP APIs that defines the structure and syntax in a technology agnostic way. These specifications are typically formalized using YAML or JSON, allowing for their easy sharing and consumption.

OpenAPI Specifications (OAS) can be integrated in the API lifecycle: from requirements elicitation, to design, development, testing and deployment.

We can distinguish two API design methodologies:

[6] https://restfulapi.net/hateoas/
[7] https://www.openapis.org

- API First: you start creating the OAS and then you create the code using the OAS.
- Code First: you write you code and annotate it to automatically generate the OAS.

The suggested methodology by OpenAPI is “API First”, which is summarized in the next figure:

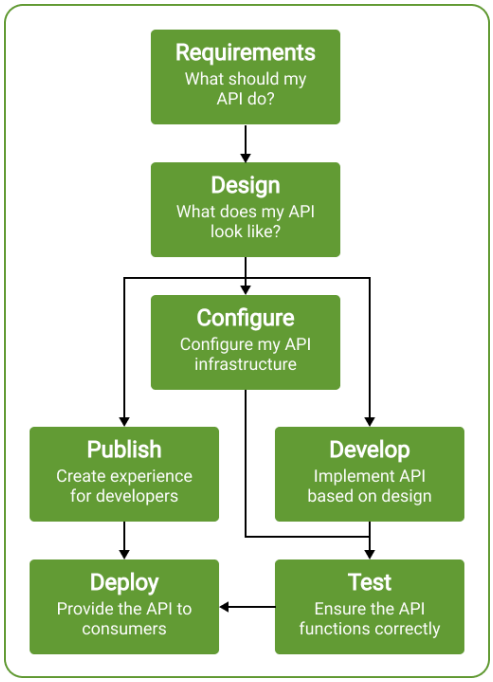


Figure 7. Integration of OpenAPI into the API lifecycle (source https://www.openapis.org/what-is-openapi)

Each of these phases are described as:

- **Requirements elicitation**: in this phase it is described what the API should do and what functionality the API should provide to its consumers.
- **Design**: in this phase, a first OAS document can be sketched to turn requirements into something tangible. These specifications can incorporate design patterns and standardized schema objects based on industry standards (e.g., known ontologies and vocabularies). When this OAS document is properly source controlled, can be used a trusted input for the development phase.
- **Configuration**: this phase configures the infrastructure with the OAS. It relates to how your API should be deployed and managed inside your IT infrastructure. For example, if the API requires a gateway, if the API requires external security, etc.
- **Publishing**: in this phase we can create documentation from the OAS. There are several tools that automatize this documentation generation, creating, e.g., a static html site (e.g., SwaggerUI[8]), that can be published to a basic html server.
- **Development**: in this phase, an OAS is converted into an actual API implementation using specific technologies. There are tools for almost every programming language that will create the skeleton for an API provider/consumer based on the OAS.
- **Testing**: having an OAS provides a definition of what the API is supposed to offer and can be used as a baseline for tooling checking that the contract in designs matches the implemented API or security checks to find weaknesses in the implementation.
- **Deployment**: deployment is a combination of the artifacts from publish and development phases, and it implies that a tested API is made available to the final consumers.

[8] https://swagger.io/tools/swagger-ui/

The website https://openapi.tools/ contains a curated catalogue of tools that can be used to support each lifecycle phase in a great variety of technologies and programming languages and supporting both design approaches: API First and Code First.

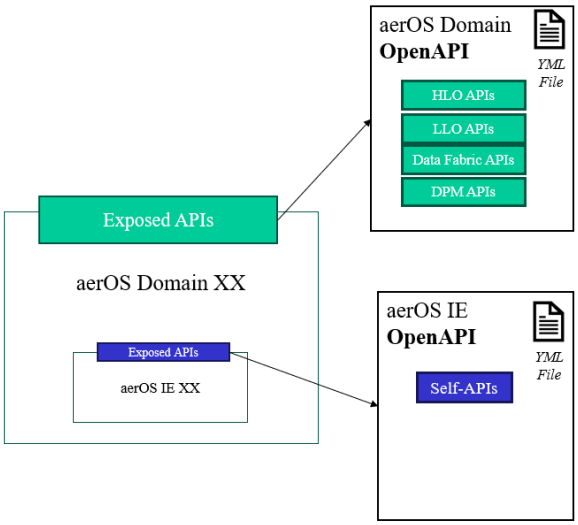


Figure 8. OpenAPIs for the MVP

Over the past months, this task has focused on the development and preparation of OpenAPIs for both the aerOS domain and the Infrastructure Elements (IEs). The aerOS will expose four distinct types of APIs in the MVP: High-Level Orchestrator (HLO) APIs, Low-Level Orchestrators (LLO) APIs, Data Fabric APIs, and Data Product Management (DPM) APIs. Each of these APIs serves a unique function within the aerOS ecosystem.

The HLO APIs are designed to facilitate complex orchestration tasks at a high level of abstraction, allowing for robust interaction and management across various aerOS services. In contrast, the LLO APIs provide granular control over specific functionalities, enabling precise manipulation of the underlying systems. The Data Fabric APIs are integral to the efficient handling and integration of data across the aerOS platform, ensuring seamless data flow and accessibility. Meanwhile, the DPM APIs are essential for managing the lifecycle of data products, underpinning the platform's data governance and utilization strategies. In parallel, the IEs have been equipped with self-* capabilities APIs, which are crucial for the self-reporting and autonomous operation of the infrastructure components. The collaborative effort across different partners in various tasks has been pivotal in delivering these APIs, with each partner bringing specialized expertise to ensure a robust and scalable MVP.

4.2.1.2.1. Leveraging OpenAPI for code generation

In exploring the capabilities of code generation within our aerOS ecosystem, we employed the following methodologies:

- Utilization of OpenAPI's code generator integrated in Swagger 2.0.
- Implementation of openapi-generator version 7.1.0, executed locally on a machine.

Both approaches produce similar results with minor to no changes, however we chose to proceed with the latter due to its broader applicability. The application includes:

- **Client SDKs**: These are the generated scripts that can send requests and receive responds from the API.
- **Server stubs**: In this case the API's functionalities described in the YAML file are integrated to the code for a web application.

In order to reach a solid conclusion on the abilities of the code generator, we instructed it to produce code on some of the most popular programming languages, specifically:

For client code:

- C++ (using cpp-qt-client)

- Python
- Ruby
- JavaScript

For server code

- Python-Flask
- C++ (using cpp-qt-qhttpengine-server)

Results and observations

Based on the experiments we conducted, it was determined that OpenAPI is indeed a fast and direct way to develop and deploy client and server applications for any API. Therefore, proving that it can be utilized in a multidomain environment, where multiple connections between components need to be established rapidly. However, we need to raise certain concerns regarding the server code generation. For example, while evaluating the server code in Python-Flask, we noticed that some of the generated code was relying on older libraries that no longer support some of the requested modules and functionalities. The client code on the other hand can send requests to the API and receiving the expected data. The C++ results appeared to be even less reliable, since both the server and the client applications proved unable to send requests or receive answers from the API. Although the code can be modified and the general structure of the application exists, the aforementioned limitations reduce the ability of OpenAPI to be adopted as a reliable tool to generate code that can connect and establish reliable and efficient data sharing between the aerOS components.

4.2.1.2.1. AsyncAPI for event driven communication

OpenAPI is a widely adopted industry standard in software engineering, playing a pivotal role in defining standardized specifications for REST-based interfaces. However, as technology evolves, there is a growing need for standardized specifications of asynchronous interfaces, a capability OpenAPI does not inherently provide. The rise of asynchronous interfaces and protocols is driven by the desire to move away from monolithic systems towards more distributed architectures, often termed "event-driven" or "reactive." This shift aims to enhance system efficiency, scalability, and fault tolerance.

To address the limitations of OpenAPI in the asynchronous realm, the AsyncAPI initiative has emerged, seeking to establish an industrial standard for specifying asynchronous interfaces. Unlike OpenAPI, AsyncAPI goes beyond by offering comprehensive support for various communication protocols such as MQTT and Kafka. This flexibility allows it to accommodate the diverse requirements of distributed systems.

4.2.1.2.2. AsyncAPI for DDS and Zenoh

Currently, AsyncAPI stands as a valuable tool for specifying asynchronous interfaces, but there is potential for further enhancement. Integrating additional industry-used standards like the Data Distribution Service (DDS) and OPC-UA or the publish/subscribe protocol Zenoh into the AsyncAPI framework would broaden its applicability and contribute to its widespread adoption, even for embedded systems and the edge. This expansion could catalyze the development of solutions tailored to the industrial context, particularly in the realm of automated interface integration. Such advancements would streamline the development of applications within industrial automation systems, fostering innovation and efficiency in this critical domain.

DDS is an open standard for real-time, scalable, and interoperable data distribution middleware. Developed by the Object Management Group (OMG), DDS is designed to facilitate seamless communication and data exchange in distributed systems that demand real-time capabilities. There are several implementations of DDS, such as OpenDDS or Cyclone DDS. Similar as DDS, Zenoh is a decentralized publish/subscribe data distribution middleware however it offers more capabilities for edge-focused applications and IoT environment by a unified API for data storage and querying.

The requirements for integrating any protocol or standard into the AsyncAPI framework are twofold. Firstly, it requires mapping primitives of the protocol specification, e.g., DDS topics, DDS subscribers, and DDS publishers, to AsyncAPI concepts, such as channels, consumers, and producers. And, secondly, protocol-specific features that do not map immediately to a AsyncAPI concept require the definition of a *binding* allowing to provide protocol-specific information related to *servers*, *messages*, *channels*, and *operations*.

We briefly recap the core concepts of AsyncAPI:

- **Server:** A server represents a message broker or a messaging system that facilitates the exchange of messages between producers and consumers. It is the infrastructure that handles the routing and delivery of messages.
- **Producer:** A producer is a component or application that publishes messages upon a state change, e.g., a button click in a graphical user interface or a sensor value change, to channels.
- **Consumer:** A consumer is a component or application that receives messages and reacts upon events, commands, or queries.
- **Channel:** A channel is a named communication pathway within a server that serves as the destination and source of messages or events.
- **Message:** A message is the unit of data exchanged between producers and consumers through a server. Messages follow a well-defined schema and fall in one of three classes: an event, a query, or a command.

When considering the AsyncAPI, DDS, and Zenoh specifications side by side, it becomes apparent that many core concepts of AsyncAPI map directly to primitives in the DDS specification and also Zenoh's publish/subscribe scheme, i.e., AsyncAPI producers map to data writers in DDS or publishers in Zenoh, AsyncAPI consumers map to data readers in DDS or subscribers in Zenoh, and AsyncAPI channels map to DDS topics or Zenoh keys. However, there are core concepts in AsyncAPI that do not have an obvious counterpart in the DDS or Zenoh specification that require more investigation to be mapped.

We decided to use Zenoh as a case study to demonstrate how to map a new publish/subscribe protocol to AsyncAPI since the mapping of Zenoh's peer-to-peer publish/subscribe communication to AsyncAPI reproduces the steps needed to do the same for DDS. Zenoh has gain a lot of traction recently and was even selected as an alternative middleware to DDS in ROS. Moreover, we explore the feasibility of mapping Zenoh's data storage and query functionality to AsyncAPI.

4.2.1.3. Low code tools

The integration of low-code tools into the aerOS project represents a significant step towards democratizing the development process and enhancing the system's flexibility. At the heart of this integration lies the implementation of behaviour trees, a graphical low-code application that do not directly orchestrate services within the aerOS domains—that role is specifically reserved for the High-Level Orchestrator (HLO) and Low-Level Orchestrator (LLO)- but instead, the behaviour trees function as a graphical low-code interface that triggers functionalities within already running applications with different parameters.

The behaviour tree, as a low-code application, enables users to define triggers that activate specific services' functionalities, without initiating or terminating the services themselves. It is an approach that ensures a user-friendly method for modifying the operational logic, where users can interactively change the services to be triggered and adjust their parameters with ease.

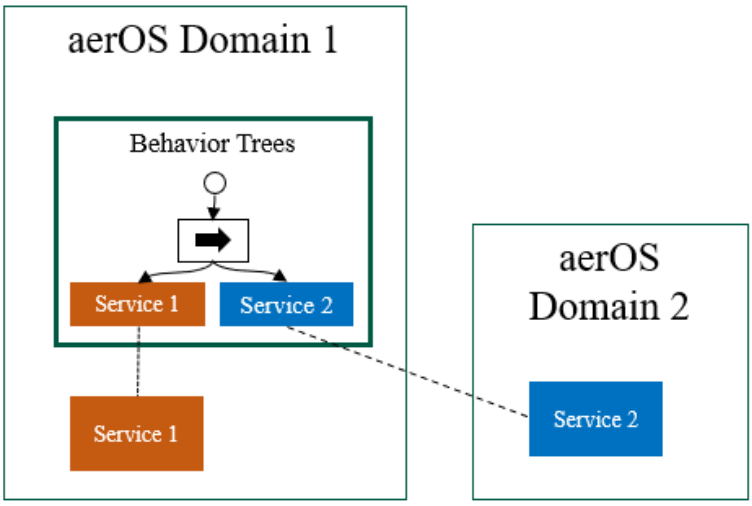


Figure 9. Behaviour trees in aerOS

In this illustrative example, we observe a nuanced application of behaviour trees within the aerOS framework, which showcases their role in triggering functionalities across different aerOS domains. Within aerOS domain 1, the behaviour tree is strategically configured to initiate specific functionalities of service 1. This graphical representation within the behaviour tree does not start or end the service but rather triggers predefined operations of the already active service 1. Simultaneously, the behaviour tree extends its utility beyond the confines of its native domain. It is adeptly set up to also trigger functionalities in service 2, which, although depicted within aerOS domain 1, can be executed in an external aerOS domain 2. This cross-domain interaction is a testament to the interoperable design of the aerOS system and the flexible nature of the behaviour trees. The behaviour tree thus serves as a pivotal interface for system administrators or users, who can manipulate the triggers without delving into the complexities of the services themselves. It is a practical embodiment of low-code principles, allowing for swift and intuitive modifications to the system's behaviour.

Furthermore, both the HLO and LLO could designed to be aware of the behaviour tree's schema, which informs them of the services that need to be prepared and ready at the system's startup. This pre-emptive knowledge allows for a more efficient and responsive system where services can be quickly mobilized in response to the triggers defined by the behaviour tree. The incorporation of behaviour trees thus represents a nuanced enhancement of the aerOS system's responsiveness and adaptability, providing users with a powerful tool to influence the system behaviour dynamically while leaving the core orchestration responsibilities to the HLO and LLO.

4.2.2. Updated Structure diagram

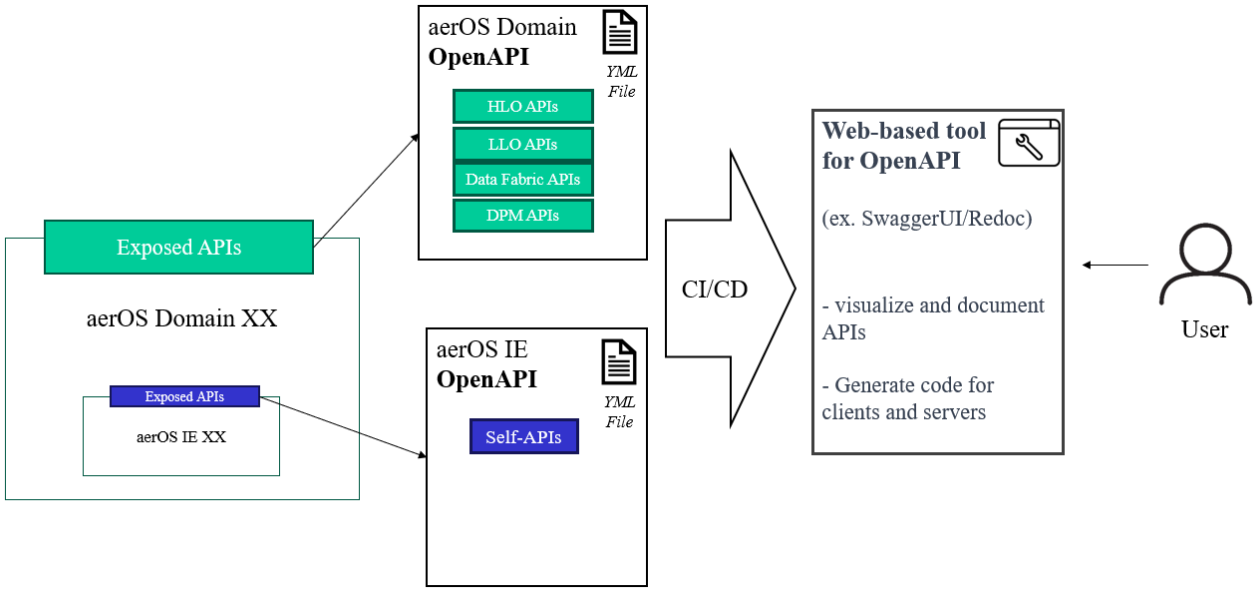


Figure 10. aerOS OpenAPI structure

Table 4. Proposed solution for aerOS with OpenAPI

Component	Description	Interactions
aerOS Domain	Represents a domain within the aerOS continuum that exposes its own set of APIs.	Interacts with the aerOS Domain OpenAPI for API documentation and definition.
aerOS IE	aerOS Infrastructural Element	Interacts with the aerOS IE OpenAPI.
aerOS Domain OpenAPI	The OpenAPI document for the aerOS Domain, detailing the available RESTful APIs.	Contains HLO APIs, LLO APIs, Data Fabric APIs, DPM APIs; used by the web-based tool for visualization.
HLO APIs	High-Level Orchestrator APIs	Interact with entry point Domain, data fabric and LLO.
LLO APIs	Low-Level Orchestrator APIs	Utilized for precise operations with containers in an aerOS Domain. Interactions with HLO.
Data Fabric APIs	Data fabric APIs based on the NGSI-LD standard	Enable seamless data flow, integration, and accessibility across the aerOS ecosystem.
DPM APIs	Data Product Management APIs	Support data governance and utilization strategies within the aerOS domain.
Self-APIs	Self-capabilities APIs	Used by the aerOS IE for self-management, and may interact with monitoring and governance tools.
Web-based tool for OpenAPI	Tool (e.g., SwaggerUI/Redoc) that visualizes and documents the APIs described in the OpenAPI	Interacts with both aerOS Domain and IE OpenAPI documents; serves the User.

	documents.	
User	The end-user or developer who interacts with the API through the web-based tool.	Uses the web-based tool to visualize, document, and generate code for APIs.

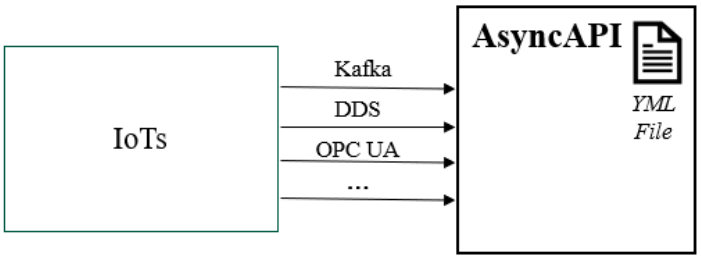


Figure 11. AsyncAPI

Table 5. Proposed solution for aerOS with AsyncAPI

Component	Description	Interactions
IoTs	Internet of Things devices that are part of the aerOS ecosystem.	Interact with other IoT devices
AsyncAPI	A specification for documenting event-driven API architectures.	Describes the interaction between IoTs and different protocols; used for asynchronous communication.
Kafka/DDS/OPC UA	Communication protocols used by the IoTs for messaging and interaction.	These protocols enable IoTs to send data to the services described by the AsyncAPI YAML file.

4.2.3. Technologies and standards deployed in MVP

Table 6. Technologies and standards deployed in MVP

Technology/Standard	Description	Justification
OpenAPI	A specification for building and documenting RESTful APIs.	Enables clear, standard-based documentation, simplifies API development, and increases interoperability.
Web-based tool for OpenAPI (e.g., SwaggerUI/Redoc)	Tools that provide visualization and interaction with OpenAPI documents.	Facilitates easy access to API documentation for developers, promoting easy testing and reducing onboarding time for new developers.
Code Generator (e.g., Swagger Codegen)	Automated code generation tools that produce client and server code from an OpenAPI specification.	Speeds up the development process by generating boilerplate code for the MVP, allowing developers to focus on implementing unique business logic and speeding up time-to-market.

4.3. Advancements in aerOS service and resource orchestration

4.3.1. Updated description and main functionalities

4.3.1.1. aerOS continuum ontology entities as a single source of truth

The aerOS continuum ontology described in section 3.1.3.1 of D4.2 has been designed having into consideration two essentials pillars in the aerOS architecture: (i) Domain federation and continuum management; and (ii) Decentralized orchestration. When it comes to aerOS orchestration, this ontology tries to facilitate the complex orchestration process in a distributed and decentralized environment such as the IoT-Edge-Cloud computing continuum, so the initial Intention Blueprint (in TOSCA format), which includes information of the service orchestration requested by a user through the Management Portal, is translated into several NGSI-LD entities so as to avoid the requirement of deploying additional databases for storing these TOSCA files in each domain, leveraging the stablished aerOS Federation and Data Fabric to store and share these entities in a decentralized way. This conceptual data model will be enhanced to describe more important data that is being identified as the project moves further, such as advanced network or communication links among services, or storage requirements needed by services components.

In the first iteration of the continuum conceptual data model, *Service* entities (e.g. an IoT service) are linked to a set of *ServiceComponents* (e.g. the IoT Edge and Central Cloud service components), which indeed are the core entities of the orchestration as the whole orchestration process is performed independently for each ServiceComponent. In addition, these entities contain specific attributes that will be later used by the orchestrator components: location requirements, Service Level Agreements, execution information, etc.

Moving to the specific interaction of aerOS orchestrator components (see section 4.3.2) with the ontology, HLO Data Aggregator uses the *IErequirement* attribute value to perform a preliminary filtering to select the candidate IEs that are able to run the component of a service in terms of computing resources, location, real-time capabilities, … Furthermore, the monitoring data of the IEs status is retrieved (IE entities) and sent to the HLO Allocation Engine, along with additional ServiceComponent running requirements (e.g. custom SLAs), to feed the Allocation AI Algorithm. After the allocation decision of this algorithm, the Implementation Blueprint (as a K8s Custom Resource) is sent to the selected LLO, which retrieves the execution information (container image, cli arguments, environment variables, network ports, …) included in the *ServiceComponentArtifacts* entity of the ServiceComponent to deploy the requested workload in the selected IE. Finally, the ServiceComponent entity is updated with the result of the orchestration process (deployed IE, status, …) to allow its further monitoring.

4.3.1.2. High-level Orchestration components decomposition

As described in D3.1 deliverable, the High-level Orchestration part in the multi-level orchestration architecture is responsible for the smart placement of the services inside the federated domains taking into account the services requirements and the infrastructure constraints. It interacts with the Low-level Orchestration to communicate the final decision.

Considering its complexity and the engagements of many partners in its development, the high-level orchestration has been decomposed into different components illustrated in Figure 9. Each component is responsible for specific duties of this orchestration level.

The **HLO Storage Engine** is responsible for converting the user service definition in TOSCA format and transforms it into a set of data entities to be stored using NGSI-LD endpoint.

The **HLO Data Aggregation and Alert system** is responsible for aggregating all the required data for the smart allocation. It also triggers the remaining stages in the placement process.

The **HLO Allocation Engine** is responsible for the AI part in the HLO. It receives the services requirements and infrastructure elements constraints to provide the allocation decision.

The HLO Deployment Engine is the component interacting with LLO and transforms the allocation decision from the **HLO Allocation Engine** and converts into a deployment request to the LLO.

4.3.1.3. Multi-Low Level Orchestrators support for multiple resource orchestrators

In the aerOS architecture, different types of infrastructure elements are considered to support rich types of compute resources such as Kubernetes clusters, limited compute modules such as Raspberry PIs etc. From the D3.1 deliverable, we have stated that Operators watching an aerOS-specific custom resource in the Low Level Orchestrator will handle the actual deployments of so-called Service Components in these compute resources.

The support of such resources requires flexibility and decoupling in the development of these operators. Depending on the containerization runtime deployed in the infrastructure element defining its type, a corresponding operator manages the deployment of service components.

In Figure 10, we describe the components constituting two types of low-level orchestrators (dockerd and K8S). It is important to note that each operator watches a different set of Service Components Custom Resources. To allow such separation, different kinds of Custom Resources Definitions are provided for each low-level orchestrator type but are consistent in their schemas.

4.3.2. Updated Structure diagram

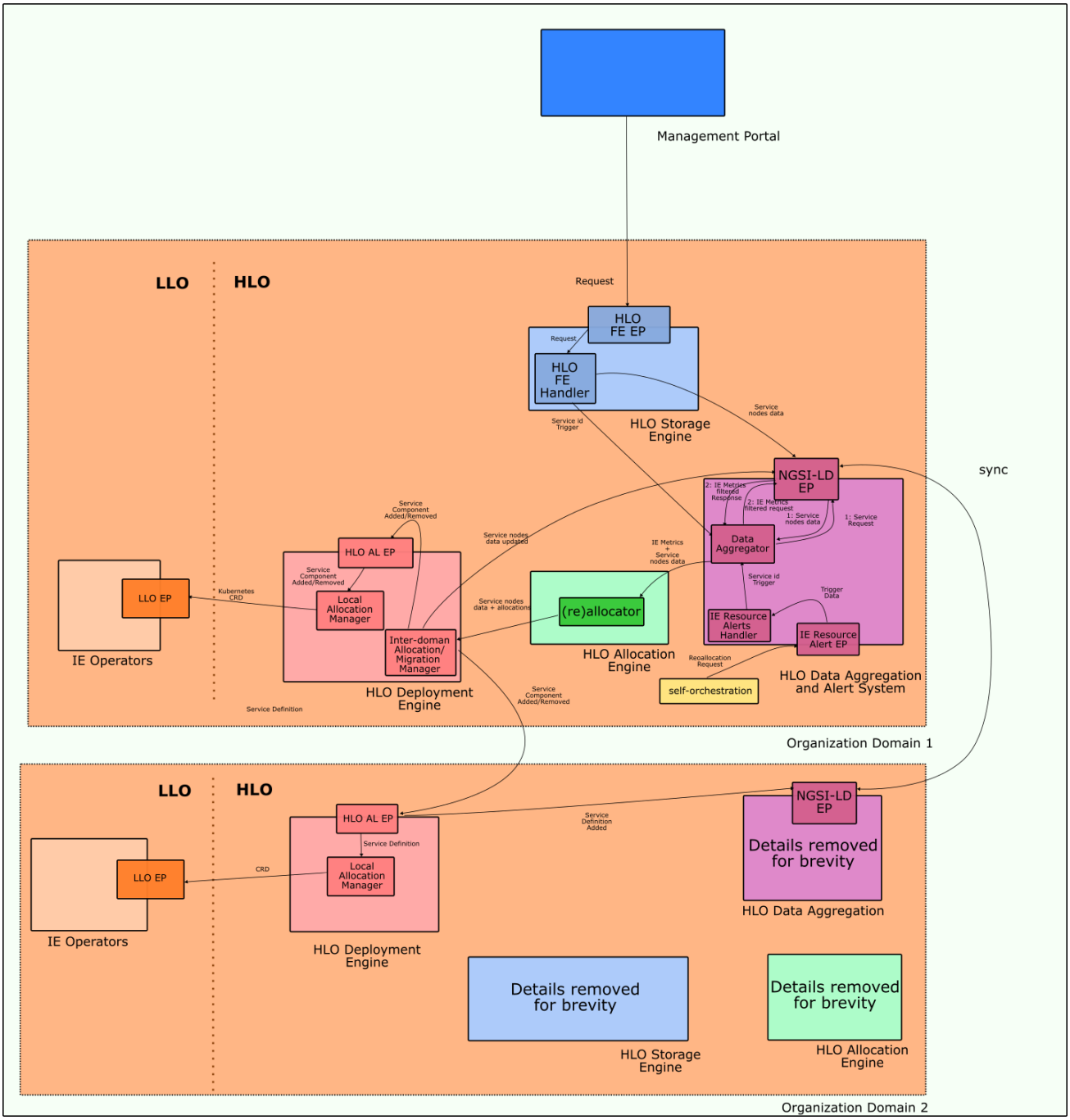


Figure 12. aerOS High-Level Orchestration Components

Table 7. aerOS High-Level Orchestration Components' description

Component	Description	Interactions
HLO Storage Engine	Component of the High-Level Orchestrator (HLO) exposing a REST endpoint responsible for receiving IoT service LCM (deployment, update, delete) requests. These requests originate from the aerOS entrypoint	**Management portal,** located in entrypoint domain. Receives TOSCA descriptor through REST endpoints. **Orion CB,** to which it pushes (using ngsi-ld API) service requirements**.** **Data Aggregator** is triggered to proceed

	domain in TOSCA format and are translated into the internal aerOS data model, represented in NGSI-LD format. Internal breakdown includes, **HLO FE EP** (HLO Front end endpoint) is exposing REST API and **HLO FE Handler** implements the business logic of this component, including data validation, translation, storage.	with the orchestration pipeline, utilizing the **Redpanda** message broker with **protobuf** formatted data.
HLO Data Aggregation and Alert System	Component of the High-Level Orchestrator (HLO) responsible to receive service deployment or migration requests and subsequently filter and retrieve, from data fabric, all computing resources (aerOS IEs) capable to support service requirements. Internal breakdown includes **Data Aggregator,** which is responsible to receive service requirements, filter and retrieve capable IE information and forwards all this information to HLO Allocation engine. **IE resource alert end point,** responsible to receive alerts from IEs self-orchestration component regarding service component migration. **IE resource alerts handler**, responsible to forward service component, that needs to be migrated, id to Data aggregator (again using Redpanda and protobuf format).	**HLO Storage Engine,** from which it receives service deployment request, when event is triggered through Redpanda message broker (with protobuf formatted payload). **Self-orchestration** component which triggers event, also through Redpanda message broker with protobuf formatted payload, with information of service component that needs to be migrated to other aerOS computing resource (IE). Local **Orion-CB** (part of aerOS data federation) is queried using filtered requests based on service component requirements, to retrieve IEs, across all aerOS continuum, capable to host newly deployed (or migrated) service component. **HLO Allocation Engine** to which it forwards service components requirements and list of IE capable to host each component. This information is conveyed using Redpanda message broker in a protobuf formatted message.
HLO Allocation Engine	Component of the High-Level Orchestrator (HLO) implementing smart algorithm which enables the most efficient allocation of each service component to the most suitable aerOS IE and forwarding decision to next orchestration level. Since all input and output is going through Redpanda message broker and is modeled using protobuf formatted messages, a variety of implementations may exist, and internal components	**HLO Data Aggregation and Alert System (**specifically Data Aggregator sub-component) is contacting, asynchronously, HLO Allocation Engine using Redpanda and submitting protobuf formatted messages including information about service components and candidate IEs for each of them. **HLO Deployment engine** is contacted, from HLO Allocation Engine, using Redpanda and submitting protobuf formatted messages which include selected IE id, LLO id, service component id.

	(e.g. data engineering, feature cleaning, specific AI algorithm, etc.) are specific to each of them.	
HLO Deployment Engine	Component of the High-Level Orchestrator (HLO) receiving (de-)allocation decision, and which can identify and addressing LLO which is responsible to access selected IE. If a link between referenced IE and service component exists, it is identified as a delete request otherwise it is a deploy request. If LLO, responsible for selected IE, is located internally, to current aerOS domain, request is forwarded to local domain allocator otherwise it is sent to the aerOS domain to which LLO, responsible for the selected IE, is located. So LCM requests arrive through Redpanda (message broker) and then allocation request is submitted in REST API, also exposed by this component. This API can be accessed either internally for local deployments, or from other domains for deployments (or migrations) decided in other domains of the continuum. All updates regarding IEs and service components decisions (de)allocations are sent to local CB to keep aerOS continuum state updated. Based on the above functional description, the components internal to HLO Deployment Engine are: **Inter-domain Allocation/Migration Manager,** which is the sub-component receiving decision from HLO Allocation Engine, accessing Deployment API (either locally or to external domain) and updates state of local domain by submitting Orion-CB ngsi-ld API. **HLO Allocation EP**, which exposes Rest API for LCM	**HLO Allocation Engine,** is sending selected IE for specific service component and related LLO information, elaborating Redpanda message broker and protobuf formatted messages. Exposed HLO Deployment Engine API **(HLO Allocation EP)** is accessed from **Inter-domain Allocation/Migration Manager (**either form local or a remote one) receiving IE and LLO id and service component data. **LLO,** is receiving, from HLO Deployment Engine, service definition template (CRD). **Orion-CB** Rest API**,** is accessed from HLO Deployment Engine, for updating domain status based on decisions and LCM performed.

	actions on indicated service component and connected IE. **Local Allocation Manager**, responsible to transform service requests to CR and forward this to the proper LLO which is connected with the selected IE.	
IE LLO (Low Level Operators)	This component acts as a thin layer abstracting all heterogeneous computing resources (aerOS IE) access. Low level orchestrators have the knowledge of accessing specific selected computing resources (IEs) within each aerOS domain. Upon receiving service definition templates, they access actual computing resources for workloads LCM activities (create, destroy, update).	Receive Service Definition Templates (CRDs) from **HLO Deployment Engine.** **Receives Implementation Blueprints custom K8s resources from HLO Deployment Engine. Depending on the information included in these blueprints and on the LLO type (K8s, Docker, …) it will deploy the requested workloads in the selected IEs,** Access computing resources (**IEs**) for workloads (service component deployments).

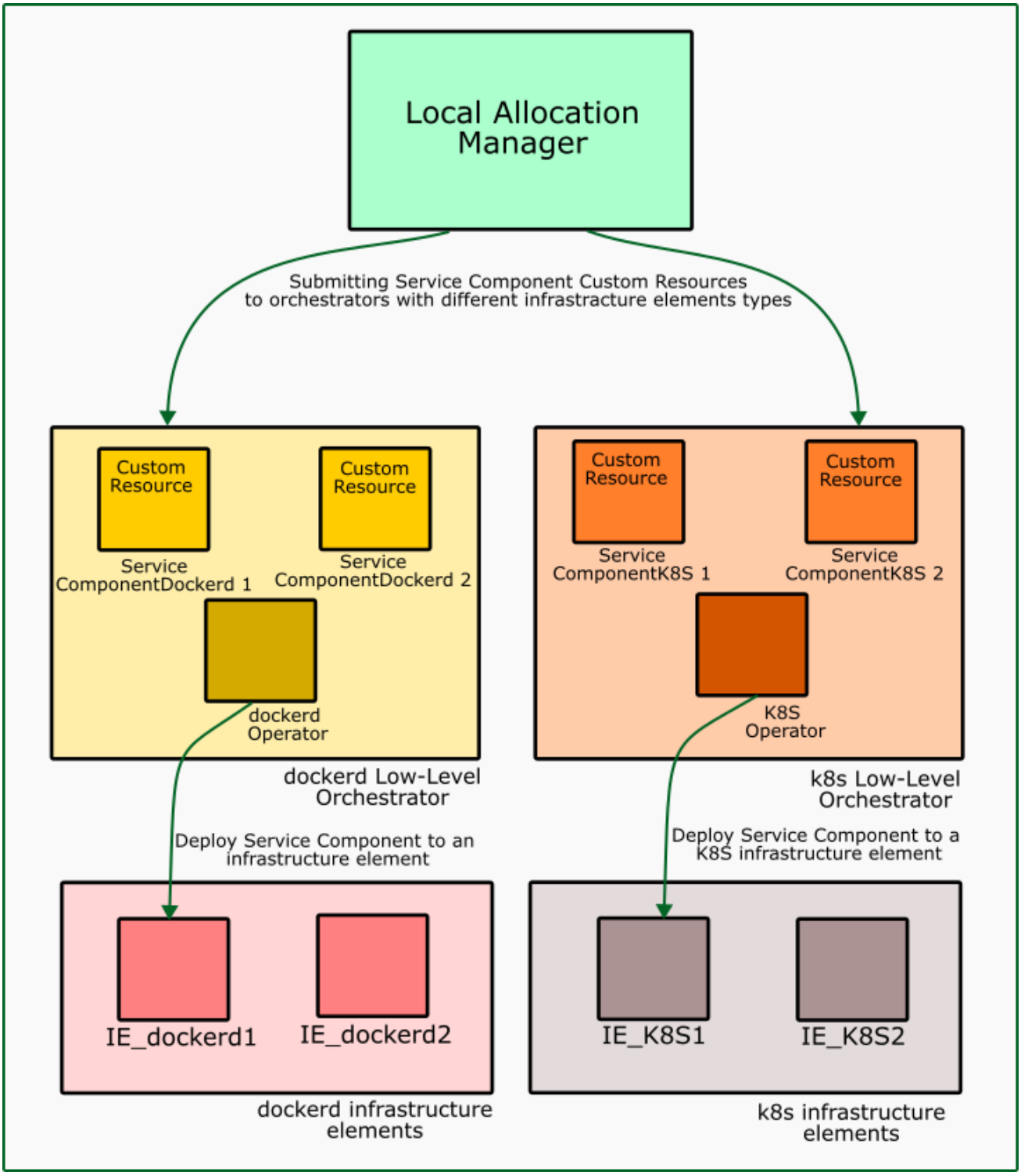


Figure 13. aerOS Multy-Low Level Orchestrators components

Table 8. aerOS Multi-Low Level Orchestrators

Component	*Description*	*Interactions*
Local Allocation Manager	The **Local Allocation Manage**r sits behind the HLO Allocation Endpoints and receive requests from the **Inter-domain Allocation/Migration Manager.** Its role is to manage the	The **Local Allocation Manager** interacts with **the Operator** inside the **Low Level Orchestrator** by submitting **Custom Resources** of different types to it, depending on the target infrastructure element for the

	allocation of the service components in a specific infrastructure element of the domain.	deployment.
Low Level Orchestrator	At the **Low Level Orchestrator**, different deployments requests for the target **infrastructure element** type are received from the **Local Allocation Manager** through **Custom Resources** submission. Depending on **Custom Resources** data, deployment requests to target **infrastructure element** type are generated.	The **Operator** inside the **Low Level Orchestrator** watches the **Service Components Custom Resources** of the corresponding type. The **Operator** then manages the deployments by submitting different types of requests to the corresponding infrastructure element.
Infrastructure Element	Depending on the type of **infrastructure element** (e.g. dockerd or K8s), it receives compatible service components deployments requests from the corresponding operator.	N/A

4.3.3. Technologies and standards deployed in MVP

Table 9. Technologies and standards deployed in MVP

Technology/Standard	Description	Justification
Protobuf	A language-agnostic data serialization format developed by Google. It is a binary serialization format used to efficiently serialize and deserialize structured data and it is commonly used for communication between different services or systems.	It has been chosen for the communication of HLO components as it provides: **Efficiency** and **performance** benefits, as a binary format is more compact than JSON, XML and other human readable commonly used formats making it less demanding in transfer and faster in processing. It is **language agnostic;** message structures are defined using a neutral interface description language. **Code generation** is automated with the use of available tools for every programming language. It is easily **extensible** also without breaking existing implementations if fields are added providing thus compatibility. The main benefit, based on all the above, is that it provides independence in components development, so all partners working on all HLO/LLO different components can work without waiting one another or having to get informed about APIs.

Redpanda event streaming platform.	Platform which provides high-performance distributed event streaming capabilities, enabling messaging and data streaming based on a defined API.	Offers the capability to trigger events and stream data that should be processed when these events rise. It provides a well-known and defined API for clients to stream or receive events and data. Development language neutral as all programming languages offer their implementing libraries. It is a quite light implementation as compared to Kafka. Decouples components and provides the capability to later expand the list of components that might need to subscribe to events and act accordingly. The choice of Redpanda provides to the development teams, working on different HLO/LLO components, to proceed independently and bind components dynamically.
Python Fastapi	A web framework for building APIs with Python based on standard Python type hints. It is designed to be easy to use, efficient, and to provide automatic validation and documentation of API endpoints. It is used for the implementation of HLO REST APIs as needed in **HLO FE EP** and **HLO Allocation EP** components.	Provides support for **fast** REST endpoints implementation. Natively provides **asynchronous** support. Enables **strong typing** and **validation** for input and output data. Produces automatic **API documentation, by** generating interactive OpenAPI and JSON Schema documentation based on the Python type hints used within code, enabling thus testing and understanding.
confluent-kafka-python	confluent-kafka-python provides a high-level Producer, Consumer and AdminClient compatible with all Apache Kafka brokers >= v0.8.	Confluent-Kafka python is backed by counfluent which is the leading company regarding Kafka, also it is good for redpanda cause it is 100% Kafka compatible, also another important aspect is the community and documentation, because it is the most used library in python regarding interacting with Kafka/redpanda.
Kopf Operator Framework	Kopf is an open-source framework for developing Kubernetes operators using Python language.	The choice of Kopf as the operator framework for the development stems mainly from the available Python expertise inside the consortium while guaranteeing minimum overhead for the K8S target infrastructure element type. Kopf is the richest framework in terms of functionality that we have reviewed inside the Python ecosystem.
Operator SDK	Go is a simple and efficient programming language developed by Google, which is used in most	Low level orchestrators are based on Kubernetes operators, so the most used and mature framework for developing them

	of the cloud-native developments (e.g. Kubernetes is written in Go). The Operator SDK is an open-source toolkit for Go to manage (build, test and package) Kubernetes Operators.	should be tested and used, among other alternatives. Furthermore, it uses Go, which is the most common language for building K8s native applications.
Orion-LD	Open-source implementation of an NGSI-LD Context Broker. This component is responsible for managing and providing real-time contextual information about various entities and their environments. In aerOS, the continuum will be represented and monitored through this contextual information.	Federated instances of Orion-LD will be in charge of retrieving all the needed data in the orchestration process from the continuum.

4.4. Advancements in cybersecurity components

This sub-section focuses on the advancements done in Task T3.4 since D3.1, namely during the months M12-M18. The challenge lies on presenting the development status of the Identity and Access Management (IAM) and the Secure API Gateway components that thoroughly described in D3.1.
In conjunction, the software elements resulting from the above-mentioned components comprise the first two elements of aerOS AAA, namely the authentication, authorization and accountability. More specifically, the IAM focuses on the authentication and authorization, while the Secure API Gateway focusing on strengthening the access control capabilities of aerOS and ensuring that access rights are preserved.

4.4.1. Updated description and main functionalities

The progress of IAM and Secure API Gateway is described in the following sub-sections elaborating on the implementation of these technologies in the aerOS ecosystem as well as on the tools that deployed to accomplish the aerOS AAA.

4.4.1.1. Identity and Access Management

The Identity and Access Management (IAM) of aerOS, as discussed in D3.1, will be based on Keycloak[9], while the authentication and authorization will be performed using the OpenID Connect (OIDC) protocol and access will be granted to aerOS users based on their roles (i.e., Role-based Access Control). Keycloak, OIDC, and RBAC thoroughly presented in D3.1; hence, in D3.2 we will not elaborate on these tools and protocols.

In this deliverable, the advances in IAM will be discussed presenting the intermediate implementation of Keycloak, OIDC, and RBAC. Furthermore, enhancing the IAM of aerOS the consortium decided to implement Keycloak with OpenLDAP[10] in order to enhance the adoption of aerOS IAM by stakeholders since with this approach all the user information of an organization can be federated automatically from the LDAP directory without needing to pass the user information to the aerOS IAM (e.g., Keycloak) manually, as it can be seen in Figure 11.

[9] https://www.keycloak.org/
[10] https://www.openldap.org/

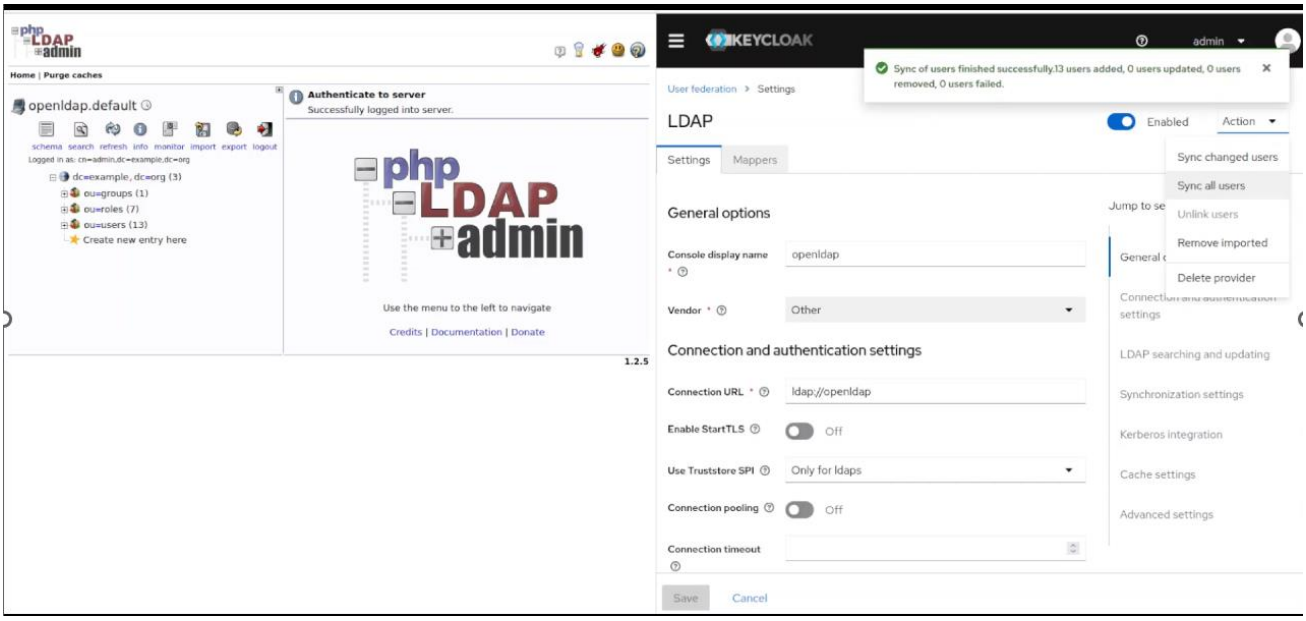


Figure 14. Synchronisation of OpenLDAP users in Keycloak

In the following it is described the setting up of Keycloak as well as the main functionalities of IAM such as the authentication and authorization using OIDC protocol, the RBAC, and the federation with OpenLDAP.

The aerOS user roles that have been deployed so far to support the RBAC activities are the listed below:

- System administrator: can access all the aerOS services.
- Domain administrator: can create new services and access all the services in his domain.
- Vertical user: can consume specific services but cannot create new ones.
- IoT service deployer: can consume and create specific services.
- Data producer: the owner of the data.

Figure 12 shows the groups that have been generated in OpenLDAP and that can be visualised in Keycloak by means of the federation that has been programmed. A group has been created for each user role that has been defined. Each of these groups has a role associated, which can be seen in Figure 13. Finally, the users have been created as shown in Figure 14, where each of them belongs to a group (with its associated role).

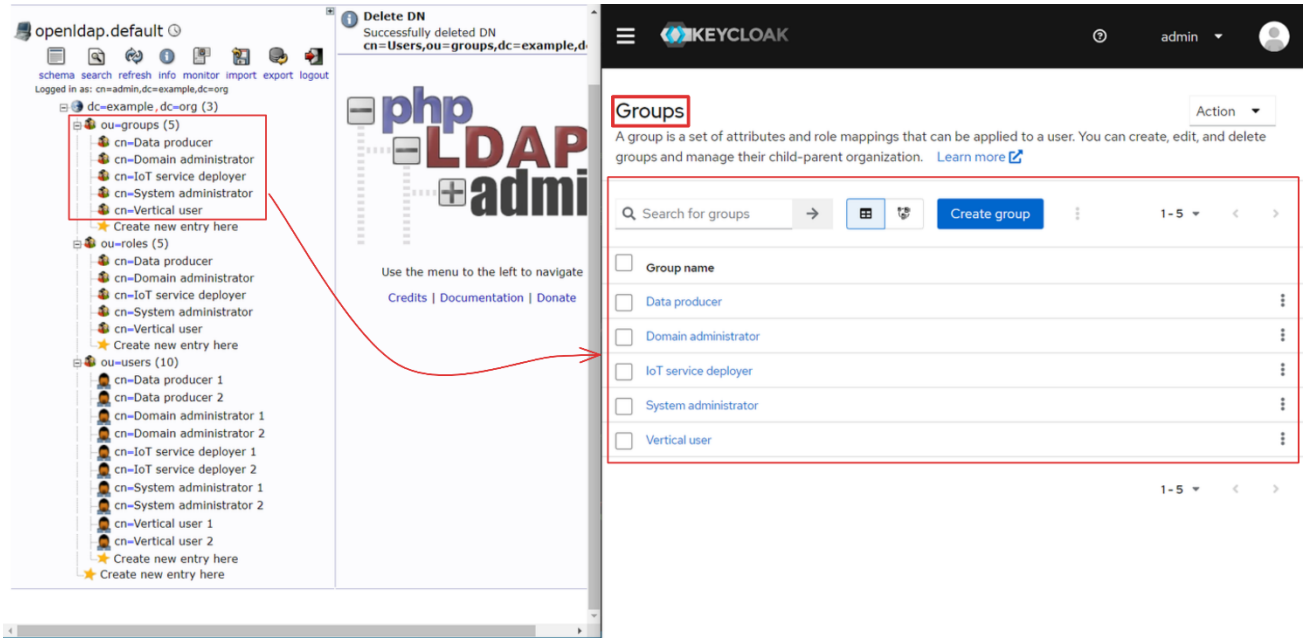


Figure 15. Groups generated for 1st MVP in OpenLDAP (and federated in Keycloak)

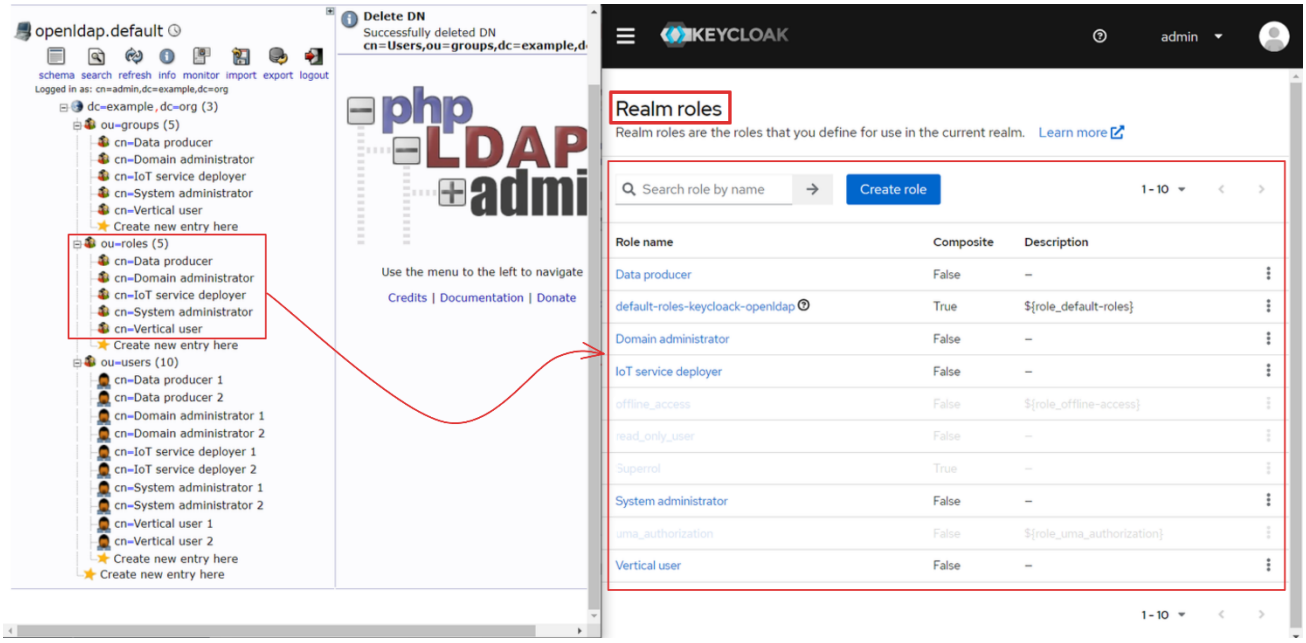


Figure 16. Roles generated for 1st MVP in OpenLDAP (and federated in Keycloak)

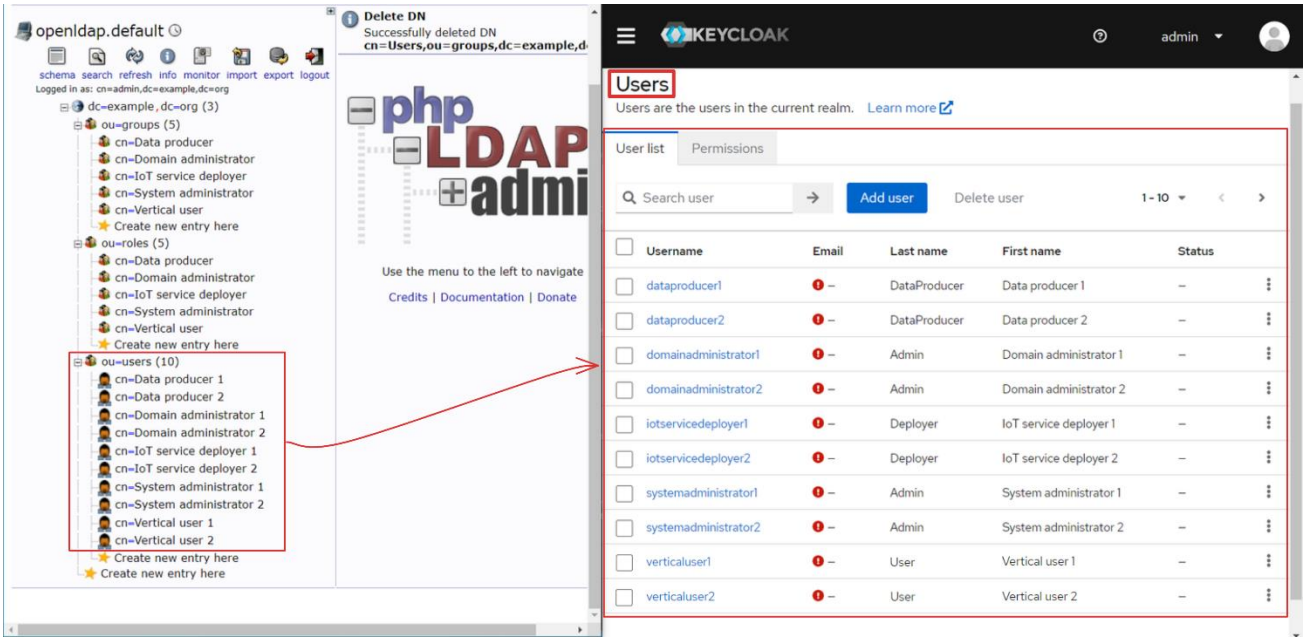


Figure 17. Users generated for 1st MVP in OpenLDAP (and federated in Keycloak)

4.4.1.2. Secure API Gateway

aerOS is comprised by multiple APIs that form the innovative meta-OS that is designed and developed in the project. However, these APIs do not incorporate security mechanism to tackle security threats, such as unauthorized access. Thus, one of the most essential elements of the aerOS architecture is the Secure API Gateway. The Secure API Gateway simplifies the process of exposing the various aerOS APIs by providing a unified exposing interface and offering advanced features for secure API management and performance, such as omitting unauthorized users from accessing the aerOS APIs. Based on these observations the KrakenD API Gateway is employed in aerOS to both enhance the aerOS cybersecurity capabilities by ensuring the security of all aerOS APIs. The rest of the section elaborates on the implementation of KrakenD in aerOS ecosystem and provides insights about its integration with the other aerOS components.

KrakenD is a stateless, distributed, high-performance effective Open-Source API Gateway written in the Go programming language that will be used to fill the architecture gap about the gateway. It will be used in aerOS to provide security to the API's that may be exposed to the Internet as well as control which users have access to which API's and which endpoints in said API's. This control will be determined according to the roles and groups established in the Identity Management component. Another objective of the gateway in the project is to homogenise the entry point to access all the resources, so the different components can access all the API's from the same place. KrakenD was also chosen for its capability to modify the incoming and outgoing traffic to suit specific needs, as well as making additional internal petitions and verifications with added scripting support. The following figure from the KrakenD Community Edition Documentation website[11] showcases these features:

[11] https://www.krakend.io/docs/overview/

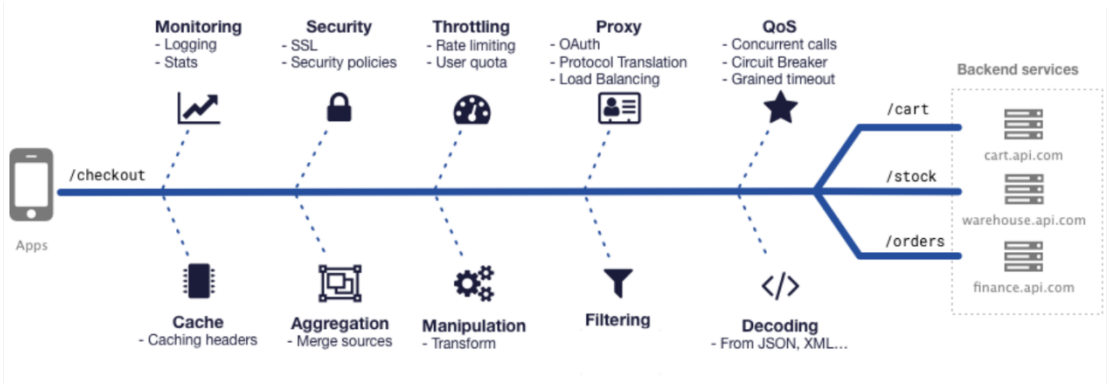


Figure 18. KrakenD and its capabilities

KrakenD has been successfully deployed in the CloudFerro Kubernetes cluster environment, alongside with the Keycloak IAM. The integration between Keycloak and KrakenD has been performed, as well as integration between KrakenD and the backend elements (i.e., aerOS APIs). Furthermore, a federated OpenLDAP database was agreed upon to act as the database for the IAM (see Section 4.4.1.1) and was subsequently integrated into the Kubernetes cluster. KrakenD was expanded so only certain allowed testing roles are allowed to access the backend, Ingress compatibility has also been installed in the cluster and KrakenD is ready and could be exposed for testing. In the following, screenshots of KrakenD are provided showcasing some of its basic functionalities (e.g., Retrieving a token from Keycloak and validating it to access an aerOS API).

4.4.2. Updated Structure diagram

Figure 18 illustrates the aerOS authentication, authorization, and access control procedure along with the relevant components that are implemented. In the presented scenario, the *client* (e.g., a user) is authenticated in the *management portal* that redirects the authentication request to *Keycloak IAM*, which retrieves user information from *OpenLDAP*. Afterwards, *Keycloak* responds with the *ID token*, which is deployed by the *client* to request access to an *aerOS API*. The request pass through *KrakenD* that validates the ID token with *Keycloak* and grants access to the API. In case that the ID token is invalid, namely the role of the client does not allow access to the requested API, the access is blocked.

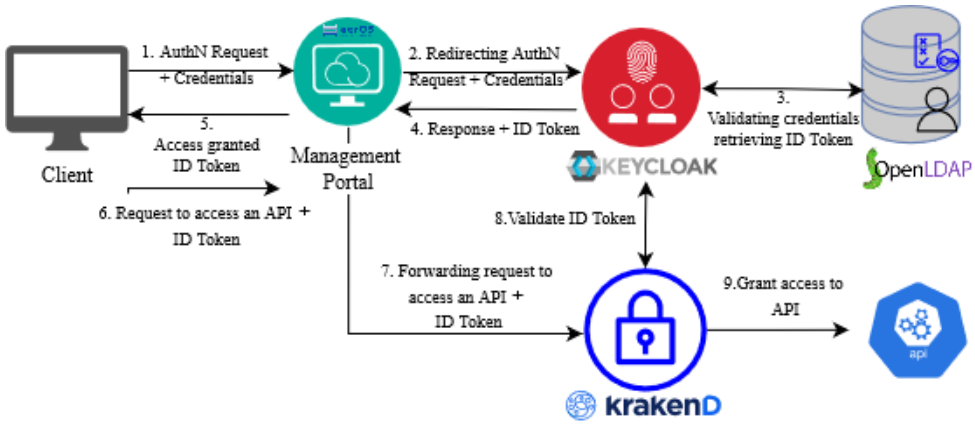


Figure 19. aerOS Authentication, authorization and access control

Table 10. List of cybersecurity tools

Component	Description	Interactions
Client	Any deployed element within the aerOS continuum that wants to access a protected endpoint.	The client obtains an ID token from Keycloak and then makes the petition to the API using the ID token.

Keycloak (IAM)	Responsible for implementing the authentication and authorization of aerOS users.	KrakenD GW API to validate the ID token. OpenLDAP to retrieve user information and support user federation.
OpenLDAP (user federation)	Registry that contains user information.	Keycloak to send user information.
KrakenD (API GW)	Access control and management of aerOS APIs.	Receives petitions from the client, verifies the ID token with Keycloak and if it is valid, allows access to aerOS APIs.
API	An aerOS API, such as OrionLD, HLO, etc.	KrakenD that manages the access to all aerOS APIs.

4.4.3. Technologies and standards deployed in MVP

The cybersecurity components deployed also in the MVP to demonstrate the aerOS cybersecurity capabilities and protection mechanisms for a user that wants to access the OrionLD API. All the aerOS APIs are protected by the KrakenD API Gateway that validates the access requests and allows or blocks the access based on the aerOS RBACs. In order to accomplish this, as presented in Figure 19, KrakenD retrieves the access token form the IAM, using its public IP, as well as the special API endpoint created to get the tokens.

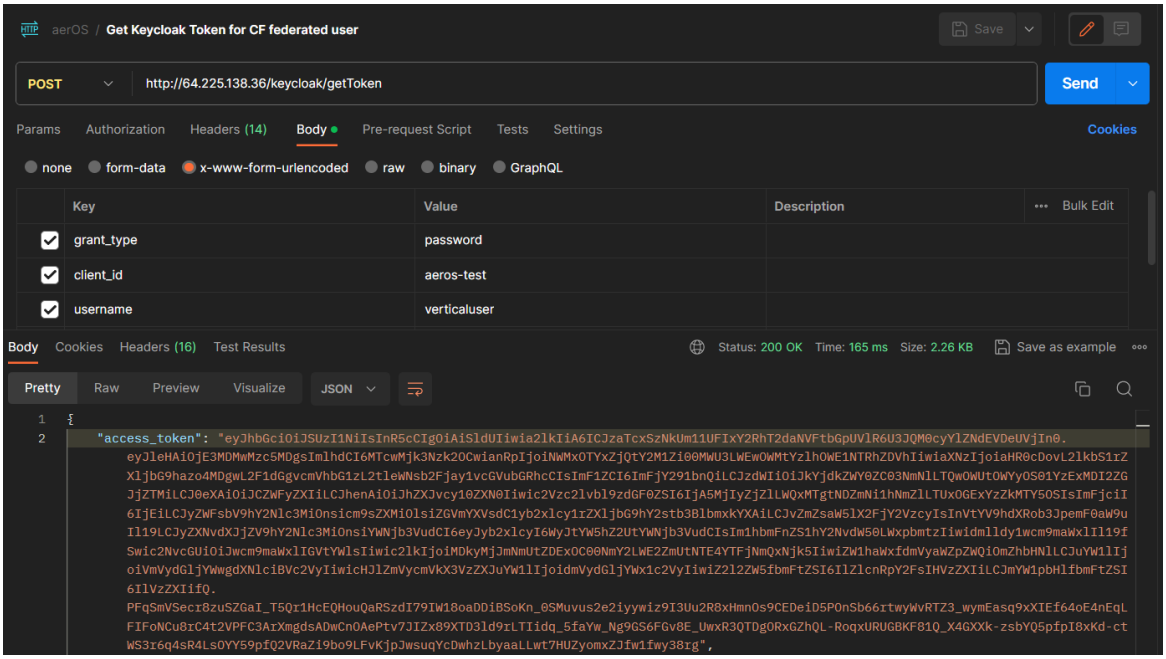


Figure 20. KrakenD retrieving access token from Keycloak

Afterwards, as depicted in Figure 20, the access token could be used to access an aerOS API, such as Orion-LD endpoint.

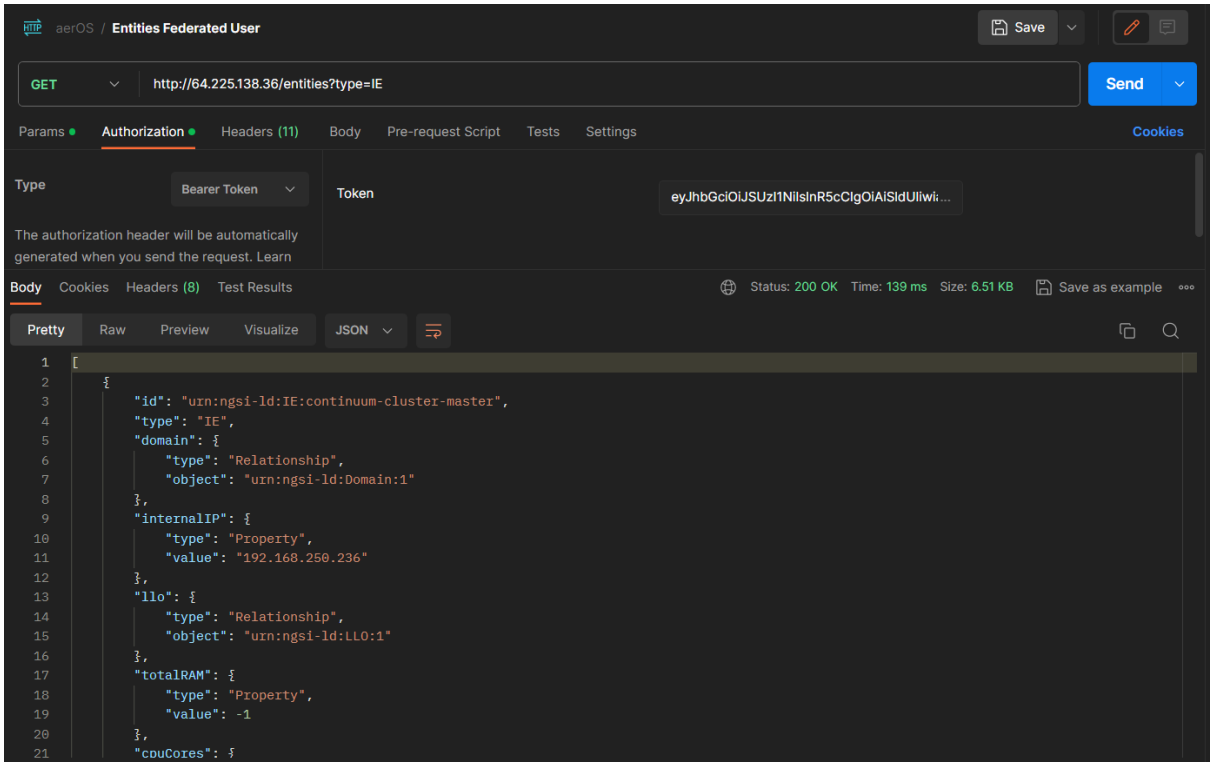


Figure 21. Deploying token to access an aerOS API

Table 11. Tools deployed in the MVP

Technology/Standard	Description	Justification
Keycloak	Detailed in D3.1.	Detailed in D3.1.
OpenID Connect	Detailed in D3.1.	Detailed in D3.1.
OpenLDAP	An open-source implementation of the LDAP protocol.	Free and open-source tool that can be integrated with Keycloak provide user federation capabilities.
KrakenD	Detailed in D3.1.	Detailed in D3.1.
RBAC	Assigning permissions to aerOS entities based on the roles of aerOS users.	It is a well-known access control mechanism that can be easily applied is aerOS due to its distinct user roles.

4.5. Advancements in node's self and monitoring tools

One of the main features of aerOS is the wide variety of IEs that exist across the computing continuum. This variety depends on its physical components, its operating system, its capabilities and even its location on the continuum. One of the objectives of aerOS is to achieve all these nodes autonomous, that is, they can function without human interaction. This particularity allows the IEs that exist in the continuum to execute actions and decisions autonomously, in addition to being able to monitor their health status in real-time.

This section of the document describes the characteristics that the aerOS nodes will have to be able to execute certain operations. These IEs are described by a set of attributes and are considered independent entities in the continuum that can execute workloads and perform internal functionality to report or modify their state towards the continuum. Making the IEs of the continuum more autonomous allows it to be more reliable in the event of outages in part of the network or services.

The following subsections describe the updated functionalities that have been designed for aerOS IEs, the architectural diagram of a node's self-capabilities and relationships between components, and the technologies and standards deployed in the MVP. This section corresponds to the evolution and developments carried out in the aerOS task T3.5.

4.5.1. Updated description and main functionalities

To allow nodes that connect to the aerOS compute continuum to be autonomous, they need to have certain capabilities. These features are offered through the aerOS self-* capabilities suite to all IEs that connect to the continuum, which are:

- **Self-awareness:** considered one of the main self-* capabilities of an autonomous system, this component analyses and obtains information from the node (CPU cores, total RAM, CPU and memory currently used, if the system has real-time characteristic, current power consumption and average power consumption – others will be added during the action), continuously monitoring its health status and workload. Due to the need to offer real-time information on the status of the IE, this module is subdivided into two components, which are executed continuously. One is in charge of obtaining the consumption (which requires more computing time) using *PowerTOP*, an open-source software designed to measure the energy consumption of a node. The other is responsible for obtaining the rest of the parameters instantly, using (among others) a Python module called *psutil*. The component that obtains the consumption needs an average of 20 seconds per execution to obtain new valid values and the other only needs 1 second to update its information. The purpose is to provide updated information to the rest of the self-* capabilities as fast as possible to modify the operation of the IE, if necessary.

  Below are screenshots of the two submodules running on the continuous development and integration cloud infrastructure of the project (provided by partner CF).

```
Name:             self-awareness-hardware-info-ds-72z8l
Namespace:        default
Priority:         0
Service Account:  default
Node:             aeros-2-jms6qnflylil-node-1/10.0.0.238
Start Time:       Tue, 26 Dec 2023 12:24:28 +0100
Labels:           controller-revision-hash=9fdfcf95b
                  name=self-awareness-hardware-info-pod
                  pod-template-generation=1
Annotations:      kubernetes.io/psp: magnum.privileged
Status:           Running
IP:               10.0.0.238
IPs:
  IP:           10.0.0.238
Controlled By:  DaemonSet/self-awareness-hardware-info-ds
Containers:
  self-awareness-hardware-info-ctr:
    Container ID:   docker://fc2f9af8aee605593232e9411f196f82c7f20c37cd4fa0df29879d2e3fa20d94
    Image:          registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-awareness/hardware_info:1.0.0
    Image ID:       docker-pullable://registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-awareness/ha
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 26 Dec 2023 12:24:29 +0100
    Ready:          True
    Restart Count:  0
    Environment:
      AEROS_KRAKEND_URL:            krakend-service:8000
      AEROS_KRAKEND_CLIENT_ID:      aeros-test
      AEROS_KRAKEND_CLIENT_SECRET:  jPWtqU5TZOJ1zKCSQ5b2tacObp4QxO8u
      AEROS_KRAKEND_USERNAME:       sjobs
      AEROS_KRAKEND_PASSWORD:       hola
      AEROS_IE_ID:                   (v1:spec.nodeName)
      AEROS_IE_IP:                   (v1:status.podIP)
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-tg9sv (ro)
```

Figure 22. Hardware info submodule running on a test cluster of infrastructure

```
Name:             self-awareness-power-consumption-ds-9lmcp
Namespace:        default
Priority:         0
Service Account:  default
Node:             aeros-2-jms6qnflylil-node-0/10.0.0.186
Start Time:       Tue, 26 Dec 2023 12:26:39 +0100
Labels:           controller-revision-hash=7d5c48bd79
                  name=self-awareness-power-consumption-pod
                  pod-template-generation=1
Annotations:      kubernetes.io/psp: magnum.privileged
Status:           Running
IP:               10.0.0.186
IPs:
  IP:           10.0.0.186
Controlled By:  DaemonSet/self-awareness-power-consumption-ds
Containers:
  self-awareness-power-consumption-ctr:
    Container ID:   docker://16c84c433ac2175041ac8e8d62dde96b5497477c4abaa22f2183ba1f0bbb7b05
    Image:          registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-awareness/power_consumption_amd64:1.0.0
    Image ID:       docker-pullable://registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-awareness/power_consum
b713b0f78e
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 26 Dec 2023 12:26:40 +0100
    Ready:          True
    Restart Count:  0
    Environment:
      AEROS_KRAKEND_URL:            krakend-service:8000
      AEROS_KRAKEND_CLIENT_ID:      aeros-test
      AEROS_KRAKEND_CLIENT_SECRET:  jPWtqU5TZOJ1zKCSQ5b2tacObp4Qx08u
      AEROS_KRAKEND_USERNAME:       sjobs
      AEROS_KRAKEND_PASSWORD:       hola
      AEROS_IE_ID:                   (v1:spec.nodeName)
      AEROS_IE_IP:                   (v1:status.podIP)
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-nqh96 (ro)
```

Figure 23. Power consumption submodule running on a test cluster of infrastructure

- **Self-orchestrator:** composed by a rules engine, a facts generator, a trigger and wrapped by an API, it is capable of managing facts, rules and alerts, obtaining information from the self-awareness and self-realtimeness modules to send warnings about problems in the IE to the aerOS orchestration systems, with the goal to improve the management and coordination of their own workloads. This improves the scalability of tasks and reduces the number of errors that occur during task execution.

  Below is a screenshot of the module running on the continuous development and integration cloud infrastructure of the project (provided by partner CF).

```
Name:             self-orchestrator-ds-wzf6h
Namespace:        default
Priority:         0
Service Account:  default
Node:             aeros-2-jms6qnflylil-node-2/10.0.0.194
Start Time:       Tue, 26 Dec 2023 12:22:57 +0100
Labels:           controller-revision-hash=c4fc46ff6
                  name=self-orchestrator-pod
                  pod-template-generation=1
Annotations:      kubernetes.io/psp: magnum.privileged
Status:           Running
IP:               10.0.0.194
IPs:
  IP:           10.0.0.194
Controlled By:  DaemonSet/self-orchestrator-ds
Containers:
  self-orchestrator-ctr:
    Container ID:   docker://09bcba48b3d0a798494c9adf30b6fc3b0b44c2f90e43361ccf09e260131c9229
    Image:          registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-orchestrator:1.0.0
    Image ID:       docker-pullable://registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-o
    Port:           8001/TCP
    Host Port:      8001/TCP
    State:          Running
      Started:      Tue, 26 Dec 2023 12:23:02 +0100
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /self-orchestrator/facts from facts-volume (rw)
      /self-orchestrator/rules from rules-volume (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-z6twj (ro)
```

Figure 24. Self-orchestrator module running on a test cluster of infrastructure

- **Self-diagnose:** capability of a node to analyse its health status through monitoring workload data, received mainly through the self-awareness and self-security modules. This information is used to obtain a normalised composite indicator adjusted to a pre-established scale of values. This will be a custom module based on a product CIC.
- **Self-security:** developed using *Suricata* (open-source network analysis and threat detection software), it monitors traffic logs in real-time from the network card to detect threats and abnormal behaviours through Log Monitoring module. The ETL (Extraction, Transformation, Load) processing module then collects the security logs, converts them into structured JSON format and sends alerts to an endpoint (self-diagnose). These warnings allow to discover different types of network attacks to detect vulnerabilities and threats at the IE level. This self-* module is currently being tested on S21Sec servers.
- **Self-API:** this self-capability consists of a global API deployed in each IE of the aerOS continuum that exposes the functions that can be executed on the rest of the self-* capabilities installed on the node, controlling the input and output data flows. Currently the data and endpoints necessary to expose have been identified.
- **Self-healing:** capability of autonomously recovering affected parts of the system both at the hardware and software level caused by failures or abnormal states. It also can restart the system to pre-established routines scheduling, if necessary.
- **Self-scaling:** possibility of horizontally increasing or decreasing the hardware resources dedicated to workloads of a node running *Kubernetes*. These changes depend on the needs of each workload, are executed in real-time, and are based on time series inference and custom logic.
- **Self-configuration:** ability to maintain the desired state of the system with the help of an abstract and reactive management of its configuration. Both the configuration itself and its possible evolution can be defined/represented based on the concepts such as “resource”, “requirement”, and “action/reaction”. Development is focused on rewriting and reorganizing the source code responsible for implementing the ASSIST-IoT automated configurator, to adapt it to the needs of aerOS.
- **Self-optimisation and adaptation:** it will aim to optimise the dissemination of the data. By using dynamic sampling techniques, it will propose new optimal sampling periods based on the current metric streams (i.e. the data obtained from self-awareness). Additionally, it will incorporate an estimation model monitoring the shifts in the metric streams, to detect data points with significant differences, since they may indicate potential anomalies.
- **Self-realtimeness:** an experimental capability that continuously monitors the performance of real-time services using their time utility (TU) that degrades with the tardiness of deadline misses. The component automatically adjust the CPU time (quota) granted to a real-time service every period to trade-off CPU utilisation and TU achieved on an IE. If a real-time service’s TU degrades below a configurable threshold self-realtimeness issues a reorchestration request.

In the next subsection the relationships between the components and their interactions are described.

4.5.2. Updated Structure diagram

The aerOS self-* capabilities is comprised of ten software components that act together and run on the nodes connected to the computing continuum. Each module is considered an independent entity within an IE and fulfils a specific function. However, to offer the described functionality they must interact with each other, creating intertwined relationships. This means that some modules depend on the information generated by others to complete their functionality and vice versa. Despite this, depending on the needs and performance of the node, one or more modules will be installed, divided into two categories. Core components are those self-* capabilities set tools that are always installed in an IE. Non-core components are those that are installed based on the performance of the node and the needs of a specific deployment. The core modules are: self-awareness, self-orchestrator, self-diagnose and self-API. The non-core modules are: self-configuration and self-healing (only live in those IEs with peripheral IoT devices), self-optimisation and adaptation (an optional plugin to the IEs), self-realtimeness (only present in those cases managing real-time containers), self-scaling (only used, on voluntary basis, if the IE is a Kubernetes worker node) and self-security (only used when additional protection

is needed beyond that provided by aerOS AAA tools). To offer a clearer vision of the set, a diagram has been created that represents the interaction between the different components and tools of the set.

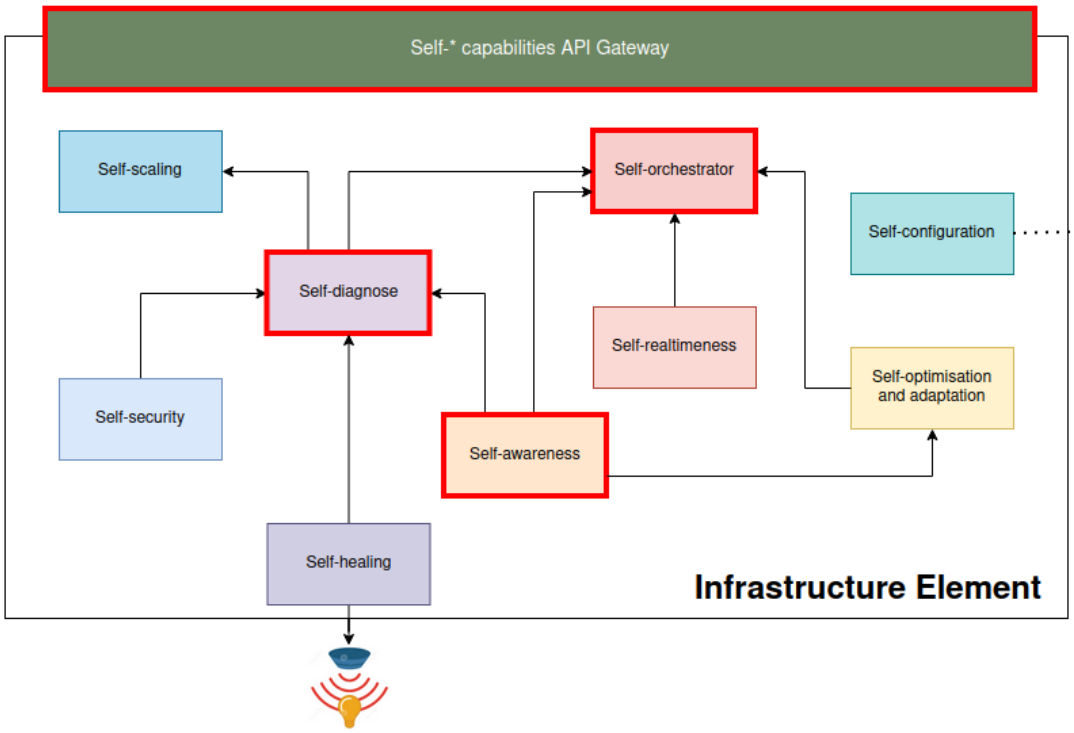
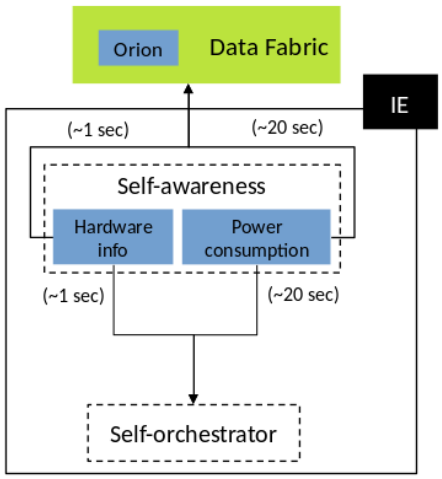


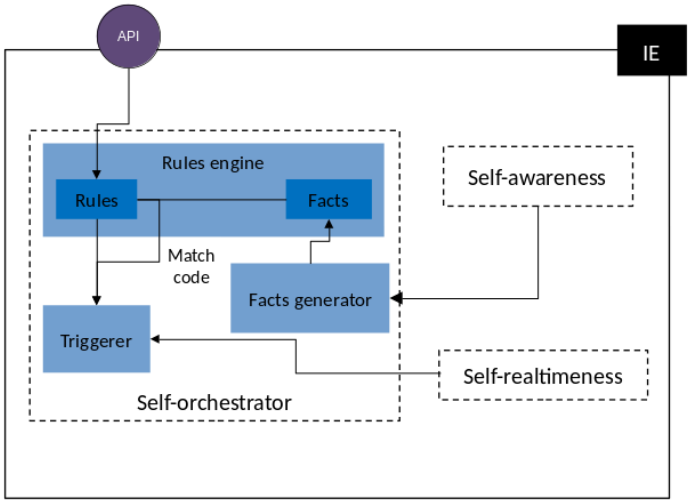
Figure 25. Relationships between the different self- capabilities of an IE*

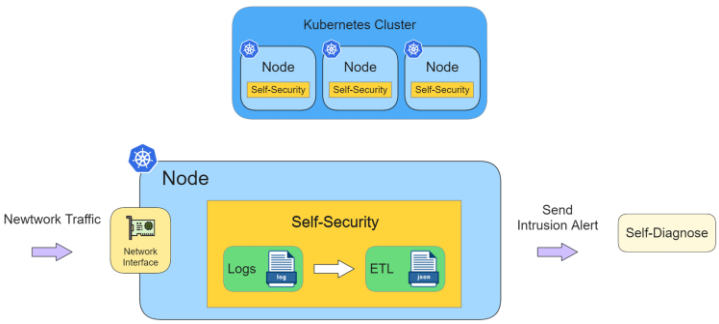
When an aerOS computing continuum node has IoT peripheral devices connected, self-configuration and self-healing modules can be installed in the IE. These systems continuously analyse the health status of these devices, sending information to the self-diagnose module to generate an index that indicates the health status of the IE. This composite index is also generated using information that comes from other modules, such as self-awareness, which is the main monitoring module, and self-security, which sends alerts about attacks on the network within the node. The possibility of exposing node actions to the outside is done through the self-API, which will include the necessary security layers. This security can be extended to the interior of the node thanks to self-security. In order to improve its own orchestration and, therefore, that of the continuum, each node has the self-orchestrator, which, fed through self-awareness, self-realtimeness, self-diagnose and self-optimisation and adaptation, determines whether to generate and launch reorchestration warnings due to IE state. A detail on the methodology for generating and handling such warnings will be explored and worked upon in the next period (M18-M30) and will be reported in D3.3. The self-awareness module sends data on the current state of the IE, the self-realtimeness sends alerts when the real-time characteristic is not met, the self-optimisation and adaptation (powered by self-awareness) sends warnings when it is expected that there may be problems in the near future with the workload and the self-diagnose will send alerts when it detects health problems on the node. Lastly, those IEs that are within a Kubernetes cluster, through the self-scaling component, will be able to horizontally scale their resources up or down. This module is powered by self-diagnose, which sends warnings when the node's health status is not as expected.

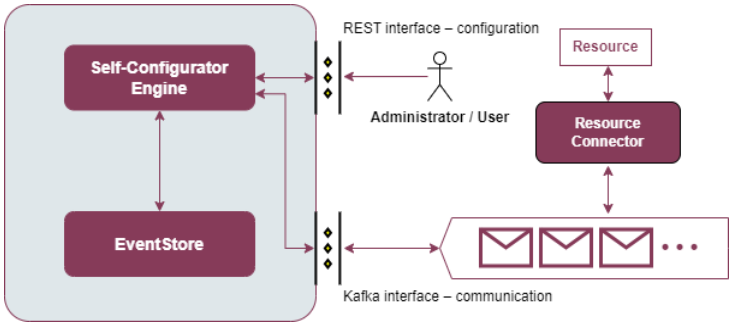
In the next table, the specific functionalities, details and interactions are further described:

Table 12. Self- capabilities components, description and interactions*

Component	Description	Interactions
Self-awareness	This is the self-capability that allows to get real-time information about the status of the IE. It gathers information about the IE and submits it to the associated Data Fabric, and is divided into two subcomponents. This module can: • Obtain parameters such as CPU data, RAM data, energy consumption, etc. • Define custom parameters. • Works on Kubernetes clusters and virtual machines. • It is capable of inform about their health status in "real-time". • Sends data periodically. This is the current schema of the development carried out in the module:  *Figure 26. Self-awareness schema* The connection with the self-orchestrator, the capture of information from several type of devices – IE- and deployment in two continuums has been tested up to M18. Immediate next steps are to enable connections with self-diagnose and self-optimisation and adaptation. These particularities of the information transmission flow will be detailed in the next iteration of the deliverable, when the implementation of the module has been completed.	It obtains information about the state of the node and directly feeds the self-diagnose, the self-orchestrator and the self-optimisation and adaptation. Additionally, it provides context information to the Context Broker associated with that IE.
Self-orchestrator	This self-capability allows to interact with aerOS global orchestration policies. This module is composed of: • Rules engine: contains the rules and facts (rule activation thresholds) to be evaluated. These rules represent the situations where a request for reorchestrating a service (e.g., the IE is too overloaded) is needed. Facts represent the current status of the IE and the network, and are directly fed	It obtains information directly from four components: (1) self-awareness (values to generate the facts), (2) self-diagnose (information about the health score of the node), (3) self-realtimeness (to

	by the self-awareness module. • Facts generator: allows to generate the activation thresholds of the rules based on the information received by the self-awareness module. • API: allows to execute CRUD (Create, Read, Update and Delete) actions dynamically on the rules stored in the rules engine. • Triggerer: generates alerts from the self-orchestrator and sends them to the aerOS global orchestration system. This is the current schema of the development carried out in the module:  *Figure 27. Self-orchestrator schema* Till M18, this component has been developed up to an MVP version, where connections with the self-awareness and self-realtimeness have been integrated and deployed in one continuum. Next immediate actions are the connections with self-diagnose and self-optimisation and adaptation. These particularities of the information transmission flow will be detailed in the next iteration of the deliverable, when the implementation of the module has been completed.	determine if the node meets the real-time characteristics) and (4) self-optimisation and adaptation (to determine whether future states of the IE should trigger orchestration actions in advance).
Self-diagnose	Component that calculates the health score, a value that represents the health status of an IE in each specific instance. Its function is to assess the health status of the device by collecting and analysing data coming from multiple sources of information and provides a health score. It will build a composite indicator (normalised, weighted and aggregated according to task decisions). This component has started to be developed till M18. However, the first functional version is expected for the next round (as this was tagged as a less priority element).	This component receives data from the self-security, self-healing and self-awareness modules, feeding in turn the self-scaling and self-orchestrator modules.
Self-security	Adapted to work in Kubernetes and non-Kubernetes environments, the two components that compose the module (Log Monitoring and ETL processing) generate cyber-	Gets information from the network card and sends alerts to the self-

	intrusion alerts that are sent to the self-diagnose module. Different configurations of Suricata are currently being tested to detect different types of intrusions. This is the current schema of the development carried out in the module:  *Figure 28. Self-security schema*	diagnose module.
Self-API	It allows to expose a single point of connection to the self-* capabilities of each node, being the global API of each IE. It will be able to retrieve certain aspects of management and will allow, for example, dynamic rules to be parametrised in the self-orchestrator module. In addition, it will be able to take the form of an API Gateway, being aligned with OpenAPI and will have the capacity to control the volumes of information that can enter and leave an IE.	It will interact with all the rest of self-* capabilities in order to manage their configuration / parameters / data.
Self-scaling	A feature of an IE that allows it to adapt to the demand for services and to be able to horizontally scale the resources dedicated to a specific workload dynamically, based on time series inference and custom logic. This is reserved for IEs within Kubernetes environments. The advance till M18 has been to adapt the previously developed (TRL3) component (from project ASSIST-IoT) to the data model and the deployment configuration for aerOS meta operating system. In the next period, where this module will be completed, it will be enhanced with new functionalities, and the integration with self-diagnose module will take place.	It receives information about the health status of the node through the self-diagnose module.
Self-configuration	The self-configuration component can be used for reactive configuration management of heterogeneous resources within an aerOS deployment. The administrator/user, through the REST interface, can define configurations, using elements such as resources, functionalities or actions that represent external events and can trigger predefined reactions. The configuration is represented by a directed acyclic graph.	This component will be one of the few that will be able to operate autonomously, without the need to interact with other functions. It only has to interact with external resources.

	 *Figure 29. Self-configuration schema* The self-configuration module can autonomously decide which alternative configurations should be applied to the system in case of an error in any of its components. To do so, it must be able to communicate with external resources. This communication is done through connectors and its function is to perform direct manipulations on the resource and to inform the self-configuration module of the resource status. The advance till M18 has been to adapt the previously developed (TRL3) component (from project ASSIST-IoT) to the data model and the deployment configuration for aerOS meta operating system. In the next period, where this module will be completed, it will be enhanced with new functionalities, and it will be deployed and tested in various continuums.	
Self-healing	This module periodically monitors the target metrics of an IE in relation to certain analytics associated to sensors status. Depending on the value obtained and the type of metric analysed, the module determines whether the value is correct or abnormal. If the value is not correct, the module is then able to applies some recovery actions into the IE and to check if the remediation attempt was successful. If the remediation is successful, the IE resumes normal operation. Otherwise, it retries a different remediation. If the number of remediation attempts exceeds a threshold, the IE is considered permanently down. Here below there is a diagram flow that represent the functioning of the module. Software-wise, there are custom PoCs being developed to analyse the status of the IEs and to identify abnormal status. The theoretical approach for certain cases has been completed, and up to M18, such cases have been replicated in a scenario with *DHT22 Digital Humidity & Temperature* sensors and *Raspberry Pi* IEs. To implement the self-healing capability, a suite of abnormal scenarios has been defined, along with the proposed healing actions to be taken: 1. Sensor Failure: ◦ *Scenario*: No measurement or measurement value that indicates outlier. ◦ *Healing*: Alert messages to exclude sensor from the set of those that provide input to the system. 2. Device Power Alert:	In the case of detecting abnormal states, or healing actions to be taken, this will feed the health score of the IE (interacting with self-diagnose capability).

- *Scenario*: Power level of the device drop below a threshold.
- *Healing*: Alert messages for battery replacement or recharging.

3. Network Protocol Violation:
   - *Scenario*: Protocol-specific violation, e.g., overwhelming the radio resources (LoRa, Duty Cycle violations).
   - *Healing*: Enforce reconfiguration to the IE.
4. Link Quality Issues:
   - *Scenario*: Radio values drop below a threshold.
   - *Healing*: Report to self-diagnose, instruct device to change link parameters.
5. Communication Failure Indication (no messages received by IE):
   - *Scenario*: Substantial amount of time without message reception might be attributed to connection lost.
   - *Healing*: Set up dedicated communication channel and poll (check-alive) the target IE.

The general software flow of all self-healing scenarios is as follows. The node is powered on and starts its normal operation. There is a value of interest (specific to each scenario) that is monitored. Once this value exceeds a threshold, a remediation attempt takes place. The success of the remediation attempt is evaluated either by the node itself or by another node (this depends on the scenario).

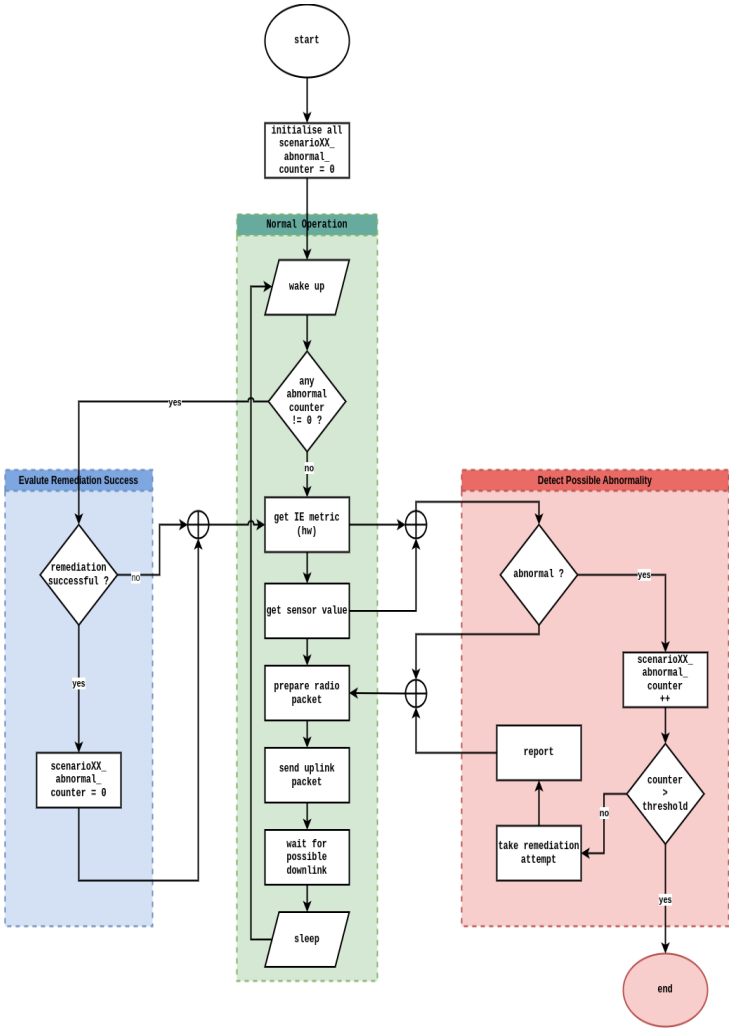
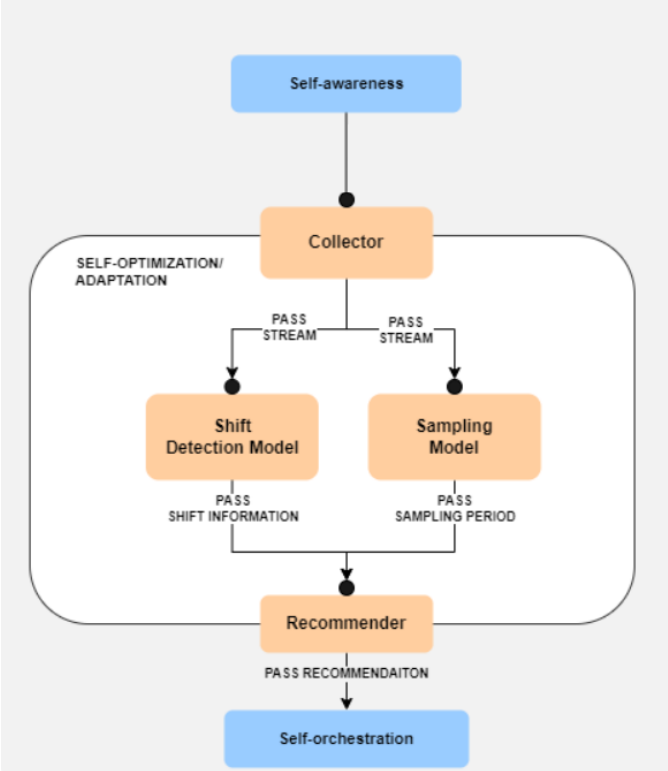


Figure 30. Self-healing schema

In order to meet its objectives, the module consists of three main components: the normal operator, the abnormal detector and the remediation evaluator. The flow may differ slightly depending on the type and capabilities of the IE or the execution scenario, however, the main flow always consists of these three components and the corresponding operations. After M18, the scenarios will be enhanced, and the custom software developed will be tested in more refined scenarios. Also, integration will take place by connecting with self-diagnose and analysing the usage of such results in one (or more) pilot(s) in aerOS.

Self-optimisation /adaptation	The goal of this module is to react in advance to potential scenarios when the IE would like to act upon (e.g., overload, network down, demand peak…). It brings the smart/predictive/proactive fashion to the self-* capabilities of an IE in aerOS. This module is to be composed of: 1. Collector (monitoring the metric streams obtained from the self-awareness service). 2. Sampling model (model that will compute the next optimal sampling period).	Fed with data on the state of the node via the self-awareness module, it sends alerts to the self-orchestrator module in case a reorchestration is needed in the near future.

	3. Shift detection model (model that will return information indicating when the change in a metric stream is detected). 4. Recommender (component exposing computed information for self-orchestrator). There below is a summary of the flow that this module follows:  *Figure 31. Self-optimisation and adaptation schema* Not being a priority for the MVP of M18, the development of this module has been reduced. Up to now, theoretical description of functioning is available, and the development of the inner components has been started.	
Self-realtimeness (kernel module)	The self-realtimeness aims at controlling the real-time performance of those containerised services (i.e., containers running in an IE) that are tagged for that purpose. It is composed of two components: **Modified kernel module:** Monitors performance of real-time services (periodic services with a soft deadline) deployed on an IE by periodically adjusting quota of containers based on the TU and tardiness of their tasks. **User space component:** Calculates each real-time service's TU from its tardiness and issues a reorchestration if the tardiness drops below a user-configurable threshold.	**Modified kernel module:** Receives containers' TU from user space component via the /proc filesystem. **User space component:** Reads real-time services (containers) tardiness from and writes updated TUs to the kernel module via the /proc filesystem. Communicates with self-orchestrator if relocation of a real-time service is required.

	Up to M18, these components have been completely developed, and has been tested in local environments and in a cloud integration environment within an aerOS domain.	

4.5.3. Technologies and standards deployed in MVP

Table 13. Self- capabilities technologies/standards, descriptions and justifications deployed in MVP*

Technology/Standard	Description	Justification
psutil (self-awareness – hardware info)	Cross-platform library for system and process monitoring in Python.	The ease of use and different functions allow for agile development and its speed allows for very short execution times.
PowerTOP (self-awareness – power consumption)	Open-source diagnostic tool that provides energy consumption by host and by process (per PID).	Allows experiment with various GNU/Linux power management configurations and obtain power consumptions from Intel, AMD, ARM and UltraSPARC processors.
json-rules-engine (self-orchestrator)	Rules engine and alert-based system to trigger orchestration requests to upper layers in the domain.	The rules are generated through simple schemas in JSON and is developed in node.js, which is fast and lightweight.
Suricata (self-security)	Is a high performance, open-source network analysis and threat detection software.	It can be integrated with Kubernetes to provide real-time security, efficiently processes and analyses data, and enhances network security and incident response.
Hierarchical Container-based Scheduling (self-realtimeness)	Linux kernel patch developed by: *Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. 2019. Container-based real-time scheduling in the Linux kernel. SIGBED Rev. 16, 3 (October 2019).*	Enables resource reservations and hierarchical scheduling of Linux control groups (cgroups) as black boxes. Processes inside a cgroup are scheduled using fixed priorities.
Custom development	Many of the self-* capabilities incorporate custom developments to achieve their functionality.	Lightweight languages and code are used. Best practices coming from DevPrivSecOps are used too.

5. Conclusions and Future Work

In conclusion, the aerOS project's first-year achievements represent a significant stride in IoT development, showcasing a meticulously designed architecture that provides a coherent environment for IoT developers. By leveraging a multitude of cutting-edge concepts and technologies, aerOS aims to unify diverse computing and network resources across the cloud-edge-IoT continuum. The project's approach is marked by a commitment to research and innovation, particularly in compute and network fabric, service fabric, and data fabric. This commitment has led to the development of a complex yet efficient system, integrating a wide range of technologies for enhanced connectivity, resource management, and orchestration. The Minimum Viable Product (MVP), a key milestone set for M18, encapsulates the essence of the aerOS vision, integrating advanced technologies and tools into a prototype demonstrating the basic functionalities of the continuum. The iterative development approach, reinforced by insights from both the development team and IoT developers, ensures continuous refinement and optimization of the system. The aerOS project, through its MVP and subsequent developments, is set to offer a scalable, secure, and resilient platform, enabling stakeholders to design and operate applications across a diverse infrastructure spectrum, from IoT devices to cloud data centers. This endeavour not only exemplifies the project's technological prowess but also highlights its potential to revolutionize the IoT landscape, making aerOS a paragon of innovation in the realm of smart services and orchestration systems. The forthcoming steps in the various domains of the aerOS project are outlined as follows, representing a pathway for future advancements.

In the domain of aerOS networking, future work is related on one hand with the integration of network services with underlying workload deployments requests to automate the chaining of cloud native VNFs to workloads connectivity across aerOS IEs, and on the other hand to extend network programmability with the integration of insights from analytics and self-monitoring components to enforce networking adaptation based on captured events and patterns recognition. The former is supported by the effort and analysis that is currently taking place with the Network Service Mesh (NSM) which is a project within the Cloud Native Computing Foundation (CNCF) ecosystem and aims to simplify the process of connecting and managing workloads across different types of environments, including cloud-native, on-premises, edge, and multi-cloud scenarios. The later will take advantage of the serverless capabilities developed within WP4 where a serverless infrastructure based on OpenFaaS has been deployed and which can support serverless functions which can respond to events triggered by analytics processes or self-* components.

The pivotal role of communication services and APIs within the dynamic cloud-to-edge computing landscape, particularly in the aerOS ecosystem, cannot be overstated. These services and APIs form the foundation for standardized, secure, and efficient interactions across the continuum, embodying the essence of interoperability and integration. The commitment to adopting standardized communication protocols such as RESTful services over HTTP and Apache Kafka, alongside the adherence to NGSI-LD standards, underscores the dedication to ensuring seamless interactions across diverse systems and technologies. The focus on automation, security, and rigorous testing and maintenance further highlights the ongoing effort to keep aerOS at the forefront of technological innovation.

The aerOS project's sophisticated orchestration framework, as outlined in its comprehensive documentation, represents a significant leap in cloud-to-edge computing and IoT integration. The aerOS continuum ontology, with its focus on domain federation and decentralized orchestration, has been adeptly designed to streamline the complex orchestration process in a distributed IoT-Edge-Cloud computing environment. By translating the initial Intention Blueprint into NGSI-LD entities, aerOS efficiently leverages its Federation and Data Fabric, thus eliminating the need for redundant databases across various domains. This approach not only enhances data sharing but also aligns with the evolving requirements of the project.

The future work of cybersecurity components in the aerOS project will involve dynamic and strategic configuration adjustments to align with the evolving needs of various project components and pilot requirements. This includes the creation of new users, roles, and groups within the Keycloak system, enhancing the robustness of identity and access management. Additionally, there will be a focus on expanding the range and capability of APIs, coupled with implementing more granular restrictions on endpoint access. These restrictions will be governed by Access Portal directives or role-based controls within KrakenD, ensuring a secure and controlled environment. Code modifications are also anticipated, aimed at enhancing functionality

and responding proactively to emerging cybersecurity challenges. These changes are crucial for maintaining the integrity and security of the aerOS ecosystem, as it continues to evolve and adapt to new requirements and technological advancements.

The future work with regards to self-* modules is structured as follows. First, the currently functional modules (self-awareness, self-realtimeness, self-orchestration and self-security) will be integrated in different pilots. Feedback gathered in such exercise will serve to enhance/adapt the components to work in more mature aerOS scenarios. Second, the modules that were less prioritised will now be emphasised: self-adaptation, self-scalability, self-healing and self-configuration. Regarding adaptation, algorithms are already being investigated that will connect aerOS content distribution broker (Redpanda/Kafka) in the context of an IE to predict specific network congestion / workload behaviours to react in advance and trigger actions (via self-orchestration). The self-scalability module coming from a previous research effort will be adapted to the new distributed challenges proposed by aerOS and will be also tested in various pilots. The scenarios for self-healing, with real IoT devices, will be now materialized, with lessons learned and automated software for recovery to be developed. Also, self-configuration state machines will be encapsulated and included in the available suite (later in the project). Lastly, the wrapping element that will allow connectivity to all the abovementioned functionalities (self-API) will be developed and integrated, starting as soon as M19.