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

D3.3 – Final distributed compute infrastructure specification and implementation

Deliverable No.	**D3.3**	**Due Date**	**28-FEB-2025***
Type	**Other**	**Dissemination Level**	**Public**
Version	**1.0**	**WP**	**WP3**
Description	*Final specification and final version of implementation of components* *The due date has been requested to be shifted to M31(31-MAR-2025) in the on-going amendment to the Grant Agreement.*		

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 SP ZOO	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
Ignacio Lacalle	P01 UPV	iglaub@upv.es
Raúl San Julián	P01 UPV	rausanga@upv.es
Rafael Vaño	P01 UPV	ravagar2@upv.es
Salvador Cuñat	P01 UPV	salcuane@upv.es
Fernando Boronat	P01 UPV	fboronat@dcom.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
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
Amparo Sancho Arellano	P05 SIEMENS	amparo.sancho-arellano@siemens.com
Philippe Buschmann	P05 SIEMENS	philippe.buschmann@siemens.com
Korbinian Pfab	P05 SIEMENS	korbinian.pfab@siemens.com
Ioannis Makropodis	P10 IQB	giannis.makropodis@inqbit.io
Vasiliki Maria Sampazioti	P10 IQB	vasiliki.maria.sampazioti@inqbit.io
Aristeidis Farao	P10 IQB	aris.farao@inqbit.io
Christos Milarokostas	P11 FOGUS	milarokostas@fogus.gr
Alexandros Kakyris	P11 FOGUS	akakyris@fogus.gr
Katerina Giannopoulou	P11 FOGUS	kgiannopoulou@fogus.gr
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
Yan Chen	P14 ICTFI	yan.chen@ictficial.com
Masoud Shokrnezhad	P14 ICTFI	masoud.shokrnezhad@ictficial.com
Hao Yu	P14 ICTFI	hao.yu@ictficial.com
Qize Guo	P14 ICTFI	qize.guo@ictficial.com
George Koumaras	P15 INF	gkoumaras@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
26-11-2024	0.1	Final Table of Contents
21-01-2025	0.2	First round of contributions
04-02-2025	0.5	Merged document with first round of contributions. Start final round of contributions
18-02-2025	0.6	Collection of final round of contributions.
04-03-2025	0.7	Merged document with the final round of contributions.
06-03-2025	0.75	Complete the document check from the lead editor
12-03-2025	0.8	Receive comments from IR and start addressing them.
14-03-2025	0.85	Check on final version after IR
17-03-2025	0.9	Submission to Project Coordination
17-03-2025	1.0	Final version for submission

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	Amparo Sancho Arellano (Siemens), Vivek Kulkarni (Siemens)
Internal Reviewer(s)	Eduardo Garro (PRO), Nikolaos Zombakis (8Bells)

Executive Summary

The document is contextualized to the works in aerOS' WP3: *aerOS secure, scalable and decentralized compute infrastructure*. The present deliverable is the third and final version of WP3 deliverables planned for M30. The deliverable is based on the aerOS module definitions presented in D3.1 (initial distributed compute infrastructure specification and implementation), D3.2 (intermediate distributed compute infrastructure specification and implementation), D2.6 (aerOS architecture definition (1)) and D2.7 (aerOS architecture definition (2)); and depicts final version of WP3 activities presenting the relevant components of the aerOS architecture composed from the following tasks:

- 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.

D3.3 is structured in a manner that clearly provides the methodological and technological advances for every task in the context of the aerOS decentralized infrastructure and performed since D3.2.

IMPORTANT: This deliverable is of type OTHER. This means that D3.3 is mostly a software deliverable. While this document reports the advances of tasks T3.1-T3.5 in the period M19-M30, it must be understood together with the software release that is uploaded alongside it.

Until mid-term, WP3 smart-networking architecture focused on a highly integrated service mesh using intra- and inter-domain strategies. Key technologies included eBPF (via Cilium) for packet management, OpenFlow for network adaptability, and RESTful APIs with Kafka and FIWARE IoT agents for cloud-to-edge communication. Scalable orchestration is achieved with Kubernetes operators, Kafka, and ML tools like Kubeflow and MLFlow. Security is enforced through KrakenD, Keycloak, and OpenID Connect for API protection and IAM. Autonomous node monitoring and self-orchestration leverage PowerTOP, psutil, json-rules-engine, and KubeEdge, enhancing resilience and efficiency in edge computing.

After mid-term, the transition from MVPv1 (M18) to MVPv2 (M30) highlights the continuous improvements made in:

- Networking and service orchestration, ensuring seamless deployment across domains.
- Cybersecurity mechanisms, enforcing secure access and trust management.
- Self- capabilities*, enabling autonomous optimization and failure recovery.

The iterative development model allowed aerOS to refine its architecture based on real-world use cases, ensuring practical applicability and robust performance. By integrating cutting-edge cloud-native technologies, AI-driven orchestration, and secure networking solutions, aerOS positions itself as a future-ready platform for managing distributed compute environments. Overall, MVPv2 successfully validates the aerOS concept, paving the way for its deployment in industrial, IoT, and cloud-edge scenarios.

The final implementation of WP3 components marks the culmination of extensive research and development, establishing it as a fully functional, decentralized, and scalable compute infrastructure for distributed IoT-edge-cloud environments. Key advancements include:

- Smart Networking: Secure, scalable, and real-time connectivity across domains using service mesh, WireGuard, ONOS, and dynamic networking solutions.
- Communication Services & APIs: Standardized API exposure (OpenAPI, AsyncAPI) improves interoperability, while low-code tools simplify integration and automation.
- Orchestration & Resource Management: AI-driven decision-making, ML-powered monitoring, and dynamic workload balancing optimize system performance.

- Cybersecurity Reinforcements: Strong IAM via Keycloak and KrakenD ensures secure access, with RBAC and OpenID Connect enhancing data protection.
- Autonomous Operations: Self-monitoring, anomaly detection, and self-healing reduce human intervention and maximize reliability.

Table of contents

List of tables

List of figures

List of acronyms

Acronym	Explanation
AAA	Authentication, Authorisation, Accountability
ACNC	Adaptable Computing-Network Convergence
API	Application Programming Interface
BLE	Bluetooth Low Energy
BS	Base Station
CBAC	Context-Based Access Control
CEI	Cloud-Edge-IoT
CIDR	Classless Inter-Domain Routing
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CRD	Custom Resource Definition
CRUD	Create, Read, Update, and Delete
DDS	Data Distribution Service
DDQL-GNN	Double-Deep Q-Learning-Generative Neural Network
DevOps	Development and Operations
DevPrivSecOps	Development, Privacy, Security and Operations
DRL	Deep Reinforcement Learning
ETL	Extract, Transform, Load
ETSI	European Telecommunications Standards Institute
eBPF	Extended Berkeley Packet Filter
FaaS	Function-as-a-Service
GNN	Generative Neural Network
HATEOAS	Hypermedia As The Engine Of Application State
HLO	High Level Orchestrator
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IdM	Identity Management
IE	Infrastructure Element
IoT	Internet of Things
K8s	Kubernetes
LCM	LifeCycle Management
LDAP	Lightweight Directory Access Protocol

LLO	Low Level Orchestrator
MANO	Management and Orchestration
MDP	Markov-Decision Process
MILP	Mixed Integer Lineal Programming
ML	Machine Learning
MQTT	Message Queue Telemetry Transport
MVP	Minimum Viable Product
NFV	Network Function Virtualization
NGSI-LD	Next Generation Service Interface – Lined Data
NS	Network Service
NSD	Network Service Descriptor
NSM	Network Service Manager
OAS	Open API Specifications
OIDC	OpenID Connect
OMG	Object Management Group
OPC UA	Open Platform Communications – Unified Architecture
OSM	Open Source MANO
PIRA	Placement, Instance Assignment, Request Prioritization, and Allocation
PPO	Proximal Policy Optimization
QoE / QoS	Quality of Experience / Service
RBAC	Role-based access control
ROS / ROS2	Robot Operating System
SDK	Software Development Kit
SDN	Software-Defined Network
SLA	Service Level Agreement
SSO	Single-Sign On
TOSCA	Topology and Orchestration Specification for Cloud Applications
TSDB	TimeSeries DataBase
TSN	Time-Sensitive Networking
VIM	Virtual Infrastructure Manager
VNF	Virtual Network Function
VNFD	Virtual Network Function Descriptor
VPN	Virtual Private Network
VPP	Vector Packet Processor

1. About this document

Deliverable D3.3 presents a concrete view of the final methodological specification and technological implementation of the components that constitute the aerOS decentralised infrastructure from WP2, which is an essential part of the aerOS Meta-OS. It builds up on the candidate technologies that thoroughly described in D3.1, D3.2 and elaborates on the final state of the composing components and their interactions. This deliverable is the final blueprint of the aerOS infrastructure and the components that developed in WP3 and posed to be integrated in aerOS use cases as detailed in WP5 deliverables.

1.1. Deliverable context

Item	Description
Objectives	**O1** (Design, implementation and validation of aerOS for optimal orchestration): Final implementation of the components related to aerOS orchestration capabilities. **O2** (Intelligent realisation of smart network functions for aerOS): Final implementation of the smart-networking components. **O3** (Definition and implementation of decentralised security, privacy and trust): Final implmentation 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): Final implementation of the NGSI-LD module and its integration with other aerOS communication services and APIs.
Work plan	D3.3 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 D3.3 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. D3.3 presents the final integration of components defined by WP3 tasks within the decentralized infrastructure. The final development of the D3.3 components are contributing to WP5 integration and use case deployments tasks.
Milestones	This deliverable is the final step from WP3 towards the achievement of the milestone *MS7 – Final software components release* (M30).

Deliverables	D3.3 is based on the components that are described in D2.6 and D2.7, and the candidate technologies analysed in D3.1 (Initial distributed compute infrastructure specification and implementation) and their intermediate development and integration as described in D3.2 (Intermediate distributed compute infrastructure implementation). Additionally, this deliverable is coordinated with deliverable D4.3, which is delivered at the same time.

1.2. The rationale behind the structure

D3.3 details the final development and integration of the functional components 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 version 2 (MVPv2). 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 MVPv2, while Section 4 elaborates on the advancements of the five WP3 tasks, detailed in separate subsections that follow the same formal structure. More specifically, the subsections 4.1-4.5 begin with an updated description of the main functionalities of the WP3 components to the ones described in D3.2. Later, they provide the updated structure diagrams along with a description of each component, and concludes with the technologies and standards that employed in the MVPv2. Finally, Section 5 concludes the deliverable.

1.3. Outcomes of the deliverable

This deliverable aims at providing the final version of the aerOS infrastructure components with the work done in WP3 to accomplish the MVPv2. As in D3.1 and D3.2, the components descriptions are abstracted at the start of each subsection of section 4 (4.1-4.5), and updated advancements of those components 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 produce 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), deploying automated procedures that minimizes the human interaction during all the operations of IEs. To accomplish this, several functional and runtime parameters, such as health and security status, are provided.

1.4. Version-specific notes

As mentioned above, this deliverable is of type OTHER. This means that D3.3 is mostly a software deliverable. While this document reports the advances of tasks T3.1-T3.5 in the period M19-M30, 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 reflects 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 M30.

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 (D3.3). 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.

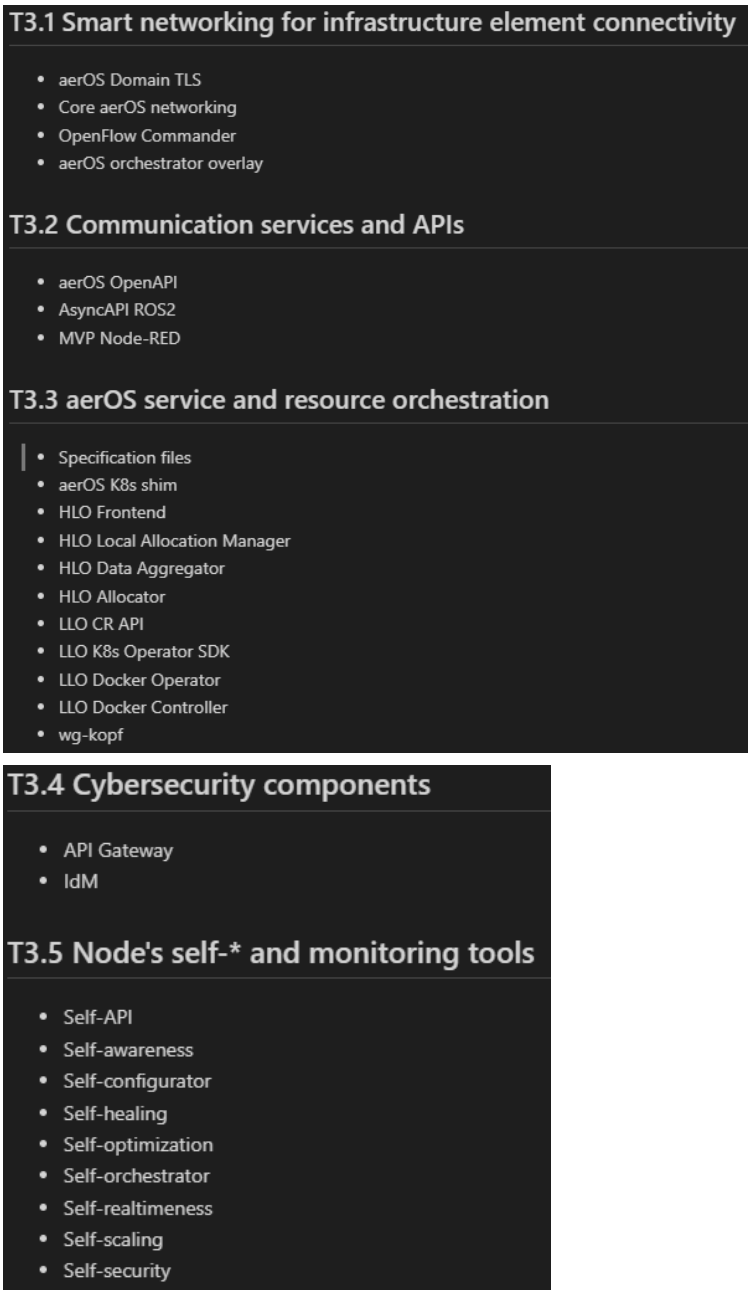


Figure 1. Software release of D3.3

2. MVP Overview

Over the course of its realization, aerOS has carefully designed an architecture aimed at providing IoT developers with a coherent environment to leverage distributed capabilities across the entire continuum. This architecture delivers a unified execution environment to support the deployment and reuse of IoT services seamlessly. With a vision to functionally unify a diverse range of computing and network resources—from cloud to edge and even IoT devices—the project has employed and integrated numerous state-of-the-art concepts and technologies.

Building upon the foundational architecture, significant advancements beyond the state of the art have been achieved by M30 through research, development, and implementation in key technical domains. These include advancements in compute and network fabric, service fabric, and data fabric, which collectively underpinned the development of new components and additional functionalities. Extending the initial MVPv1, which was delivered by M18, MVPv2 (delivered in M30) consolidated recent project advancements and addressed various development and integration complexities, providing an enhanced platform which can validate and demonstrate the final technological achievements of aerOS.

The MVPv2 has been structured in different flows, which demonstration has been recorded and will be uploaded to aerOS' official YouTube channel as soon as the post-editing activities are finalized.

aerOS Meta-OS encompasses 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 that need to be employed in order to provide the minimum for the execution environment that aerOS requires. Additionally, cybersecurity tools and trust management are essential to ensure private and secure communications and access to services over all the aerOS continuum. All IEs and aerOS domains seamlessly expose APIs for fully defined communication among components and services. Respectively, Data Fabric technologies and integrated components are designed to support the transition from heterogeneous IoT data to a unified Data Dabric, 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. Even more, AI tasks are designed to run over different 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 employed to realize the continuum and all these are 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). The complex nature of the above tasks and the integration of so many diverse technologies and implementing components introduced the requirement for an iterative development which would 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 the complexities and successfully achieving the project's goals could not be accomplished in a single stage. Thus, following the agile methodology of the project, a clear, staged strategy was defined and implemented. Initially, the aerOS team developed a Minimum Viable Product (MVP) by M18 to integrate the aforementioned technologies and tools into a functional prototype. By month 30, this approach has evolved further, leading to the completion of MVPv2. Building upon the insights gained from the initial MVP, the aerOS team refined the architecture concepts and expanded the platform's capabilities, addressing new use cases and challenges. MVPv2 not only realizes all the core functionalities of a Meta-OS for continuum, as designed by aerOS, but also introduces additional components and features that enhance the overall system's performance and scalability. Throughout this process, aerOS has maintained a focus on resource efficiency, validating with MVPv2 that the aerOS remains a lightweight implementation while preserving the platform's core functionalities. MVPv2 also includes advanced safeguards and mitigations, enabling seamless deployment to pilot locations and allowing the team to validate and fine-tune real-world scenarios. This iterative development approach has proven invaluable in demonstrating the feasibility, viability, and effectiveness of aerOS's architecture in real and diverse environments.

While the initial MVP encompassed the most compelling -first version of- aerOS functionalities, MVPv2 integrates all components of architecture building blocks and is thus a valuable ecosystem for demonstrating core concepts of aerOS architecture for a continuum Meta-OS. MVPv2 integrates two aerOS domains, which are deployed in two distinct locations, in geographic and administration terms, to demonstrate its functionality over the public cloud. Additionally, a mobile domain, although it is not a part of the continuous development and deployment process, is ad-hoc integrated when we need to exhibit the process, and the simplicity of this process, of integrating new infrastructure and extending the continuum.

One of the core two domains is designed to be the entrypoint domain, while the other stands as a plain aerOS domain, which could be deployed anywhere across the continuum. The entrypoint domain is located in 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, and this is the purpose of supporting a third one mobile domain which is provided with minimal legacy equipment from UPV.

Like its predecessor, MVPv2 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 and D3.2, the figure below represents the building blocks which WP3 addresses.

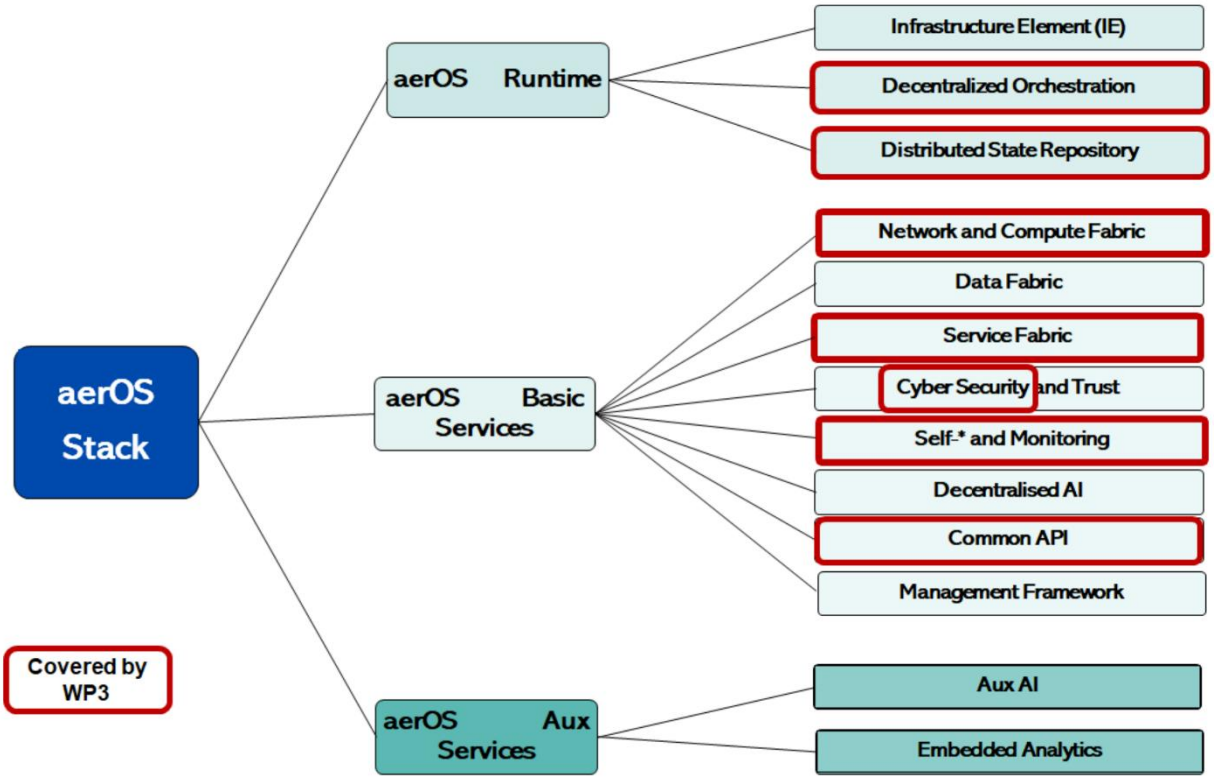


Figure 2. Building blocks of WP3

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. While first MVP version, prioritized components which were considered to provide functionalities prominent in establishing a prototype to demonstrate aerOS continuum, MVPv2 integrated many more functionalities that unleash the potential of an efficient management of resources and services for continuum actors in various industry verticals from edge to cloud.

With the target of supporting, through the deployment of MVPv2, a fully integrated environment that demonstrates advanced federated orchestration capabilities—managing a variety of services across diverse heterogeneous resources—WP3 has focused on providing the underlying mechanisms to enable this process. These mechanisms have been further refined and validated by month 30, ensuring their seamless integration into existing isolated computing infrastructures and their smooth transformation into aerOS-capable domains, as exemplified by the aerOS pilot sites.

While tasks T3.1-T3.5 have devoted to completing the established formal goals, the efforts required to materialise the MVPv2 have been directed, mostly, to:

- Network programmability and automated connectivity management, targeting cross-domain overlay connections built over public networks, interaction with network devices external to aerOS ecosystem, and providing binds with telco NFV standards.
- API development and standardization based on industry specifications, which can boost interoperability, security, and automation.
- Low code tools integration for introducing streamlined data exchange and enabling easy users interaction.
- Extended orchestration capabilities, incorporating networking isolation, AI-driven decision-making with explainability, and energy-aware resource selection.
- Complete security and access control mechanisms, including IAM, RBAC, and secure API gateways, ensuring controlled access to aerOS services, alongside secure development pipelines for component integration.
- Self-* capabilities, enhancing IE monitoring, anomaly detection, self-optimization, self-healing, and real-time status updates, which can support dynamic and resilient service orchestration across the continuum

The following paragraphs provide a summary of the outcomes of each task, of WP3, which have been delivered and included in MVPv2 realization. Additionally, their relevance in establishing aerOS continuum establishment, is roughly presented.

The first release of MVP (v1) established the foundations for network integration in a connected continuum. Inter-domain connectivity and service exposure were, thus, already ensured by M18. For MVPv2 focus shifted towards advancing the programmability and automation of network connectivity. The crucial point was to ensure the automated connection between service components (isolated pieces forming a service flow). For doing so, cross-domain overlay connections were needed. Now, every time that such flow is orchestrated, isolated components can automatically and securely connect among each other, regardless their location. Components that can interact with external networking infrastructure, based on programmability of external OpenFlow capable devices, were investigated with the goal to provide a close binding with hosting premises. Additionally, identifying the advantages of aligning with industry-standard frameworks, in this period aerOS networking capabilities were integrated with key telco domain and NFV technologies. This includes the task to integrate with openCAPIF, which implements the CAPIF (Common API Framework) specification, developed by ETSI, with the aim to facilitate interoperability with telecom networks. By aligning with CAPIF, the programmability and automation of aerOS network is enhanced, enabling seamless service orchestration and integration across diverse environments. This effort is part of a broader goal to bridge cloud-native networking with traditional telecom infrastructure, ensuring compatibility with future NFV and telecom standards.

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, aerOS did not only streamline API documentation but also facilitated their integration across diverse tools and platforms. Additionally, the adoption of the low-code tool Node-RED has been pivotal in enhancing user interactions within the aerOS ecosystem. It has enabled the creation of a user-friendly UI interface that allows users to effortlessly send data to Orion-LD and

automatically publish information in IOTA, further enriching the system's interoperability and data handling capabilities. 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 MVPv2.

The capacity of orchestrating microservice applications across heterogeneous Infrastructure Elements of an aerOS continuum is one of the most prominent innovations of the project. While in MPVv1 the foundational components that can ensure structured orchestration were successfully developed and validated, in the second period, capabilities with a special focus on network automation sere advanced. These include, among others, AI-driven decision-making, and enhanced resource efficiency. Developments in first period (M18), come up with a double layer orchestration process. HLO and LLO layers are key components of aerOS, enabling efficient service deployment across the continuum. LLO provided orchestration for Kubernetes (K8s), container, and Docker-based workloads, while HLO managed IE selection, requirement processing, and deployment coordination. Now within this period and for MVPv2, automated networking capabilities were integrated to enable dynamic connectivity management, and HLO and LLO components were extended to support isolated overlay subnets allocation per service, improving segmentation and security. HLO was further enhanced with energy-aware selection criteria, optimizing deployments for power efficiency. Additionally, we expanded AI-driven decision-making in HLO with explainability mechanisms, increasing transparency in IE selection and orchestration decisions. To support these advancements, continuum models were extended, such as internal protobuf messaging, and topology descriptors (TOSCA) that incorporate these new parameters, ensuring a more intelligent, adaptive, and efficient orchestration framework.

Secure and controlled access to resources is critical, therefore, all processes related to aerOS Cybersecurity have been finalized and incorporated into the second version of the MVP to demonstrate their integration with the entire aerOS Meta-OS. As a result, the authentication, authorization, and access control capabilities of aerOS are deployed and showcased in MVPv2, demonstrating how users with different access rights can be effectively managed by the aerOS IAM system using the RBAC mechanism and the KrakenD secure gateway. The combination of these tools enables the blocking or allowing of access to aerOS APIs (e.g., NGSI-LD endpoints). For the second version of the MVP, regarding the aerOS secure API Gateway, additional functionalities were introduced over time as different components required extra endpoints. Moreover, IAM facilitates authorized access to the aerOS Management Portal (for more details, refer to D4.3), ensuring restricted access to different domains within the portal based on user roles and groups. Since the initial version of the MVP, which was iterated on throughout the project, these roles and groups have been updated based on project needs. Consequently, access to the registered resources and functionalities within the Management Portal is tightly regulated and only allowed for authorized users. While the above features deal with securing the aerOS runtime environment, aerOS development team has also put in place a foundational toolset to support the development and secure integration of components, around the GitLab platform depicted from task T2.4 with pipelines for secure code development and continuous integration. Although, this toolset is not incorporated into MVPv2, it provides valuable functionality for securely developing and deploying aerOS components.

Also, in the MVP, the most relevant self-* modules (those that live and act within the scope of a single IE) have continued to be developed, improved, integrated and tested. It is worth mentioning that only a sub-set of the self-* modules were incorporated into the MVPv2 due to functional reasons (some of the components have very specific purposes to respond to particular cases that were not replicated in the demonstration flows of MVPv2).

The self-awareness component has been included in MVPv2, as it automates the process of publishing IE capabilities and updating, on real-time, running IE availability. For this new version of the MVP, the amount of information that can be extracted from each IE in the continuum has been increased, the integration with the rest of the self-* components has been improved, and the possibility of modifying the data sampling frequency (via REST API) has been added to optimize the IE resource consumption. The information obtained by the self-awareness module 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. The self-orchestrator module has also been conveniently updated, increasing the amount and variety of information capable of managing its rules engine or reducing its resource consumption, among other new features. This component is in charge of sending re-orchestration alerts to the HLO as and when necessary.

For protection and mitigation of IE intrusion events, self-security has improved its threat detection capabilities and is now able to detect a wider variety of attacks towards an IE. Not only the MVPv1 self-* module integrations have been reinstated and improved, but more self-* capabilities have been included. It is now possible to handle critical situations in a more elegant, faster and efficient way. Based on the objective of providing a complete set of functionalities, self-optimization and adaptation has also been developed to detect anomalies based on the information received by self-awareness, and optimize IE resources by requesting early re-orchestration to the self-orchestrator to avoid a saturation in IE resources. Self-healing has also been planned and developed, to detect failures in both the IE and IoT devices connected to the node, through different scenarios. The self-API (another implemented module) allows the connection with the internal APIs of the self-* modules used. This results in a robust result that can be deployed in aerOS-compatible continuums.

MVPv2 has served as the primary environment for validating the stability of architectural concepts and evaluating the viability and synergy of components. The aerOS development team, composed of numerous technical partners, has worked collaboratively to deliver seamlessly integrated components. While development adhered to specifications, contracts, APIs, and data model definitions, MVPv2 provided the necessary deployment environment for verifying the interaction and interworking of these components as an integrated system. This environment has been instrumental in ensuring the seamless operation of aerOS as a unified platform.

MVPv2 development has been the basis to plan and execute an enhanced demonstrator, with a detailed scenario capable to showcase the project's advancements and highlighting its capabilities in realistic scenarios. aerOS's iterative development phase has been critical, incorporating vertical stakeholders' feedback to enhance existing features, resolve issues, and introduce new functionalities guided by real-world demand.

The above-mentioned outcomes are diverse and stem from distinct domains of expertise. It is the integration of these components into MVPv2 that speaks about project's progress and refinement and ensures the identification of tasks addressed at this stage. MVPv2 integrates, validates and concludes technical milestones within the project lifecycle. In the initial design phase, the MVPv1 guided the prioritization of developments by defining the minimal set of features required to make aerOS viable for its first set of internal users and capable of demonstrating its foundational vision and functionalities. Subsequently, MVPv2 has built upon the initial insights, refining the architecture concepts, introducing enhanced capabilities, and expanding functionalities based on user feedback and evolving technical requirements.

At this final stage, MVPv2 serves as the foundation for deploying the aerOS stack across project pilot use cases. The validation of core functionalities and system stability has paved the way for replicating aerOS deployments in the five pilot locations. These pilots, while not all requiring the full stack, have selected and are currently deploying the services most relevant to their specific needs and vertical domain purposes (for more detail refer to deliverable D5.3). MVPv2 facilitates this selective deployment by enabling stakeholders to understand the functionalities provided and choose those that align with their requirements.

It is important, at this point, to also mention aerOS DevPrivSecOps platform as a critical enabler of MVPv2's success. The aerOS development lifecycle is managed through an on-premises GitLab platform hosted by UPV (https://gitlab.aeros-project.eu). This GitLab platform not only provided a unified development environment but also ensures that every iteration, enhancement, and refinement to MVPv2 is systematically documented and version controlled. GitLab streamlines workflows from coding and testing to deployment, allowing real-time tracking of MVPv2's evolution.

The software accompanying this deliverable, hosted in aerOS GitLab, is fully demonstrable within the aerOS MVPv2.

3. Final Implementation

3.1. Advancements in Smart networking for Infrastructure Element connectivity

3.1.1. Updated description and main functionalities

aerOS is designed as a Meta-OS, establishing the continuum as a network of interconnected aerOS domains. Each domain is equipped with the same capabilities and is itself a network of connected IEs. Building around this concept and encompassing a design which does not include components with a central controlling role, with single point of presence, any domain can be self-contained and additionally can be easily integrated as a peer in the continuum. This topology is reflected in aerOS networking. 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.

Task T3.1 aims to establish a fully functional network and compute fabric from edge to cloud. It ensures connectivity for IEs, allowing them to register as part of the continuum and execute specific workloads during IoT service deployment. Furthermore, WP3 lays the foundation for securely connecting domains and building a federated ecosystem. At the same time, it enables overlay communication for service components orchestrated across IEs in different administrative or geographical domains.

The research and development within WP3 were responsible for fulfilling these capabilities. As outlined in architectural deliverables (D2.6 and D2.7), IEs publish their capabilities and offer computational resources by integrating into administrative domains. These domains share a common set of core functionalities, and are known as aerOS domains. The capability to network IEs within each domain and abstract their cross-domain connectivity forms the foundational underlay required to establish the continuum. Ensuring programmable but at the same time secure and controlled networking capabilities is crucial, particularly when operating over public networks.

Basic network functionalities were part of previous developments. Building on these capabilities, IEs could connect within aerOS domains, which securely exposed services. These domains integrated a networking pipeline to control access, expose endpoints, and route requests to orchestration and federation services. Orchestration decisions could securely route deployment requests to selected IEs across all aerOS domains within the continuum. Additionally, IoT data could be federated across domains, enabling on-demand consumption throughout the continuum. This functionality allowed the creation of applications capable of interacting with each other by sharing application data seamlessly on top of data fabric federation capabilities. These federation capabilities were enabled with the support of the network stack for aerOS services exposure as described in D3.2. The period from M18 to M30, new advanced networking features were introduced, including isolated network overlays establishment for direct IoT service chaining, enabling real-time communication, when smart orchestration decisions dictate services placement on different IEs (thus, domains) across the continuum. Additionally, advancements and integration with tools and technologies based on open standards are developed and integrated with the aim to provide open exposure of aerOS capabilities to third parties.

These, along with the refinement of network functions required to ensure secure exposure, load balancing, and access to federated orchestration services across domains. The solutions developed go beyond simply achieving network connectivity; they establish a robust framework for integrating technologies capable of dynamically adapting network parameters, enabling programmability of network functionalities, and supporting performance monitoring. These advancements build upon tools and technologies that emphasize the separation of control and data planes, ensuring scalability and flexibility.

At a higher level of abstraction, workloads operating over physical or virtual resources, i.e., IEs, are designed to remain agnostic of the underlying virtual networking infrastructure while addressing their own connectivity requirements, including policies, security, load balancing, and other critical aspects. The research and development conducted during this phase have gone beyond creating a connectivity layer for IEs, extending to the connectivity of workloads distributed across different aerOS domains located across the continuum,

supported by the establishment of virtual overlays. This foundational layer integrates abstractions and automation typical of Software-Defined Networking (SDN), providing seamless and adaptive integration of resources. By doing so, it ensures that aerOS can effectively support heterogeneous and distributed infrastructures, aligning with its overarching vision of a unified continuum environment.

Task T3.1 efforts have been organized across seven key research lines:

1. Smart networking within the K8s context
2. Intra-domain network service mesh
3. Inter-domain network service mesh
4. Integration of Network Service Mesh with Service Mesh
5. Synergy between Network Service Mesh and SDN
6. Combining Network Service Mesh and NFV
7. TSN support for the aerOS continuum

By month M30, development efforts have further concentrated on tools and technologies that span multiple research lines, aiming to refine and consolidate the innovations made thus far. It is worth noting that while all research lines have progressed, their level of focus and advancement has varied depending on priorities and the complexities of integration. Significant emphasis has been placed on ensuring seamless integration of isolated overlays for cross-domain networking needed to support orchestration process decisions. This ensures a seamless interoperability between the network service mesh, SDN, NFV concepts and the overall aerOS architecture. In the following sections, the progress done within this final reporting period, for T3.1, regarding networking functionalities, and topologies is presented.

3.1.1.1. Network mesh for real time cross-domain service communication

As explained above and detailed in D3.2, a full network stack was developed by M18, providing core aerOS networking capabilities. Building on this foundation and the accessibility of aerOS services, application data can now be shared using the federation functionality of the aerOS data fabric. This enables data produced in one domain to be seamlessly shared and retrieved by consumer applications anywhere in the continuum via aerOS federation. In this final period, a prominent goal for aerOS networking was to enable service components—part of the same service deployment request but allocated across different IEs in multiple aerOS domains—to resolve and securely access each other directly, regardless of their location across the continuum.

This functionality was implemented during this period based on a new functionality which provisions for the deployment of an isolated overlay for each service. These overlays span from IE to IE, across remote domains, but not fully routing IE to IE, but just connecting service components of the allocated service. This means that IEs are not part of this overlay but just the hosted service components. So, an IE can host several service components, constituents of different services each of them, and these service components can be part of different, distinct and isolated overlays. Thus, the hosting IE is not itself part of any overlay, it is just networked within the aerOS domain, but several workloads hosted in this IE can be, each one of them, part of different overlays. Beyond direct connectivity, among workloads, the overlay also provides essential network services such as secure and private networking and DNS resolution.

To explain this, one can consider a scenario of deployment of a service, initiated from aerOS portal, which needs to perform end-to-end network performance testing, analyse the results, and visualize them. This could require five service components: a server, a client, an orchestrator component to initiate procedures and collect results, a time series database (TSDB), and an analytics function. While the client and server must be positioned at opposite ends of the network for measurement, the remaining components can be placed anywhere in the continuum. However, they all need to resolve and securely communicate with one another. For instance, the orchestrator must be able to instruct the client to generate traffic, the client must reach the server, the orchestrator must retrieve results from the server and push them to the TSDB, and the analytics component must access and process the data. While such interactions are straightforward within a local network, when service components are distributed across the continuum and deployed over different aerOS domains, additional provisioning and a complex support process are required.

This functionality has now been fully developed and integrated as part of aerOS networking capabilities. The provisioning of such an isolated overlay is managed by the aerOS HLO but also involves multiple other aerOS entities. Once the HLO determines the optimal placement for all service components across the continuum, an additional step is performed: establishing an overlay network exclusively for that service. The two main provisions of this networking functionality are **secure connections**—enabled by the domain WireGuard server—and **name resolution**—provided by the accompanying Dnsmasq server. WireGuard and Dnsmasq are part of the networking stack in every aerOS domain, as described in D3.2. While WireGuard was already integrated, Dnsmasq was recently introduced to support this new functionality. These two components operate as a bundle. Although each aerOS domain includes such a bundle, the one responsible for providing the overlay for a deployed service is the domain that initially received the deployment request via its exposed HLO API.

To develop this capability the following objects have been studied as information included in them needs to be aligned within aerOS:

- WireGuard server configuration, which defines the parameters required for the server to expose connectivity, authenticate and assign IP addresses to clients, and correctly forward and masquerade traffic across the overlay.

```
[Interface]  Server configuration
Address = 10.13.13.1/24, 10.13.0.1/24
ListenPort = 51820
PrivateKey = UE+7i[redacted]

PostUp = iptables -A FORWARD -i wg0 -j ACCEPT; iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT; iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE

[Peer] # peer 1    Peers identification
PublicKey = +zbqPIaf7nWD8oyQeJTdeFZdZgtGRwZpCzqDVPCmgD4=
AllowedIPs = 10.13.0.2/32
```

Figure 3. Wireguard server configuration in aerOS

- WireGuard client configuration, which includes the necessary parameters for establishing secure connectivity, such as the private key for encryption and authentication, the server's URL and port, and the assigned IP address within the overlay network.

```
[Interface]  Client configuration
Address = 10.13.13.2/32
PrivateKey = wCo4TE8dGyzBC9XADw0b9vaeO/[redacted]
ListenPort = 51820
MTU = 1360

[Peer]  Server information section
PublicKey = w6r8RJaMOkGxdhXkR29jE/MNh5ivgZtfbgXWX/Y4xj0=
Endpoint =  ncsrd-dev-domain.aeros-project.eu:51820
AllowedIPs = 10.220.0.0/16, 172.25.0.0/16, 10.160.1.0/24
persistentKeepalive = 25
```

Figure 4. Wireguard client configuration in aerOS

- Dnsmasq server configuration, which holds all the mapping of names to overlay IPs

```
server=8.8.8.8
                dns registries
address=/peer1-name/10.13.0.2
address=/peer2-name/10.13.0.3
```

Figure 5. Dnsmasq server configuration in aerOS

To establish an overlay per service—where service components can securely connect and resolve one another—the necessary configuration objects must be generated during the orchestration process. The required information should either be available within the continuum or dynamically created as needed. Finally, these

configuration objects must be properly assigned to the respective services to ensure seamless operation. The following aerOS components have been extended, modified, or developed to manage and utilize this information effectively:

- **aerOS Domain Private and Public Key**: The private key is used in the server configuration object, while the public key is shared with all clients that need to connect. These keys are generated once per domain (with the option to regenerate if necessary) by an initialization script running within a core aerOS service. This service exposes internally API to manage domain keys. These keys are securely stored within the domain, such as in a vault or a K8s secret when the domain is based on a Kubernetes cluster. Additionally, the public key is included as an attribute of the "Domain" continuum entity, enabling its retrieval across the continuum for seamless WireGuard client configuration.

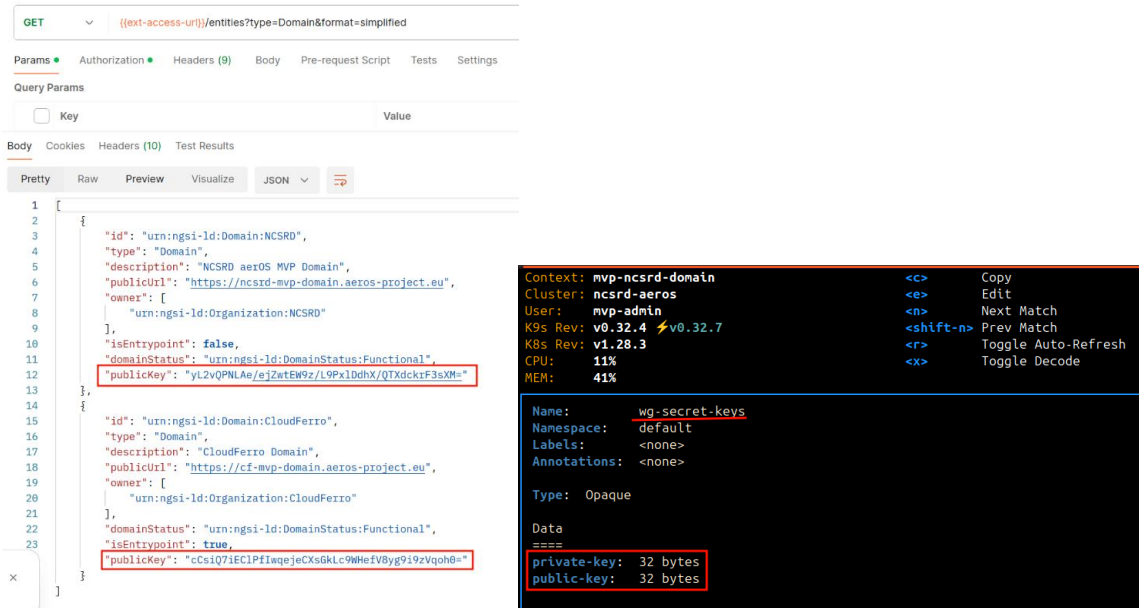


Figure 6. Left: aerOS continuum "Domain" entity including public key. Right: secret with private key.

- **aerOS continuum "Service" entity,** is extended to include the information of the domain that hosts the overlay server. The name provided to this attribute is "domainHandler" (see figure below).

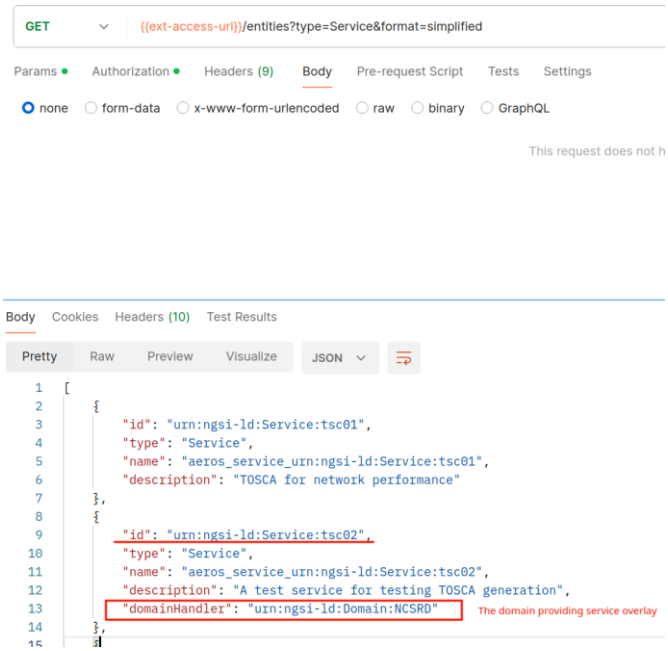


Figure 7. aerOS continuum "Service" entity including information of domain providing the overlay service

- **HLO deployment engine,** has been massively extended so that along with the allocation requests (see D3.2) it performs two more actions:
  - Locally reconfigure WireGuard and Dnsmasq server and provide all information for the clients that will be connected (allocated service components)
  - Built configuration objects, which will be sent to the domains which host the selected IEs where service components will be allocated, that will provide connectivity to the overlay to the service components.
  - Private-public keys pair generation per each service component, as these are part of above-mentioned configuration objects, which are integrated in the private key on service component configuration and the public in server configuration.
- **LLO**, is extended to be able to handle information about the overlay and proceed to force workload to also perform connectivity handshake using the client configuration object provided by the HLO.

The figure below is part of the logs of the aerOS orchestrator and exposes this information which is used to configure connectivity in the overlay.

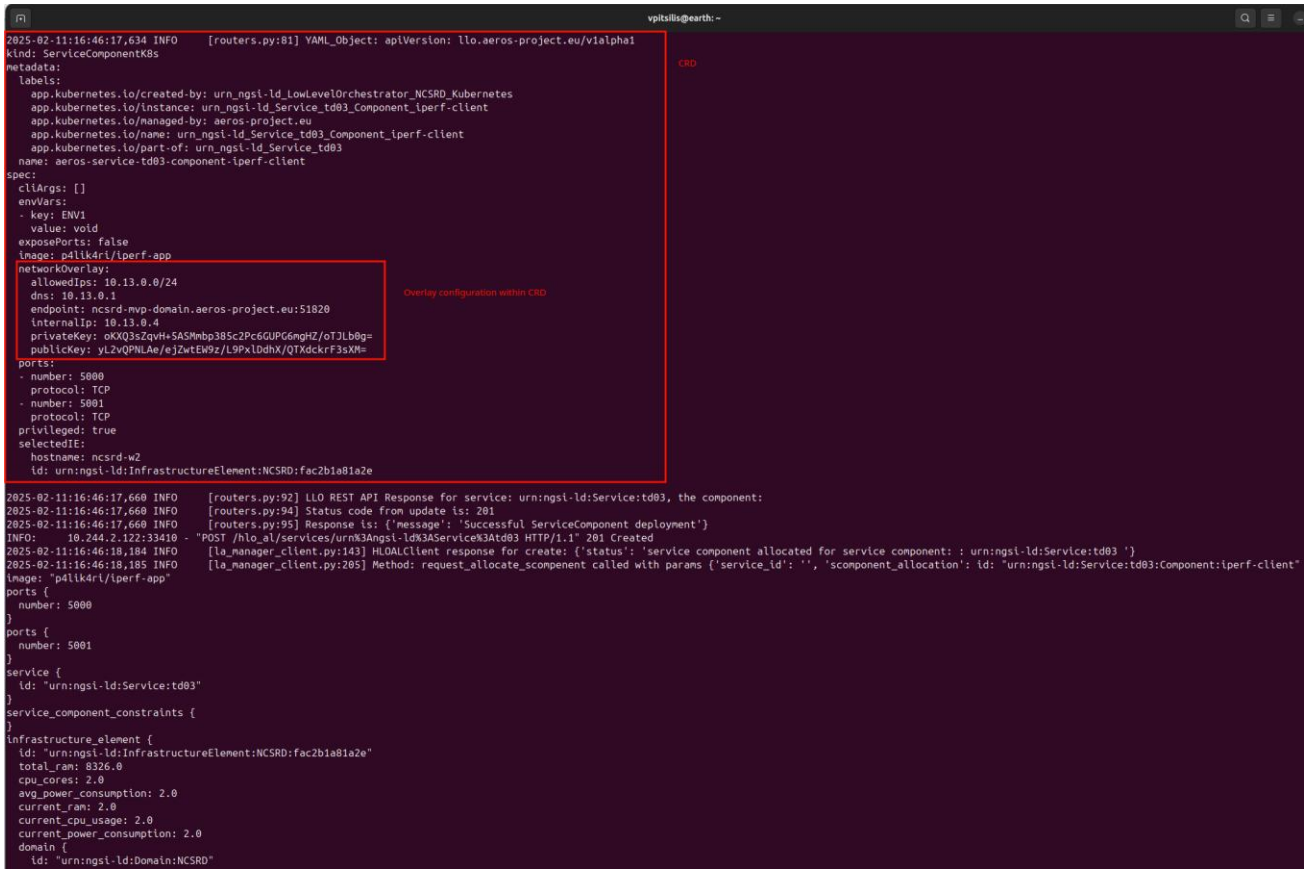


Figure 8. Object of the orchestration process including information for connecting to the networking overlay

The figure above illustrates how the allocation object sent to the LLO has been extended to include networking information. It clearly shows the integration of the previously discussed WireGuard client configuration object, which is now included and transmitted to the LLO. This configuration is then utilized during workload deployment to establish secure connectivity to the remote server using the provided URL, port, and key. The following figures demonstrate the orchestration results of a service comprising four service components, specifically highlighting the overlay establishment. Notably, the names of the service components are defined within the TOSCA-formatted deployment request sent to the HLO API.

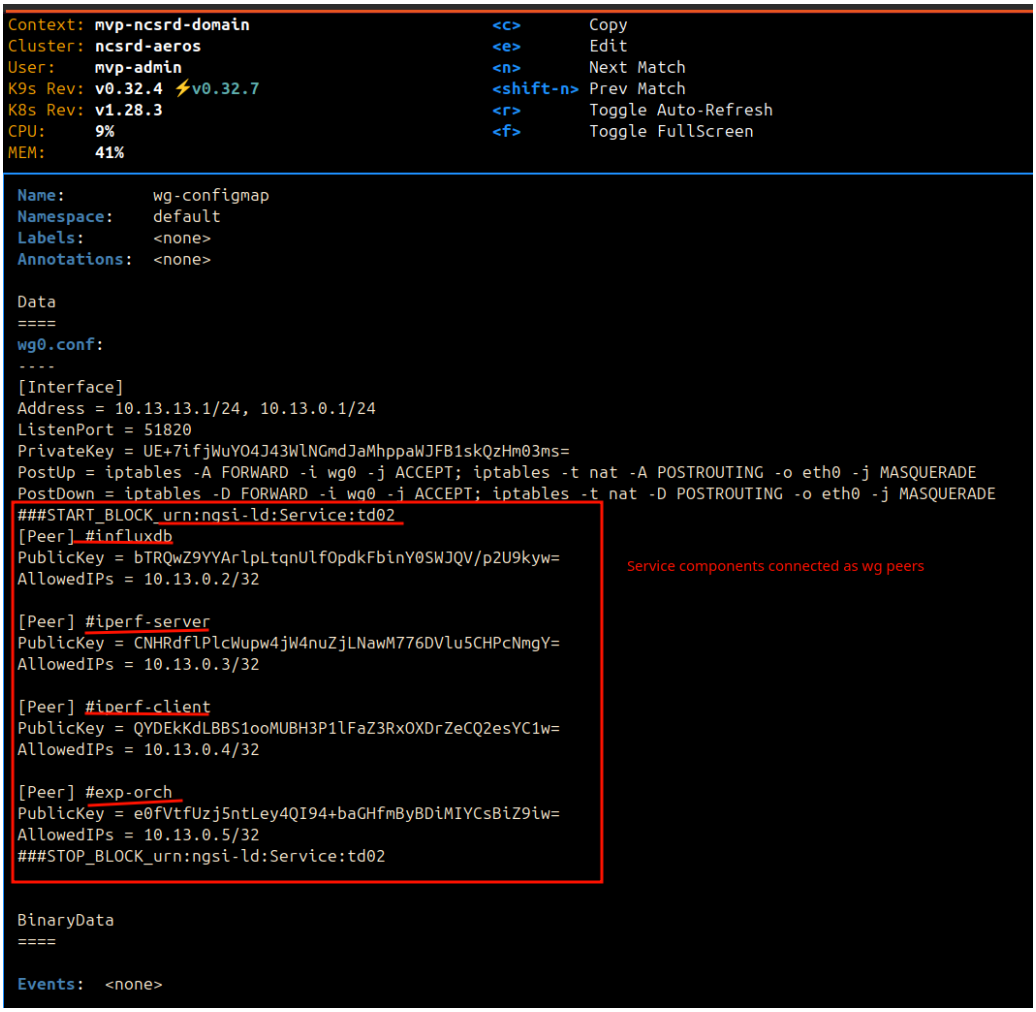


Figure 9. Server configuration object, including clients' information

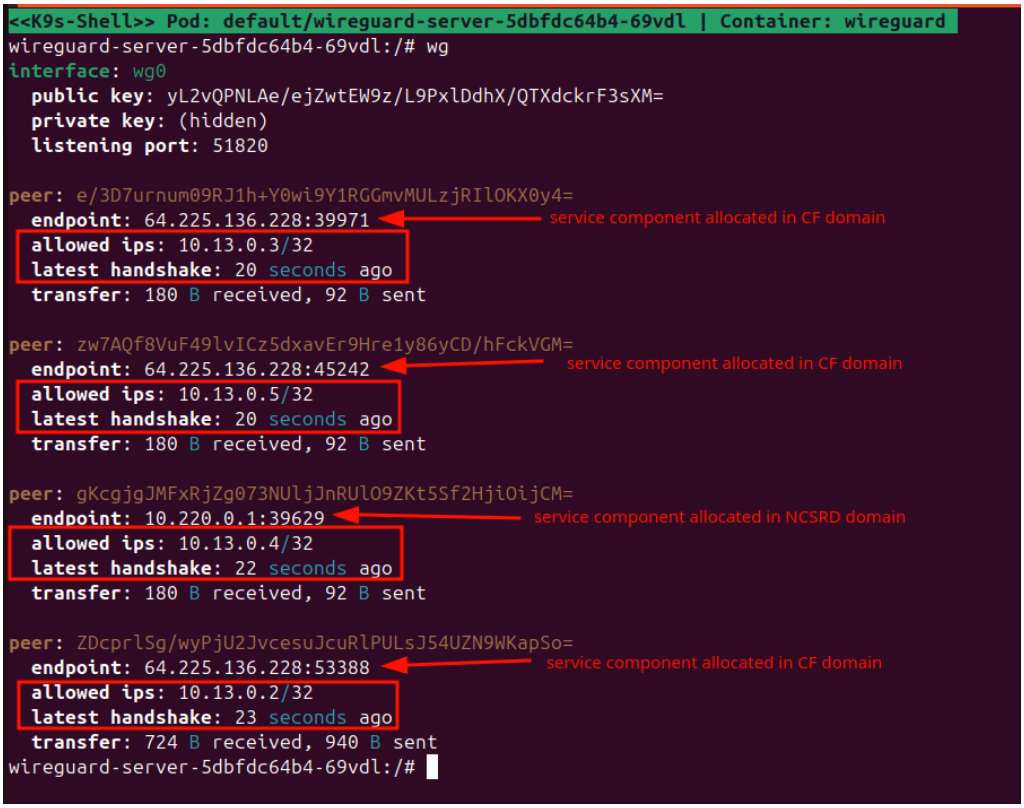


Figure 10. Service components connected to the overlay. Data from WireGuard server shell

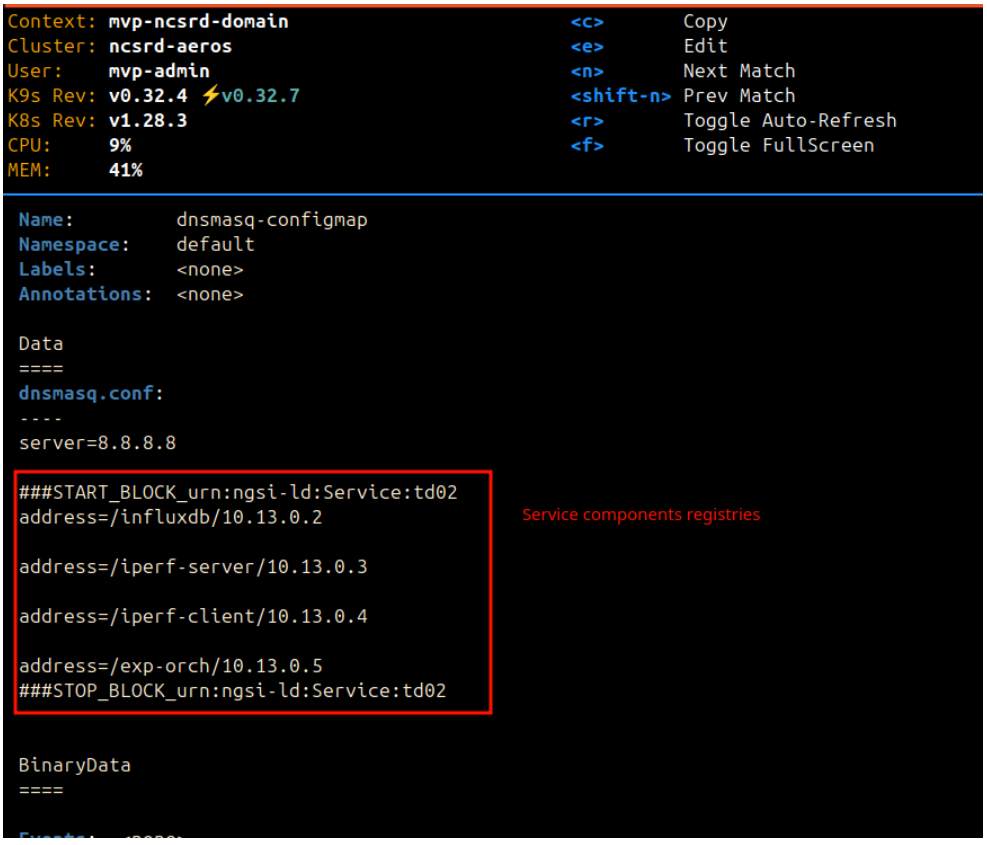


Figure 11. Dnsmasq configuration associating service components with service name

```
root@aeros-service-td03-component-iperf-client-65c57474cf-z5m77:/app# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.244.2.35  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::5c33:faff:fe56:2367  prefixlen 64  scopeid 0x20<link>
        ether 5e:33:fa:56:23:67  txqueuelen 1000  (Ethernet)
        RX packets 51  bytes 4604 (4.4 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 53  bytes 4264 (4.1 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
                                                    wg interface
wg0: flags=209<UP,POINTOPOINT,RUNNING,NOARP>  mtu 1420
        inet 10.13.0.4  netmask 255.255.255.255  destination 10.13.0.4
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000  (UNSPEC)
        RX packets 3  bytes 276 (276.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19  bytes 956 (956.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@aeros-service-td03-component-iperf-client-65c57474cf-z5m77:/app# ping iperf-server
PING iperf-server (10.13.0.3) 56(84) bytes of data.
64 bytes from 10.13.0.3 (10.13.0.3): icmp_seq=1 ttl=63 time=71.6 ms
64 bytes from 10.13.0.3 (10.13.0.3): icmp_seq=2 ttl=63 time=64.7 ms
64 bytes from 10.13.0.3 (10.13.0.3): icmp_seq=3 ttl=63 time=64.6 ms
^C
--- iperf-server ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 64.599/66.988/71.647/3.294 ms
root@aeros-service-td03-component-iperf-client-65c57474cf-z5m77:/app# ping influxdb
PING influxdb (10.13.0.2) 56(84) bytes of data.
64 bytes from 10.13.0.2 (10.13.0.2): icmp_seq=1 ttl=63 time=65.4 ms
64 bytes from 10.13.0.2 (10.13.0.2): icmp_seq=2 ttl=63 time=64.4 ms
64 bytes from 10.13.0.2 (10.13.0.2): icmp_seq=3 ttl=63 time=64.5 ms
^C
--- influxdb ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 64.386/64.738/65.370/0.447 ms
root@aeros-service-td03-component-iperf-client-65c57474cf-z5m77:/app# ping exp-orch
PING exp-orch (10.13.0.5) 56(84) bytes of data.
64 bytes from 10.13.0.5 (10.13.0.5): icmp_seq=1 ttl=63 time=64.5 ms
64 bytes from 10.13.0.5 (10.13.0.5): icmp_seq=2 ttl=63 time=64.7 ms
^C
--- exp-orch ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 64.521/64.608/64.696/0.087 ms
root@aeros-service-td03-component-iperf-client-65c57474cf-z5m77:/app#
```

Figure 12. Overlay connectivity and service names resolution. Network operations within a service component shell

A quite abstract representation of the final overlay and the layered nature of aerOS networking is demonstrated in the following figure. It demonstrates the fact, also mentioned above, that although each domains hosts a WireGuard server, for each new service deployed the one that is instrumented to provide the overlay, over the continuum, is the one hosted in the aerOS domain which received the orchestration request.

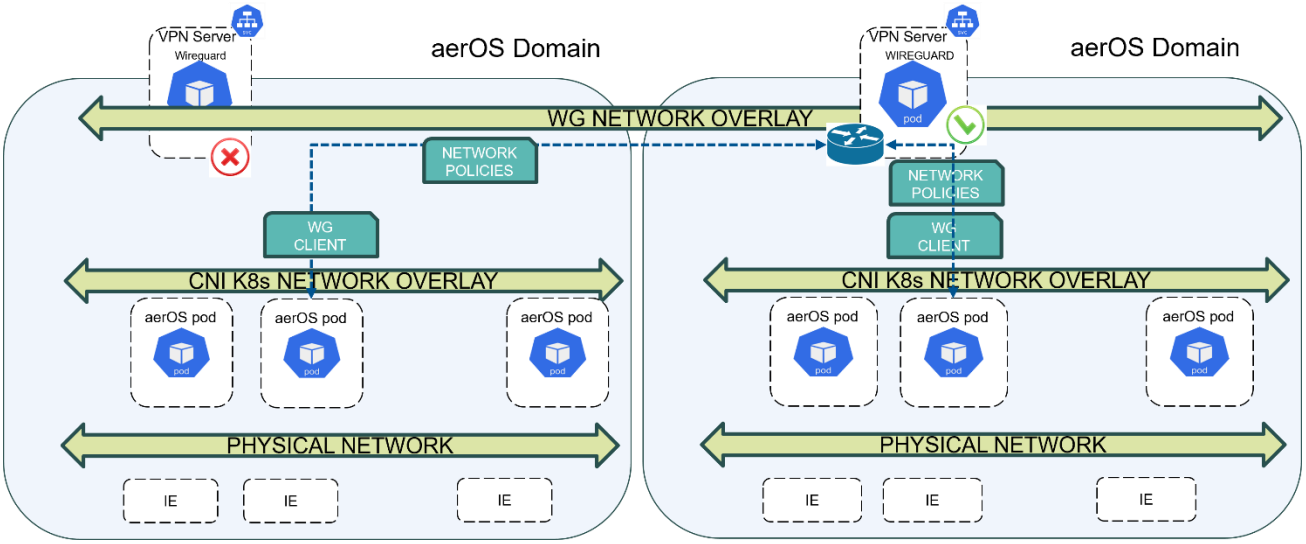


Figure 13. aerOS overlay diagram.

Finally, a sequence flow of the allocation of an isolated overlay, as part of the service orchestration, is demonstrated below. This assumes the orchestration of two a service which includes two service components, and one is allocated in the same domain which received the request by the user, and which also hosts the WireGuard network server, and the other service component is allocated to another aerOS domain across the continuum.

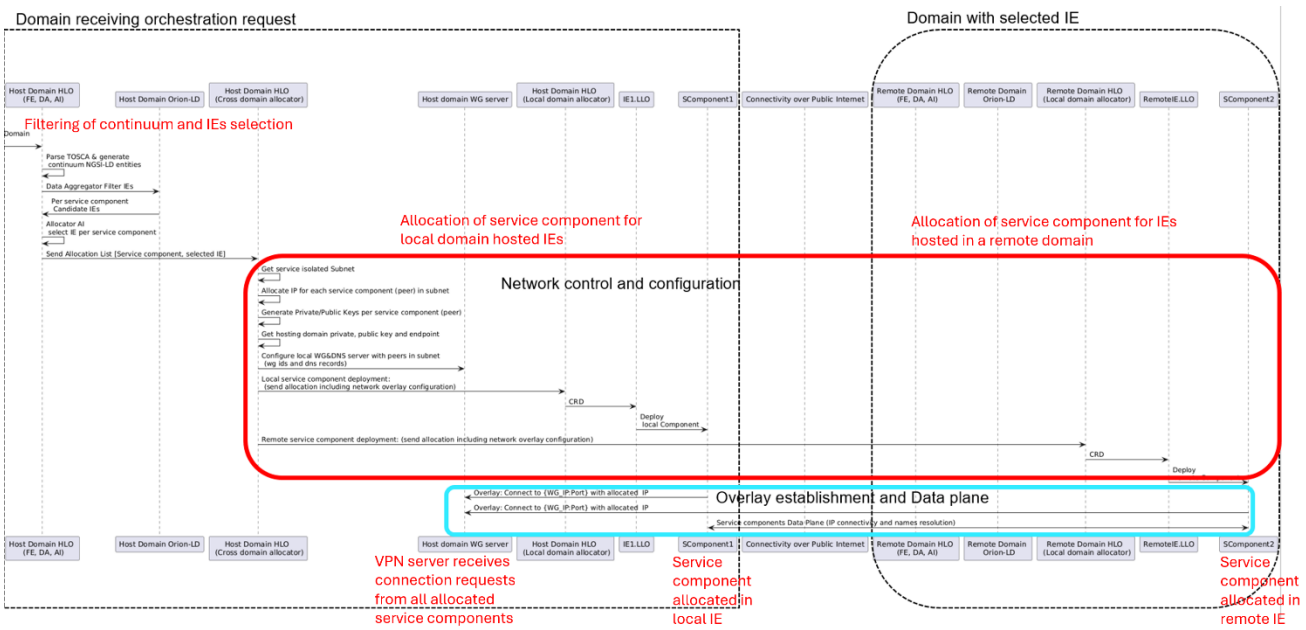


Figure 14. aerOS cross-domain overlay orchestration sequence flow detail.

3.1.1.2. Open Network Exposure for Standardized API Access

To enhance the functionality and interoperability of aerOS orchestration and federation framework across the cloud-edge continuum, a new component has been developed to expose aerOS APIs with the **3GPP Common API Framework for IoT (CAPIF)**. This integration aligns with industry standards and unlocks the potential of integrating aerOS into modern 5G and telecom ecosystems. CAPIF provides a standardized approach to API exposure, ensuring compatibility and seamless collaboration with external systems, devices, and third-party services within a globally recognized and regulated framework. This effort is part of our broader goal to bridge cloud-native networking with traditional telecom infrastructure, ensuring compatibility with future NFV and telecom standards.

CAPIF provides support for **secure API exposure**. Orchestration and federation involve critical operations such as resource allocation, workload management, and policy enforcement, and CAPIF enables **dynamic API discovery and management**, making it easier for clients, such as edge devices or third-party applications, to locate and interact with aerOS APIs.

CAPIF's adoption offers API discovery mechanisms, which can ensure aerOS services are accessible and usable in a dynamic, multi-vendor ecosystem, aligning thus aerOS framework with the emerging **5G and Network-as-a-Service (NaaS)** models. By exposing our APIs through CAPIF, telecom operators and service providers may leverage aerOS orchestration and federation capabilities directly within their 5G environments. This opens up opportunities for advanced use cases such as **network slicing, edge resource orchestration**, and **IoT data federation**, positioning possibly aerOS within the telecom ecosystem. To implement the exposure of aerOS APIs under the 3GPP CAPIF specification, OpenCAPIF was utilized. Developed by ETSI, the OpenCAPIF initiative extends the applicability of CAPIF beyond the telecom domain, addressing broader industry requirements. This strategic effort opens new possibilities for aerOS framework's role as a key player in the evolving landscape of cloud-edge continuum and 5G innovation.

3.1.1.3. aerOS programable networking

The aerOS networking capabilities are designed to be self-contained, enabling seamless integration of advanced SDN functionalities within the broader system. During this period a component which can intermediate the interaction with external network services or infrastructure has been developed in the grounds of aerOS auxiliary networking functionalities. Although this is not a core component and is not intended to be part of the MVPv2 it can find use in configuring networking behaviour of hosting domains as it is built to communicate specific SDN controllers (ONOS tested), providing the necessary OpenFlow commands to manage connected switches and ensure intended traffic control throughout the network.

This service is developed as a cloud native application, the **ONOS Flow Manager**, which is a Python-based tool that facilitates interaction with the ONOS SDN controller and allows for the dynamic deployment of OpenFlow rules to Open vSwitch (OVS) devices. The script serves as the backbone for configuring traffic-forwarding policies, managing flows, and applying packet filters. It offers flexibility by enabling administrators to define criteria such as source and destination IPs, MAC addresses, and output ports, ensuring granular control over network behaviour.

The service receives as input a list of parameters, including the ONOS controller access details, and a set of parameters which describe matching criteria, actions, and priority levels. Once validated, these specifications are translated into OpenFlow-compatible commands and sent to the ONOS controller using its REST API and support:

- Multiple flow filters (e.g., source IP, source MAC, destination IP, destination MAC).
- Action-based modifications (e.g., output port, destination IP, MAC).
- LLDP packet redirection to the controller.

Additionally, it has been adapted for compatibility with EAT to enable its triggering based on events recognized by analytics. The goal of integrating these advanced SDN capabilities, is to set the groundwork for a flexible and programmable networking environment. This component represents an effort towards achieving a continuum-aware network fabric, where external network services and infrastructure can be seamlessly integrated to support dynamic and scalable IoT deployments.

3.1.2. Updated Structure diagram

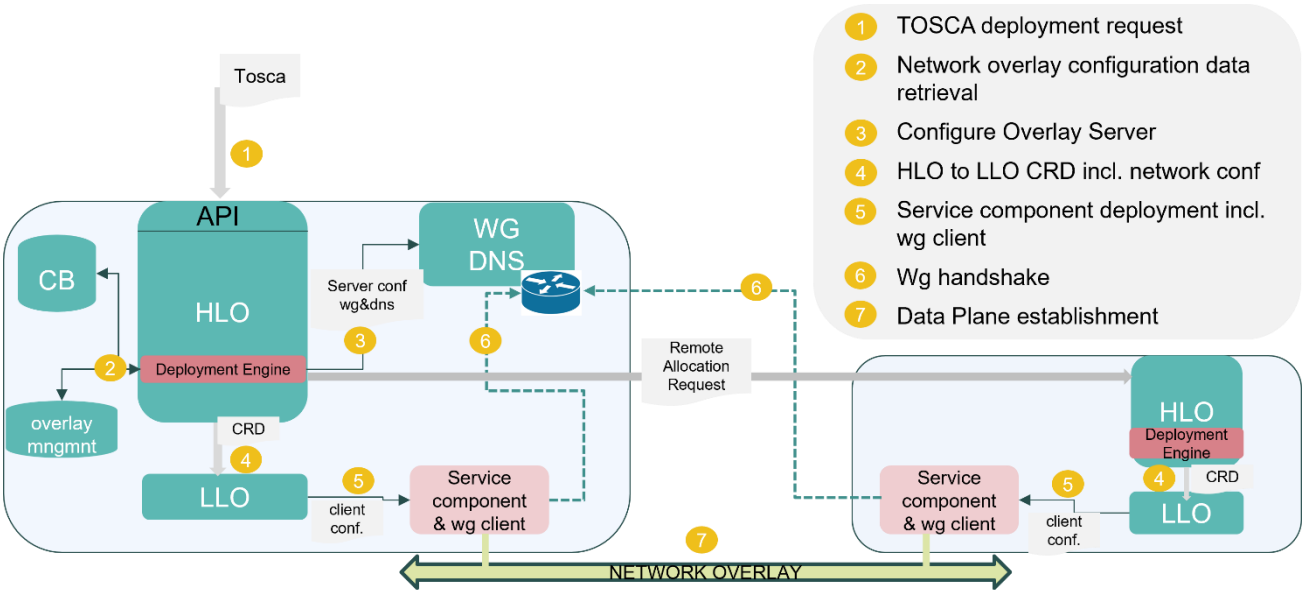


Figure 15. aerOS cross-domain network overlay provision during service orchestration.

Table 1. Network mesh for real time cross-domain service communication

Component	Description	Interactions
HLO Deployment Engine (Cross domain Allocator)	Located in aerOS domain which received the deployment request. Retrieve from Orion-LD, and generate data needed to build overlay configuration objects for server and client peers (service components). Reach domains of all selected IEs and send descriptors including networking data.	Proceeds to NGSI-LD queries to the **Orion-LD** broker and to API calls to **overlay management** component and **HLO deployment engines (local domain allocation part)**. All of these are REST based interactions, the access to the first two are in local domain and the third one could be in local or any other domain, which hosts the selected IE, across the continuum.
Overlay management	Service within aerOS domain which undertakes the management of networks available, and networks allocated within the domain. The range is configurable and can be set at service initiation, e.g., 10.13.0.0/16 means that it will provide a "slice" like 10.13.1.0/24 or 10.13.2.0/24 for each service which overlay hosting is provided by this domain.	Exposes API which is consumed by **HLO Deployment Engine (Cross domain Allocator)** to provide an available subnet.
HLO Deployment Engine (Local domain Allocator)	In each aerOS domain selected to host a service component. Based on descriptors received from the step above, creates the needed configuration objects, which include networking information, and reaches the suitable LLO for	Exposes, REST based, API which is consumed by the **HLO Deployment Engine (Cross domain Allocator)** of the domain which received the service deployment request by the user (IoT developer). It calls, REST based, **LLO API** to provide any CR needed for the deployment of the

	the IEs that will host each service component. It submits to the LLO the final descriptive resource needed by LLO to enforce service component instantiation including connectivity to the overlay.	service including connection to the networking overlay.
Orion-LD	Keeps part of the information needed for the networking configuration objects (public key and public IP port of the domain which will provide networking overlay). Keeps also information needed to track the domain that provides server for network overlay for each service. This is updated once the service is deployed.	Exposes NGSI-LD, a REST based API, to provide information regarding data needed to build the overlay connectivity.
LLO	Guides and enforces service components deployment in the selected IEs, including deployment of WireGuard client that connects to server and thus registers service component to the overlay.	Exposes, REST API which is accessed by **HLO Deployment Engine (Local domain Allocator)** to submit the deployment descriptor (CR) for the selected IE. Proceeds to deploy component to selected IE (API based interaction).
WireGuard Server	Provides secure overlay (VPN) and name resolution needed for service components to reach one another.	Receives configuration from **HLO Deployment Engine (Cross domain Allocator).** This is done in two steps. First is the update of the configuration objects and the second is access to an internal (to HLO) API which abstracts operations to WireGuard server and thus updates configuration and re-initiates the wireguard service. Receives handshake requests from wireguard clients, which accompany each service component. This interaction is based on WireGuard protocol.
WireGuard client	Runs as a complimentary service (side car) to each service component, proceeds to handshake to register to the network overlay and takes over all the network connectivity.	Deployed by LLO as part of the allocation of the service component on the selected IE and reaches to WireGuard server on the domain which hosts the service wireguard server to perform handshake and then route traffic. This interaction is based on wireguard protocol.

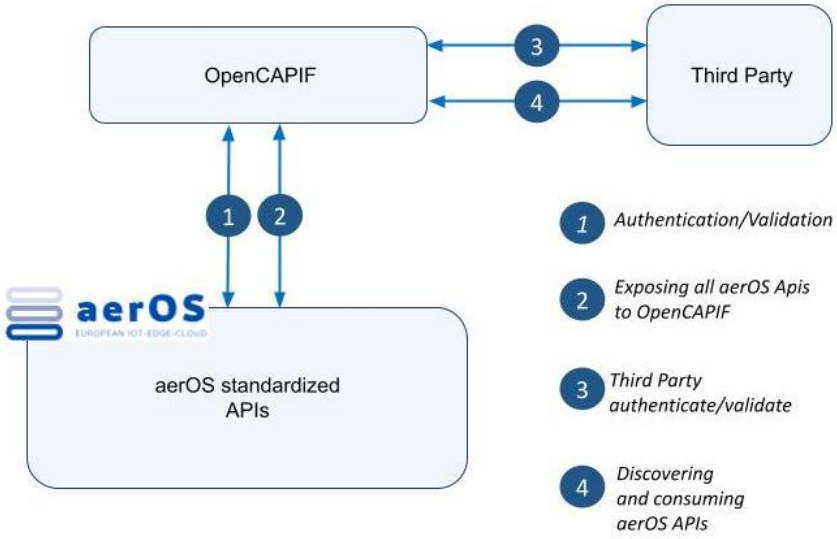


Figure 16. aerOS integration with OpenCAPIF.

Table 2. Open Network Exposure for Standardized API Access

Component	Description	Interactions
aerOS APIs	All APIs exposed by every aerOS domain, including APIs for orchestration services, for federation across the continuum.	**OpenCAPIF** registers these APIs and every **third-party application** which discovers aerOS APIs might access them. All interactions are REST based.
OpenCAPIF	Implementation of CAPIF specification, which provides APIs discoverability and exposure, and thus access to underlying offered services.	Interacts with **aerOS API** to register and expose it (REST). Interacts with **third-party** applications to expose aerOS APIs and provide the means for a subsequent direct secure connectivity.
Third-Party	An external application which would like to consume aerOS domain services as exposed by the APIs.	Interacts with **openCAPIF** to query APIs offered and get security keys for subsequent access. REST based interaction. Access **aerOS APIs**, which again is a REST based interaction.

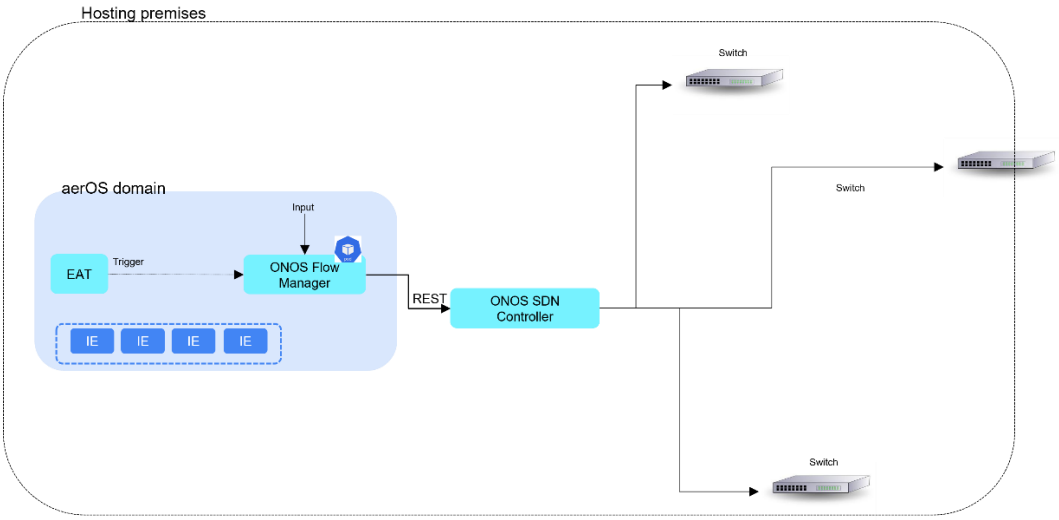


Figure 17. aerOS aux service for SDN.

Table 3. ONOS OpenFlow Manager

Component	Description	Interactions
ONOS Flow Manager	aerOS service capable of transforming, user or component provided, guidelines into ONOS commands, including network setup instructions, and sending them to an ONOS SDN controller.	Exposes REST API to actors who want to access it. Interacts with **ONOS SDN** using its exposed REST APIs
ONOS SDN Controller	SDN controller managing network operations. Processing high-level instructions, and translates them into flow rules, for a dynamic configuration of the network.	Receives network application requests from **ONOS FlowManager** (REST API). OpenFlow protocol-based interaction with managed **network switches**.
Network Switch	Enforce decisions received by ONOS flow rules enabling flexible and programmable network behaviour.	Interaction with ONOS SDN controller over OpenFlow protocol.

3.1.3. Technologies and standards deployed in MVP

Table 4. Technologies and standards for aerOS networking implementation

Technology/Standard	Description	Justification
Wireguard	Lightweight, high-performance VPN protocol that uses modern cryptography to provide secure, fast, and simple point-to-point encrypted network connections.	Addresses the requirement to build a secure, isolated subnet overlay which provides networking to all service components of a service.
dnsmasq	Lightweight DNS forwarder, DHCP server, and TFTP server designed for small networks, providing caching, name resolution, and IP address management with minimal resource usage.	Addresses the requirement of service components to be able to resolve and reach one another based on a defined name which corresponds to routable IP (within the service isolated overlay).
Curve2551	High-performance elliptic curve used for secure key exchange, offering strong cryptographic security, efficiency, and resistance to common attacks while enabling fast and secure encryption protocols like WireGuard and TLS 1.3.	Instrumented, with the support of wg tool (genkey option), to provide secure keys to be used for service components to handshake and register in network overlay.
ETSI OpenCAPIF	An open-source implementation of 3GPP CAPIF, developed within ETSI. It provides a reference implementation to facilitate CAPIF adoption, allowing developers and telecom operators to integrate CAPIF-	Provide the means to demonstrate for aerOS possibilities to integrate within telco operators' environments.

	compliant API exposure and management into their networks.	
3GPP CAPIF	A standardized API framework defined by 3GPP (TS 23.222) to provide a unified, secure, and controlled way for network functions and third-party applications to expose and consume APIs within 5G and telecom networks. It includes API exposure, authentication, authorization, and monitoring.	

3.2. Communication services and APIs

In the dynamic and interconnected world of cloud-to-edge computing, the role of communication services and APIs has become increasingly pivotal. These services and APIs are the cornerstones 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 data contracts. The next sub-chapter summarizes the API concepts, guidelines and best practices adopted in aerOS, which were thoroughly reported in D3.2. The following sub-chapters delve deeply into the provisioned APIs from the different core aerOS services (covering OpenAPI and AsyncAPI) and into low-code tools such as Node-RED and Behaviour trees and generation of skills from AsyncAPI for low-code tools.

3.2.1. API concepts, guidelines and best practices proposed in D3.2

In deliverable D3.2 it was reported the API concepts, guidelines, and best practices, particularly in the context of REST APIs. The main insights provided on that report are the following:

- The absence of standardized procedures for creating endpoints, encoding body payloads, or defining return codes for both successful and erroneous invocations was highlighted, emphasizing the need for use-case/domain-specific guidelines. The report referenced efforts by organizations like ETSI to establish principles for mobile edge services APIs and discussed how existing guidelines can be found across cloud providers and technical articles, although they often pivoted around common ideas but differed slightly in recommendations.
- Best practices for URI design, including the use of valid URIs following the IETF RFC 3986 standard was also outlined. These URIs should be combined with verbs representing HTTP methods and nouns for collections of objects, with plural names preferred in URIs. It also introduced pagination techniques to optimize resource access, suggested versioning methods to support multiple API versions, and detailed the use of HTTP status codes to indicate the outcome of client requests. Further, it advised including error details in API responses, modelled as JSON objects with properties like "error" and "description" to aid client-side error handling. Asynchronous operations were addressed, recommending the use of a "202 Accepted" status code for operations requiring longer processing time, alongside a status endpoint for clients to check operation status. Hypermedia As The Engine Of Application State, a technique using hypermedia links in response contents, fostering API evolution without client logic breaking, was also mentioned.
- The maturity in API design was discussed by making use of the Richardson Maturity Model, which evaluates the maturity of web services based on their adherence to REST principles. It comprises four levels:
    - Level 0, which involved basic service-oriented applications without using URIs or HTTP verbs
    - Level 1, introducing URI usage for resource access but not fully utilizing HTTP verbs
    - Level 2, achieving significant maturity by employing HTTP verbs and URIs

- Level 3, the highest level, incorporating HATEOAS to enhance discoverability and self-descriptiveness.

3.2.2. aerOS OpenAPI

aerOS commitment to foster a standardized approach in CEI continuum led project partners to embrace OpenAPI Specifications (OAS)[1], 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. There are two OpenAPI design methodologies: API First (first creating the OAS, and then create the code), or Code First (first writing code and annotating it to automatically generate the OAS). The aerOS OpenAPI lifecycle was structured in phases, beginning with **requirements elicitation**, where the desired functionalities of the API for its consumers are defined. This moves into the **design** phase, where an initial OAS document is outlined, incorporating industry-standard schemas and allowing for rigorous source control as a preparatory step for development. During **configuration**, the focus shifts to adapting the IT infrastructure to accommodate the API's needs, such as gateways or security requirements. The **publishing** phase then follows, which involves generating API documentation using tools like Swagger UI to be hosted on a basic HTML server for easy access. **Development** translates the OAS into a functional API, with tools available across programming languages to construct essential API structures. The **testing** phase leverages the OAS to verify the consistency and security of the API implementation, ensuring alignment with the initial design contracts. Finally, **deployment** integrates output from the publishing and development processes to roll out the fully tested API to end-users, marking its readiness for real-world application. In D3.2 it was informed about two methodologies for OpenAPI code generation, and the openapi-generator run locally was chosen for its broader applicability.

Task T3.2 has focused on the development and preparation of OpenAPI specifications for both the aerOS domain and the Infrastructure Elements (IEs), as can be seen in the next figure.

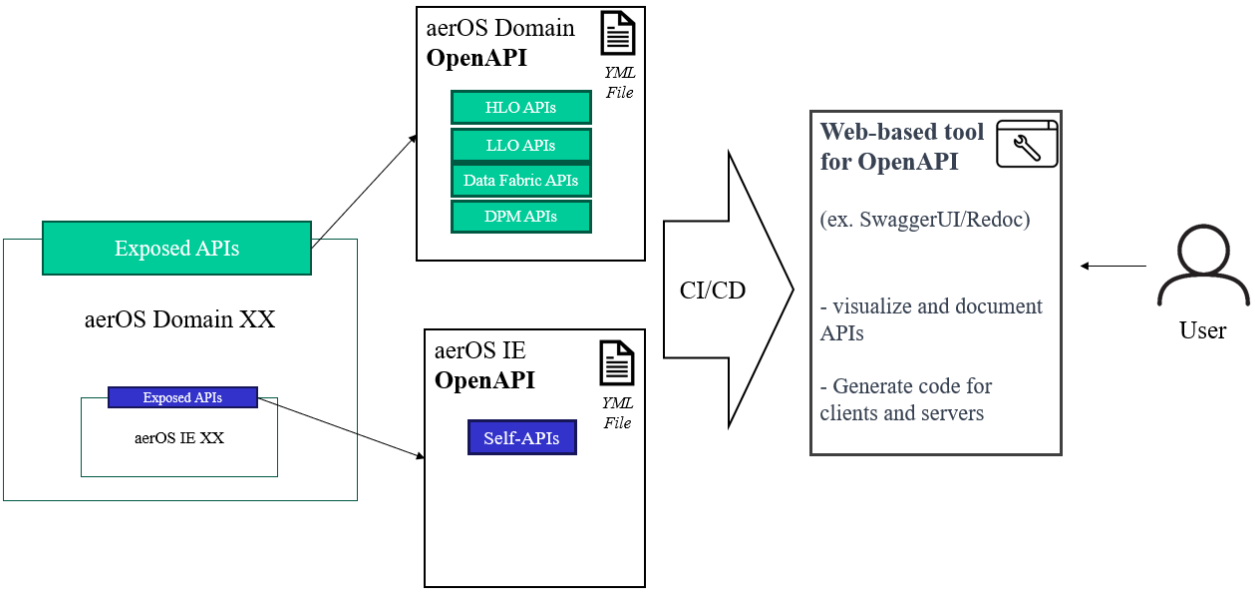


Figure 18. aerOS communications and services through OpenAPI

In that sense, core/basic aerOS services have exposed their APIs via OpenAPI specifications. Task 3.2 has collected and consolidated all individual specs to provide a unified interface with all declared methods. The following sections briefly introduce them.

1 https://www.openapis.org

3.2.2.1. Context Broker API

This aerOS contextual information is managed by the Context Brokers, which store the most recent value of the attributes of NGSI-LD entities of the continuum, i.e., IE, services, etc. The NGSI-LD Context Broker choice for aerOS has been the FIWARE Orion-LD[2]. It contains an official repository of ORION-LD API that provides comprehensive documentation of the API[3]. Regarding aerOS approach, there are 4 main sections or paths to be considered: Context Information Provision, Context Information Consumption, Context Information Subscription, and Context Source Registration Subscription.

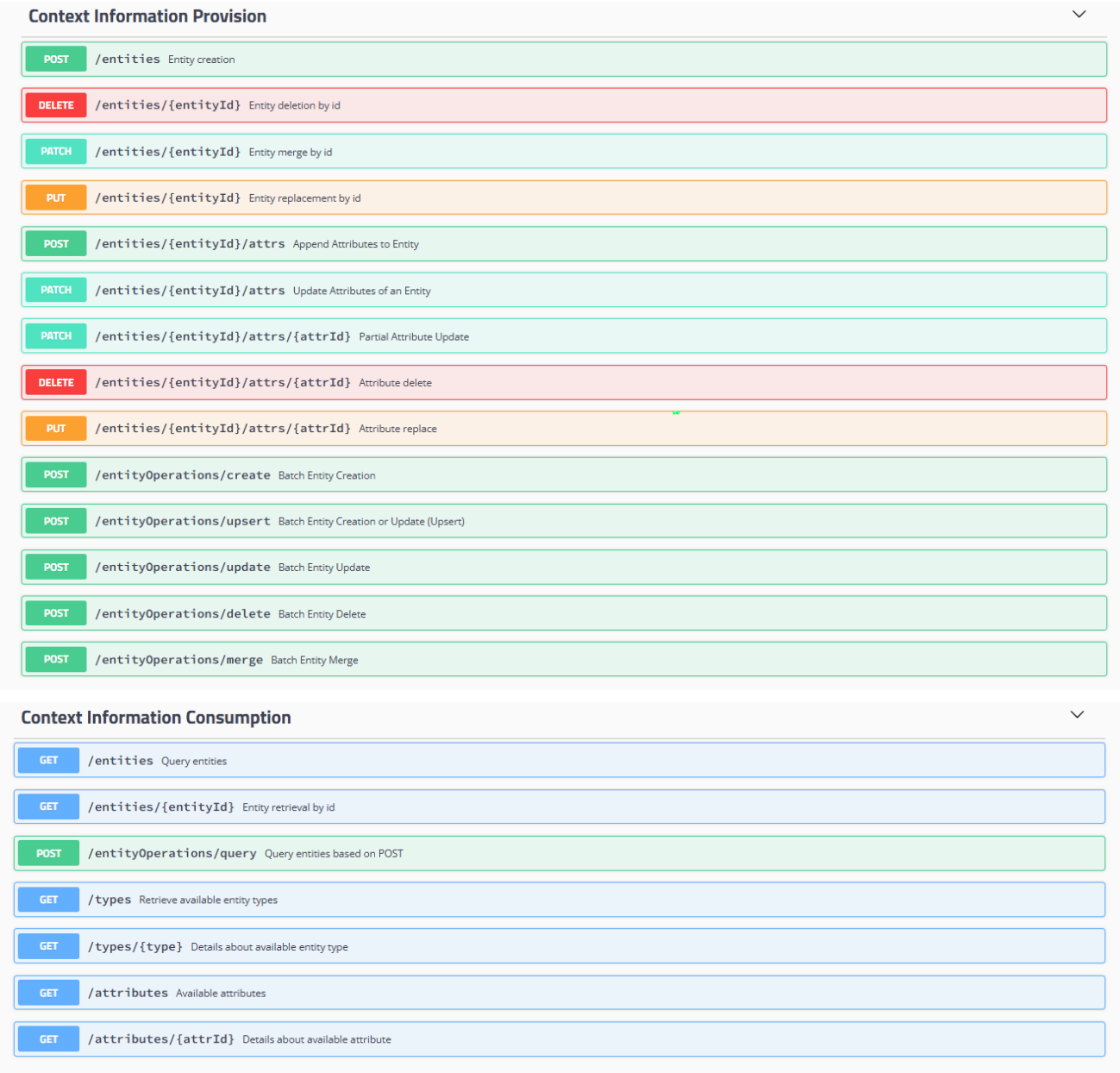


Figure 19. ContextBroker API inside aerOS OpenAPI

2 https://github.com/FIWARE/context.Orion-LD

3 https://forge.etsi.org/swagger/ui/?url=https://forge.etsi.org/rep/cim/ngsi-ld-openapi/-/raw/v1.7.1/openapi-3.0.3/ngsi-ld-api.yaml

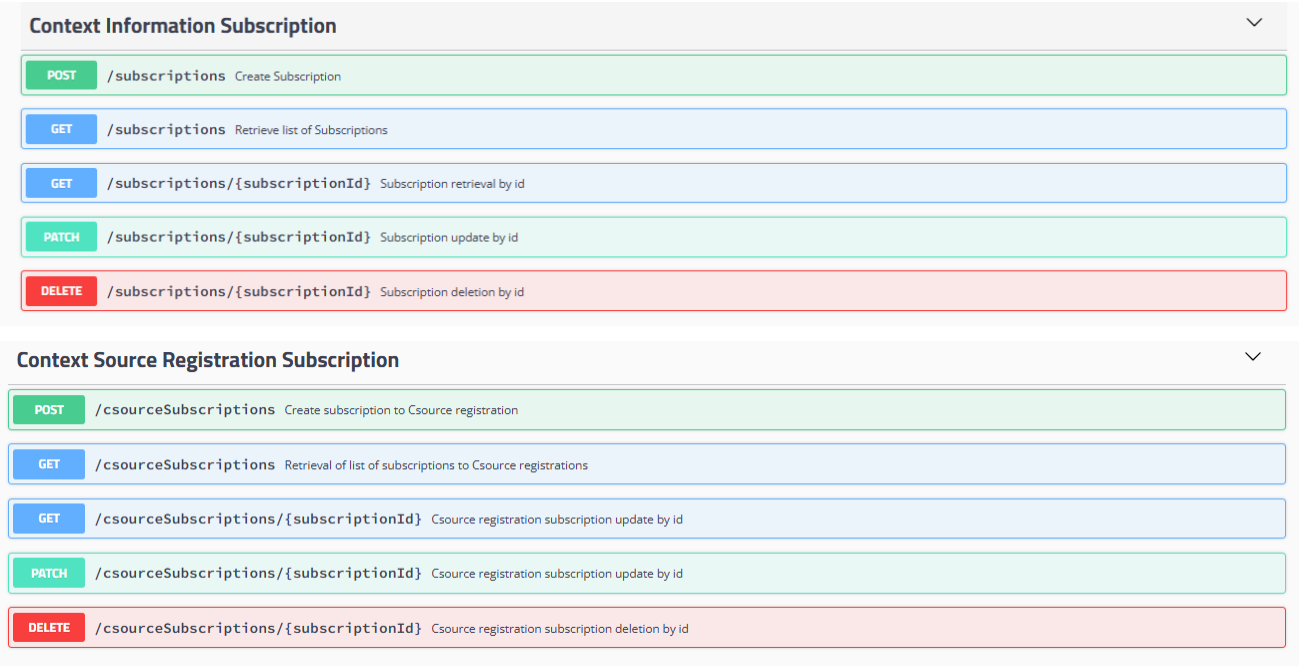


Figure 20. ContextBroker API inside aerOS OpenAPI (2)

3.2.2.2. Federator API

The Federator facilitates the bidirectional exchange of information with other domains of the aerOS continuum. To do so, a central registry located in the entrypoint domain keeps an inventory of all integrated domains and promote new domains registration and connection with those that are already part of the continuum, by means of the Federator API.

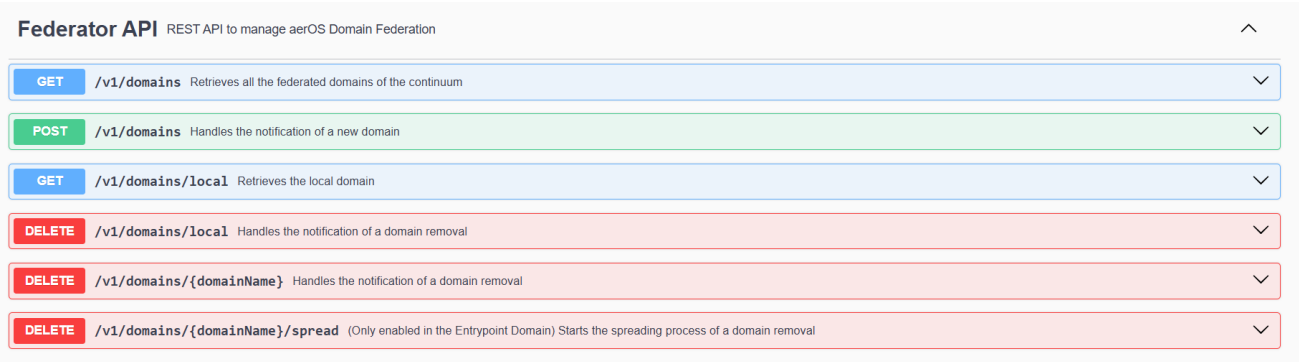


Figure 21. Federator API inside aerOS OpenAPI

3.2.2.3. HLO API

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. It can be accessed from two different components, FrontEnd (HLO-FE), and Deployment engine (HLO-AL).

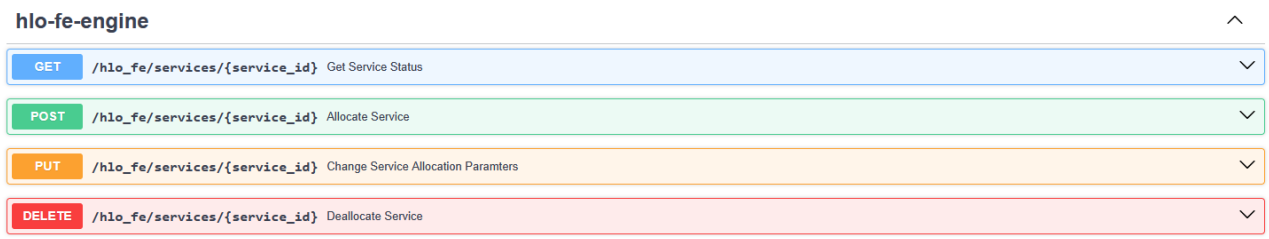


Figure 22. HLO API inside aerOS OpenAPI

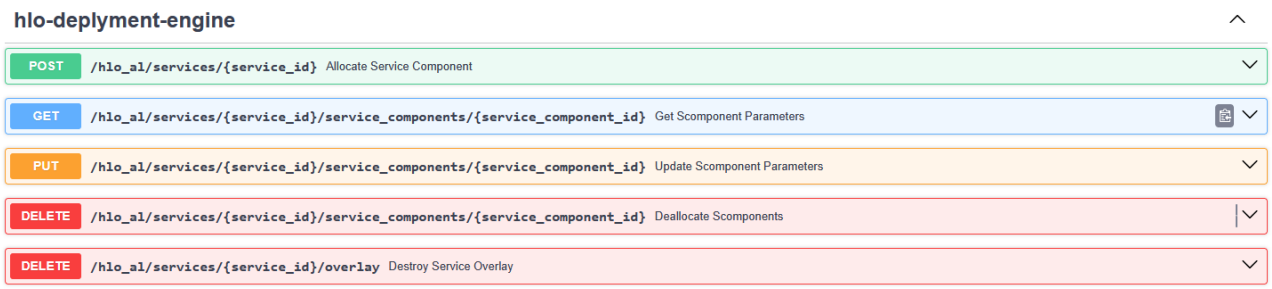


Figure 23. HLO API inside aerOS OpenAPI

3.2.2.4. LLO API

LLO provides granular control over specific functionalities, enabling precise manipulation of the underlying systems through the LLO API:

Figure 24. LLO API inside aerOS OpenAPI

3.2.2.5. Data Fabric API

The Data Fabric paradigm introduces a metadata-driven architecture that automates the integration of data from heterogenous sources and enables uniform access to the data through a standard interface. Hence, it is integral to the efficient handling and integration of data across the aerOS platform, ensuring seamless data flow and accessibility. Two main OpenAPI paths are available: Data Catalog Service and Data Security Service.

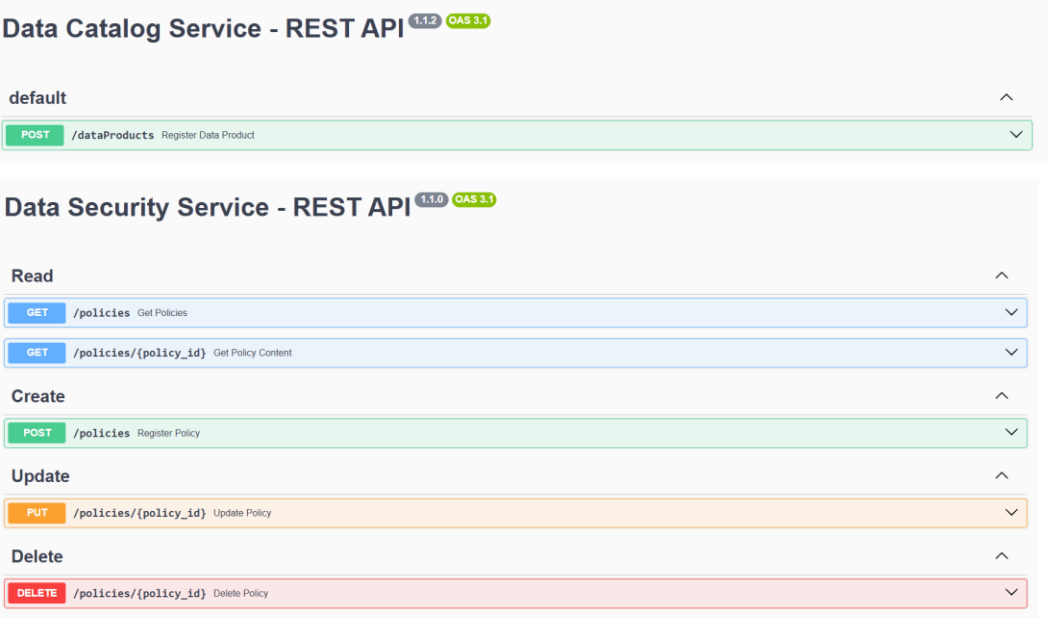


Figure 25. Data Fabric API inside aerOS OpenAPI

3.2.2.6. Data Product Manager API

The aerOS Data Fabric, by means of the Data Product Manager, exposes an interface towards data owners to onboard new data products and orchestrate the pipeline that turns raw datasets into data products. Thus, it is an essential part of the data fabric for managing the lifecycle of data products, underpinning the platform's data governance and utilization strategies.

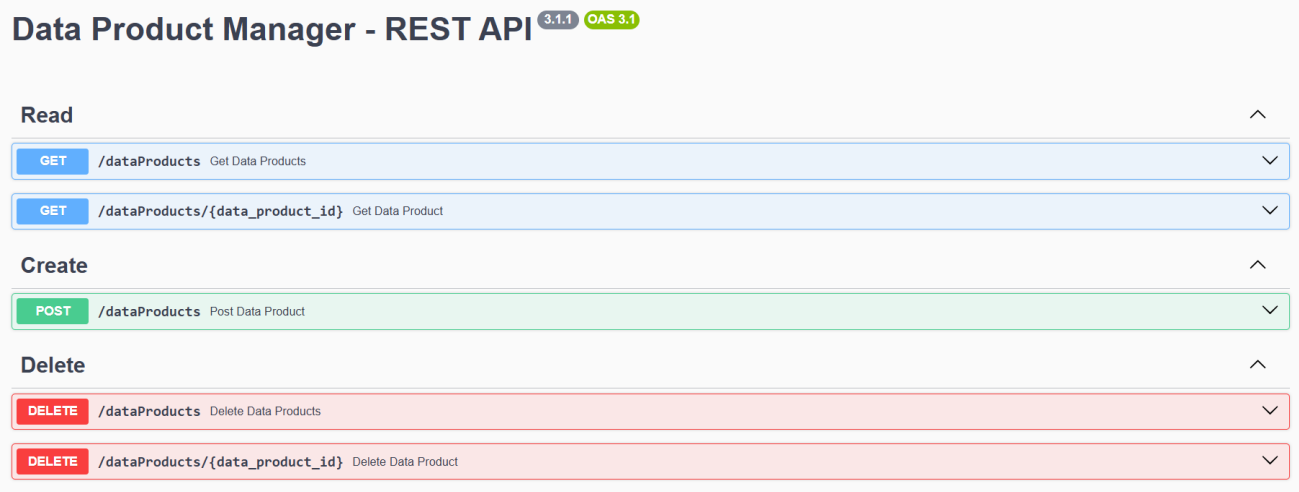


Figure 26. Data Product Manager API inside aerOS OpenAPI

3.2.2.7. Self-Capabilities API

They are crucial for the self-reporting and autonomous operation of the infrastructure components. As explained in previous WP2-WP3 deliverables, there are multiple operations, which are accessible through the Self-API component. It includes Self-orchestrator, Self-security, Self-Optimization, Self-scaling, Self-healing, and Self-Configurator. At the time of writing this deliverable, the latest version of the first four already included their swagger OpenAPI in their artifacts. The remaining two will be integrated before the project ends.

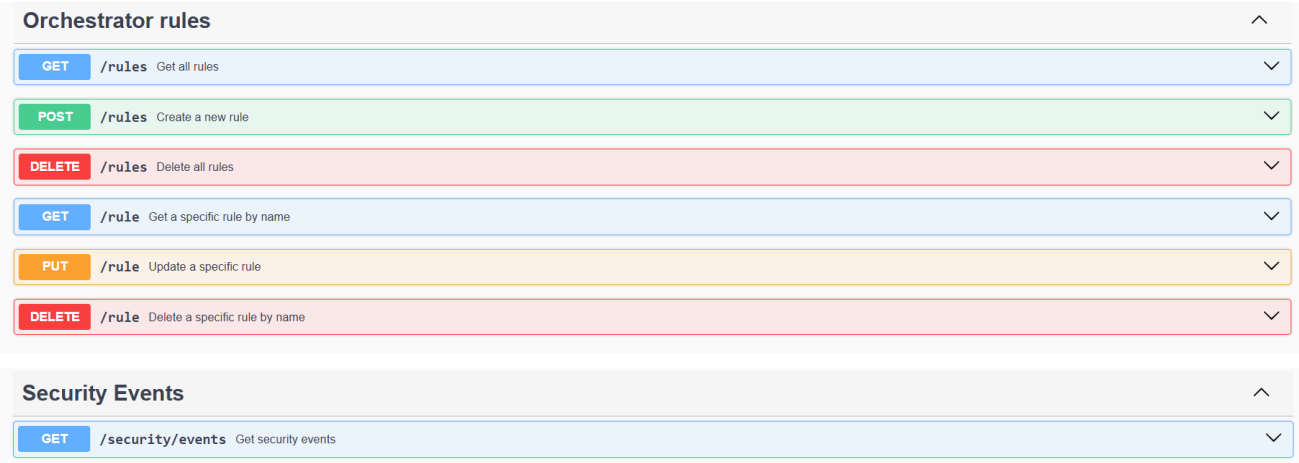


Figure 27. Self-Capabilities API inside aerOS OpenAPI

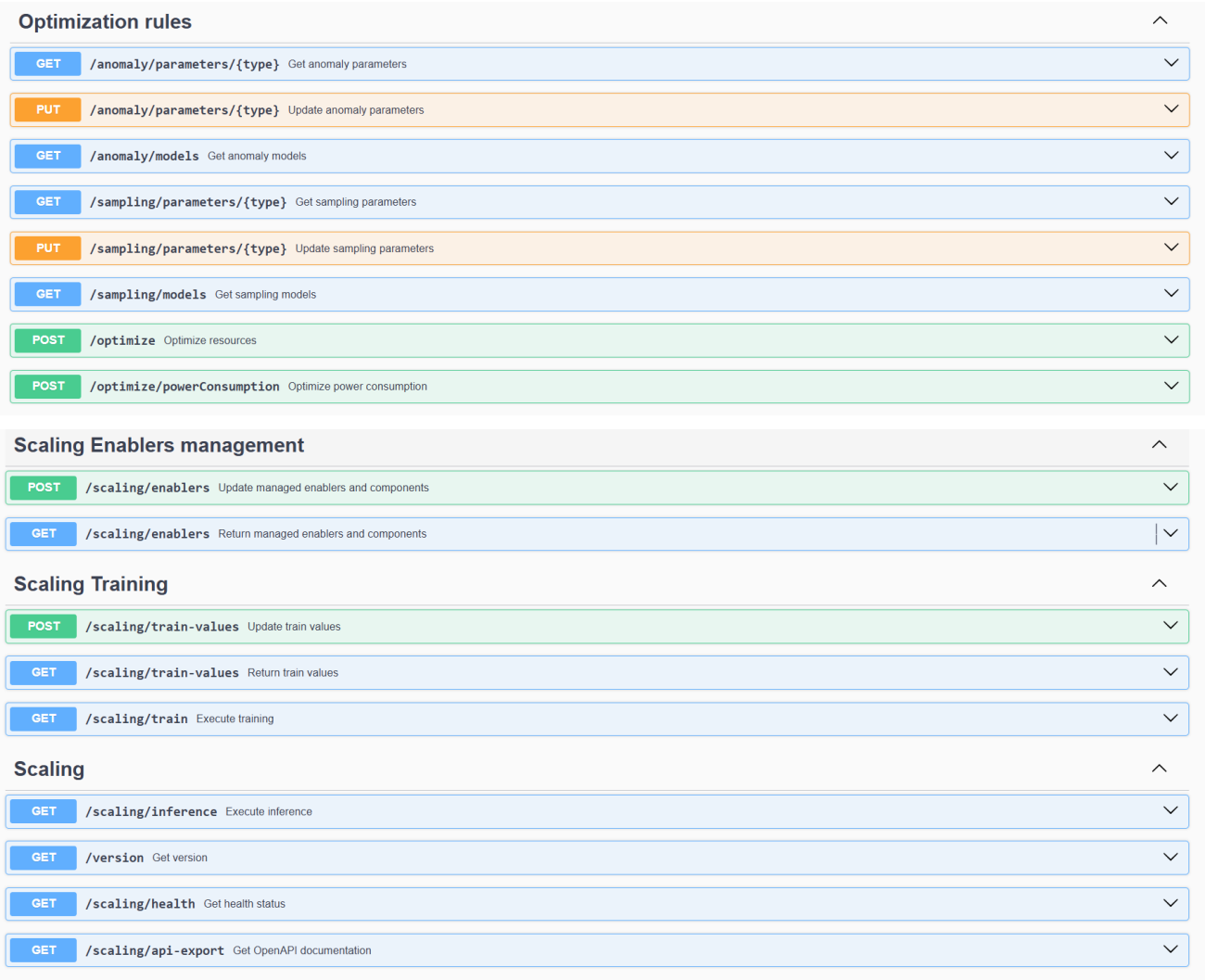


Figure 28. Self-Capabilities API inside aerOS OpenAPI (2)

3.2.2.8. IdM

A key component of the aerOS cyber security system is the aerOS Identity Management (IdM), whose ability is to register and evaluate policies for resource and data access. It utilizes Keycloak IdM[4], which provides comprehensive functions to strengthen cybersecurity, by managing the authentication and authorization of aerOS clients with a non-official but thoroughly documented Open API specifications[5]. The most relevant paths used in aerOS are presented below:

4 https://www.keycloak.org/
5 https://github.com/ccouzens/keycloak-openapi

Figure 29. IdM API inside aerOS OpenAPI

aerOS IdM has been integrated with OpenLDAP[6] in order to enhance the adoption of aerOS IAM by stakeholders, facilitating the automatic federation of user information from the LDAP directory. This eliminates the need for manual transfer of user data to aerOS IdM, streamlining user management and group associations.

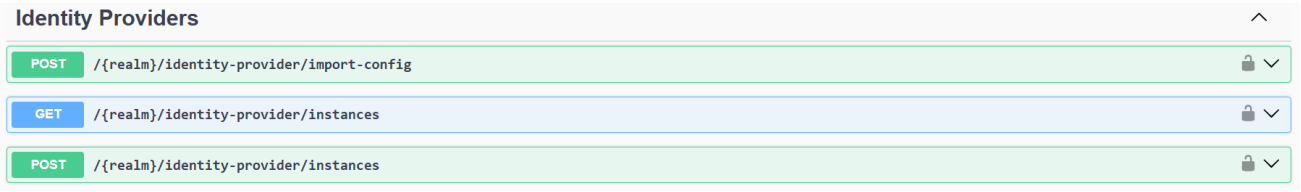


Figure 30. IdM API inside aerOS OpenAPI (2)

aerOS implements a system of precise control and management over resources, which is seen in the establishment of different roles. Each role is associated with specific access rights within the aerOS services environment and linked to a corresponding group in OpenLDAP.

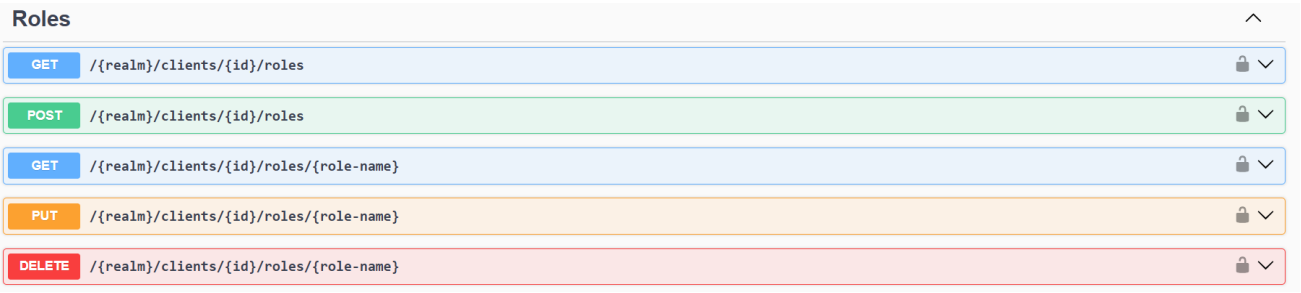


Figure 31. IdM API inside aerOS OpenAPI (3)

6 https://www.openldap.org/doc/admin26/OpenLDAP-Admin-Guide.pdf

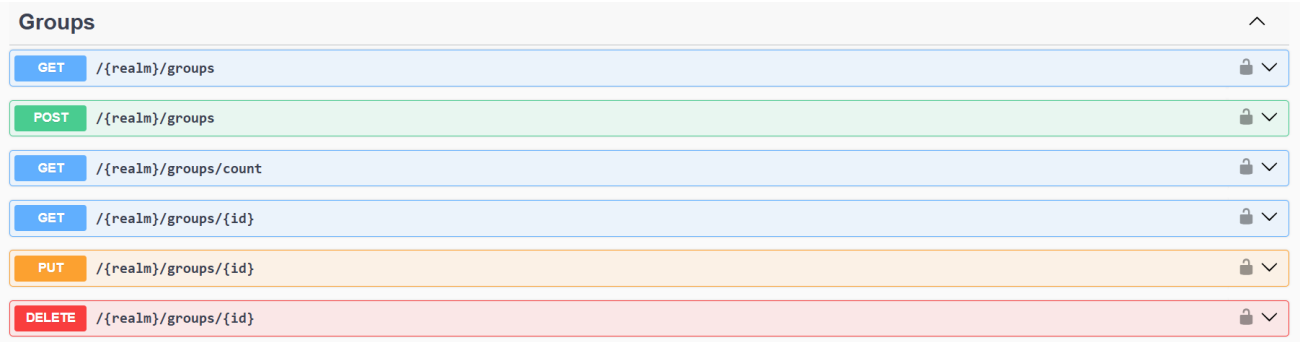


Figure 32. IdMAPI inside aerOS OpenAPI (4)

In turn, users group provide methods required to handle the lifecycle of user profiles via IdM.

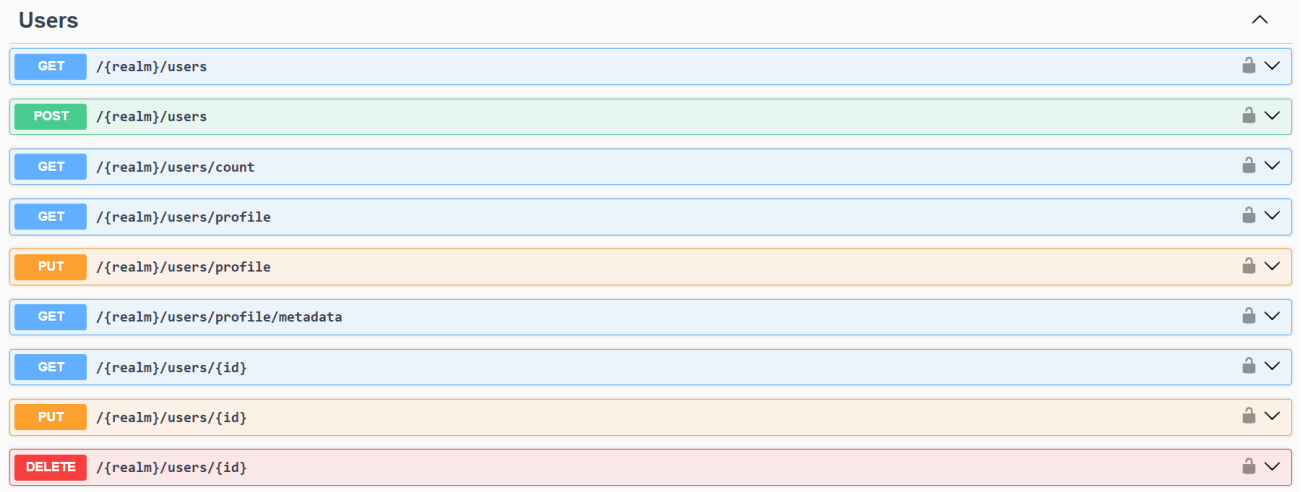


Figure 33. IdM API inside aerOS OpenAPI (5)

3.2.2.9. IOTA

One of the ways leveraged in aerOS to achieve trust is by taking advantage of open-source edge technologies such as IOTA's distributed ledger Tangle framework[7]. The Tangle is a data structure replicated across a network of nodes (IE's in the aerOS continuum) that contains all the information necessary to track messages and ensure traceability of the payloads distributed across the network. A brief extract of the official IOTA Open API documentation[8] is listed below:

7 https://wiki.iota.org/get-started/introduction/iota/introduction/
8 https://editor.swagger.io/?url=https://raw.githubusercontent.com/iotaledger/tips/main/tips/TIP-0025/core-rest-api.yaml

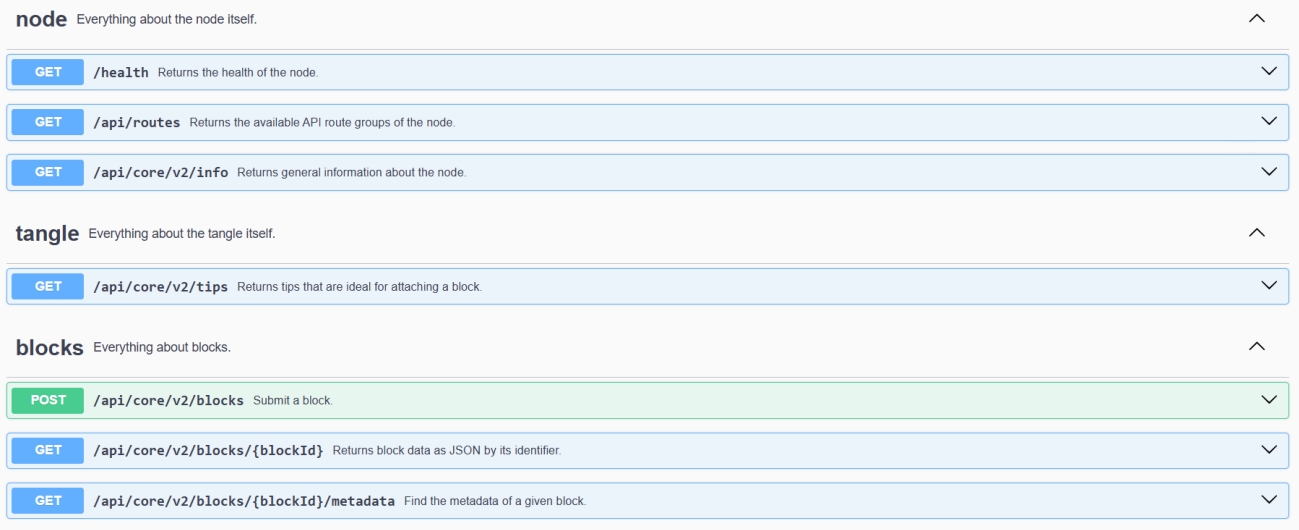


Figure 34. IOTA API inside aerOS OpenAPI

3.2.3. aerOS AsyncAPI

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.

The following figure illustrates the followed approach in aerOS to embrace the benefits that AsyncAPI can provide in the project needs. The following subsections detailed the AsyncAPI services implemented in the context of this task.

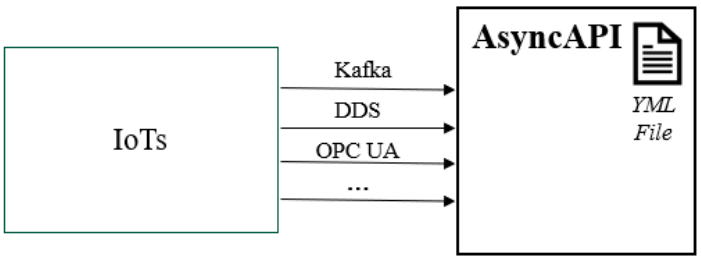


Figure 35. aerOS communications and services through AsyncAPI

3.2.3.1. Advancing AsyncAPI with Industry-standard Protocols

AsyncAPI currently serves as a crucial tool for specifying asynchronous interfaces. However, there is significant potential for enhancement by integrating additional industry-standard protocols such as Data Distribution Service (DDS)[9], ROS2[10], OPC UA[11], and the publish/subscribe protocol Zenoh[12]. Incorporating

9 https://www.dds-foundation.org/what-is-dds-3/
10 https://www.ros.org/

these protocols into the AsyncAPI framework would expand its applicability, promoting adoption across embedded systems and edge computing environments. This expansion is oriented to drive the development of solutions specifically tailored to industrial contexts, particularly in terms of automated interface integration. Such advancements would streamline application development within industrial automation systems, fostering innovation and efficiency in this vital sector.

Integrating any protocol or standard into the AsyncAPI framework requires addressing two primary challenges. Firstly, there is the need to map the primitives of the protocol specification—such as DDS's topics, data readers, data writers, subscribers, and publishers—to AsyncAPI concepts like channels, operations, and messages. Secondly, features unique to the protocol that do not directly correspond to an AsyncAPI concept necessitate the definition of a binding. This *binding* provides protocol-specific information pertaining to *servers*, *messages*, *channels*, and *operations*.

We briefly recap the core concepts of AsyncAPI v3.0.0:

- **Message:** A message is the unit of data exchanged between senders and receivers through a server. Messages follow a well-defined schema and fall in one of three classes: an event, a query, or a command.
- **Server:** A server represents a message broker or a messaging system that facilitates the exchange of messages between senders and receivers via channels. It's the infrastructure that handles the routing and delivery of messages.
- **Channel:** A channel is a named communication pathway within an AsyncAPI server that acts as the destination and source of messages or events.
- **Operation:** An operation specifies how messages between components are communicated. AsyncAPI differentiates between send and receive operations of messages on a channel. Additionally, operations support a reply semantic.

3.2.3.2. AsyncAPI for DDS

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. DDS is a publish/subscribe data distribution middleware comparable to MQTT. However, DDS operates decentralized and offers more capabilities for edge-focused applications and IoT environments.

When considering the AsyncAPI and DDS specifications side by side, it becomes apparent that many core concepts of AsyncAPI map directly to primitives in the DDS specification, i.e., AsyncAPI write operations map to data writers in DDS, AsyncAPI read operations map to data readers in DDS, and AsyncAPI channels map to DDS topics. However, there are core concepts in DDS especially concerning the specification of quality-of-service requirements that have no direct counterpart in AsyncAPI and demand a DDS AsyncAPI binding.

3.2.3.2.1. An Experimental AsyncAPI Binding for DDS

In this section, it is described an experimental DDS binding for messages, server, channels, and operations that enables specifying DDS event-driven applications in AsyncAPI.

An AsyncAPI **Message** is the only concept that directly maps to DDS without the need for a binding. The supported data types available in AsyncAPI specification and the DDS IDL map directly so that each key/value pair in the payload of a message directly map to a key/value pair of a data class in DDS.

In contrast, the concept of a **Server** present within the AsyncAPI specification has no direct application in a decentralized data distribution middleware such as DDS. However, in the AsyncAPI specification the *host* field is mandatory so that we can use it to specify a DDS peer discovery host that discovery packets are sent to

11 https://opcfoundation.org/about/opc-technologies/opc-ua/
12 https://zenoh.io/

in addition to the default multicast address. In DDS every communication between a *data writer* and a *data reader* requires the specification of a *domain*. A domain represents a subsection of the DDS network that is uniquely identified by a 32-bit unsigned integer. We introduce a domain object in the AsyncAPI DDS server binding that allows configuring various aspects of the DDS domain specification as summarized in Table 1. The DDS domain object in turn supports the specification of a *discovery* object containing options to configure the detection of domain participants. The discovery object and related *ports* and *interfaces* objects are summarized in Tables 2, 3, and 4. Lastly, the DDS server binding allows to specify a *quality of service (QoS) provider* that allows specifying a path from where the DDS application described by the specification at hand loads available QoS policies.

Table 5. DDS Domain Object

Field Name	Type	DDS Versions	Description
id	integer	1.4	The identifier of the DDS domain, a 32-bit unsigned integer.
discovery	object	1.4	Discovery options for the domain.
allowMulticast	string	1.4	Whether multicast discovery is allowed. Comma-separated list of: false, spdp, asm, ssm, true, default.
dontRoute	boolean	1.4	Allows setting the SO_DONTROUTE socket option.
enableMulticastLoopback	boolean	1.4	Must be true for intra-node multicast communication.
entityAutoNaming	string	1.4	Specifies the entity auto naming mode. Either empty (default) or fancy.
externalNetworkAddress	string	1.4	Explicitly overrule the network address DDS advertises in the discovery protocol which defaults to the address of the preferred network interface. It can be used to allow DDS to communicate across network address translation devices.
externalNetworkMask	string	1.4	Specify the network mask of the external network address. The default value is 0.0.0.0.
fragmentSize	integer	1.4	The size of a DDSI fragment. The default is 1334 B.
interfaces	object	1.4	The network interfaces used for discovery and user traffic.
maxMessageSize	integer	1.4	The maximum size of UDP payload.
maxRexmitMessageSize	integer	1.4	The maximum size of a retransmitted message.
multicastRecvNetwork InterfaceAddresses	string	1.4	A comma-separated list of network interface addresses to receive unicast traffic on. Alternatively, one of the following: all (listen on all multicast-capable interfaces), any (listen for multicast on the operating system default interface), preferred (listen on interface with highest priority), or none (listen on no interfaces).
multicastTimeToLive	integer	1.4	The time-to-live value for multicast packets. The default is 32.
redundantNetwork-	boolean	1.4	Whether to enable redundant networking on

ing			selected network interfaces.
transport	string	1.4	The transport to use for DDSI traffic: default, udp, udp6, tcp, tcp6, raweth.
useIPv6	boolean	1.4	Whether to use IPv6 for DDS traffic.

Table 6. DDS Discovery Object

Field Name	Type	DDS Versions	Description
DSGracePeriod	string	1.4	Controls how long discovered endpoints will survive after the discovery service disappears. This allows reconnection without loss of data if the discovery service restarts. The default is 30 s. Recognized units are day, hr (hours), min (minutes), s (seconds), ms (milliseconds), us (microseconds), and ns (nanoseconds).
defaultMulti-castAddress	string	1.4	The multicast address used for all traffic except for participant discovery. Defaults to the Simple Participant Discovery Protocol (SPDP) address 239.255.0.1.
enableTopicDiscover-yEndpoints	boolean	1.4	Whether to enable the use of topic discovery endpoints. The default is false.
externalDomainId	string	1.4	An override for the domain id is used to discovery and determine the port number mapping. The value default disables the override.
leaseDuration	string	1.4	The duration of the lease for the domain participant. The default is 10 s. Recognized units are day, hr (hours), min (minutes), s (seconds), ms (milliseconds), us (microseconds), and ns (nanoseconds).
maxAutoParticipant-Index	integer	1.4	This element specifies the maximum DDSI participant index if the participantIndex is “auto”. The default is 9.
participantIndex	string	1.4	The participant index used for discovery. The value auto selects the index automatically. The default is default using none if multicast discovery is used or else auto.
ports	object	1.4	The port numbers used for discovery and user traffic.
SPDPInterval	string	1.4	The interval at which SPDP messages are sent. The default corresponds to about 80% of the participant lease duration with a maximum of 30 s. Recognized units are day, hr (hours), min (minutes), s (seconds), ms (milliseconds), us (microseconds), and ns (nanoseconds).
SPDPMulti-castAddress	string	1.4	The multicast address used for participant discovery. Defaults to the SPDP address 239.255.0.1.
tag	string	1.4	A tag that domain participants to be discovered must match in addition to the domain ID.

Table 7. DDS Ports Object

Field Name	Type	DDS Versions	Description
base	integer	1.4	The base port number. The default is 7400.
domainGain	integer	1.4	The gain applied to the domain id to determine the port number. The default is 250.
multicastDataOffset	integer	1.4	The offset applied to the base port number to determine the multicast data port number. The default is 1.
multicastMetaOffset	integer	1.4	The offset applied to the base port number to determine the multicast meta port number. The default is 0.
participantGain	integer	1.4	The gain applied to the participant index to determine the port number. The default is 2.
unicastDataOffset	integer	1.4	The offset applied to the base port number to determine the unicast data port number. The default is 11.
unicastMetaOffset	integer	1.4	The offset applied to the base port number to determine the unicast meta port number. The default is 10.

Table 8. DDS Interfaces Object

Field Name	Type	DDS Versions	Description
autodetermine	boolean	1.4	Whether to let DDS determine the network interfaces automatically. The default is true.
address	string	1.4	The address of the interface to use.
name	string	1.4	The name of the interface to use. If both address and name are provided the address must match the interface name.
allow_multicast	string	1.4	A comma-separated list controlling of some of the following keywords: “spdp”, “asm”, “ssm”, or either of “false” or “true”, or “default” to control if DDS uses multicast on the interface.
multicast	string	1.4	If set to default it will use the value as returned by the operating system. If set to true it will enable multicast on the interface regardless of the operating system state.
preferMulticast	boolean	1.4	Whether to prefer multicast over unicast when unicast would suffice.
presenceRequired	boolean	1.4	Whether the interface must be present.
priority	integer	1.4	The priority of the interface. The default is 0.

In summary, the DDS binding in the AsyncAPI server specification has been used to configure all aspects of the DDS domain the event-driven application participates in.

A communication between participants is described by an AsyncAPI **Channel** that maps to DDS topic(s). The DDS channel binding extends the channel specification by a *QoS policies* object. A full list of applicable QoS policies can be found in the DDS specification v1.4 and each QoS policy should be provided as part of the DDS binding to enable their proper configuration. Furthermore, each AsyncAPI channel is associated with a set of messages that correspond to DDS data types. In contrast to an AsyncAPI channel, a DDS topic only

supports a single data type thus if an AsyncAPI specification describes an application with several messages per channel, we must ensure that the DDS binding and supporting tools (e.g. the code generator) map each valid combination of AsyncAPI channel and message to a distinct DDS topic.

Table 9. DDS Operation Binding Object

Field Name	Type	DDS Versions	Description
qosPolicies	list	1.4	Defines QoS policies for the operation. Find a list of applicable QoS policies below. If the DataReader or DataWriter inherits the QoS settings from their Publisher or Subscriber, respectively, the QoS policies are not required.
publishers	list	1.4	The publisher objects the DataWriter belonging to a send operation is associated with.
subscribers	list	1.4	The subscriber objects the DataReader belonging to a receive operation is associated with.

Table 10. DDS Publisher Object

Field Name	Type	DDS Versions	Description
name	string	1.4	The name of the publisher object the DataWriter is associated with.
qosPolicies	list	1.4	QoS policies applied to the Publisher.

Table 11. DDS Subscriber Object

Field Name	Type	DDS Versions	Description
name	string	1.4	The name of the subscriber object the DataReader is associated with.
qosPolicies	list	1.4	QoS policies applied to the Subscriber.

Table 12. Subset of DDS QoS Policy Objects

QoS Policy	Field Name	Type	DDS Versions	Description
Reliability	**kind**	string	1.4	One of best_effort or reliable.
	max_blocking_time	string	1.4	The maximum blocking time. The default is 100 ms. Recognized units are day, hr (hours), min (minutes), s (seconds), ms (milliseconds), us (microseconds), and ns (nanoseconds).
Deadline	**period**	string	1.4	The period of the deadline. The default is INFINITE. Recognized units are day, hr (hours), min (minutes), s (seconds), ms (milliseconds), us (microseconds), and ns (nanoseconds).
Durability	**kind**	string	1.4	One of volatile, transient_local, transient, or persistent.

```yaml
  sendPosition:
    action: send
    channel:
      $ref: '#/channels/robotStatePosition'
    bindings:
      x-dds:
        publishers:
          robotLocator:
            name: RobotLocator
            qosPolicies:
              reliability:
                kind: reliable
                maxBlockingTime: 1s
              durability:
                kind: volatile
              deadline:
                period: 20ms
  sendOrientation:
    action: send
    channel:
      $ref: '#/channels/robotStateOrientation'
    bindings:
      x-dds:
        publishers:
          robotGyro:
            name: RobotGyro
            qosPolicies:
              reliability:
                kind: reliable
                maxBlockingTime: 1s
              durability:
                kind: volatile
              deadline:
                period: 40ms
```

Figure 36. AsyncAPI specification of a reliable service using the DDS binding

Participants of a DDS domain communicate using read/write operations between *data writers* and *data readers*. AsyncAPI **operations** and their *send* and *receive* actions map directly to DDS data writers and data readers. In addition, DDS data writers and data readers may be associated with a DDS Publisher or DDS Subscriber, respectively, and may inherit the QoS settings of their Publisher and Subscriber or set their own QoS policies. As a result, the DDS operations binding (c.f. Table 5) allows us to define a list of *QoS policies* per operation or specify a DDS *publisher* object (c.f. Table 6) for write operations and a DDS *subscriber* object (c.f. Table 7) for read operations. Table 8 highlights the specification of a subset of available QoS policies available to DDS publishers and subscribers.

Figure 1 provides an example of using the experimental DDS binding (indicated by x-*) to specify a reliable service that receives sensor readings and actuator status variables of a robot and publishes its predicted trajectory every 20ms.

An AsyncAPI specification using the described DDS bindings can be processed by existing tooling, e.g. by AsyncAPI Studio to generate documentation, by using the extension mechanism. We further validate the correctness and functionality of resulting DDS applications by providing a rudimentary code generator for CycloneDDS and python.

3.2.3.2.2. CycloneDDS code generator for AsyncAPI

Note: This development is not endorsed in the software attachment due to privacy concerns.

The AsyncAPI React template rendering engine has been leveraged to implement a code generation solution capable of producing Python code utilizing the CycloneDDS framework. This code generator is built upon the experimental DDS binding of the AsyncAPI specification, validating the soundness of the mapping between DDS concepts and AsyncAPI primitives by yielding functional CycloneDDS applications from an interface specification. The organization's efforts have focused on implementing and validating the representation of fundamental DDS concepts, including topics, data writers/readers, publishers, subscribers, and a subset of quality-of-service objects within the AsyncAPI binding. The AsyncAPI React generator SDK facilitates the implementation of code generators by parsing a specification and providing a straightforward API[13] to interact with the parsed AsyncAPI objects, such as messages, channels, operations, and bindings.

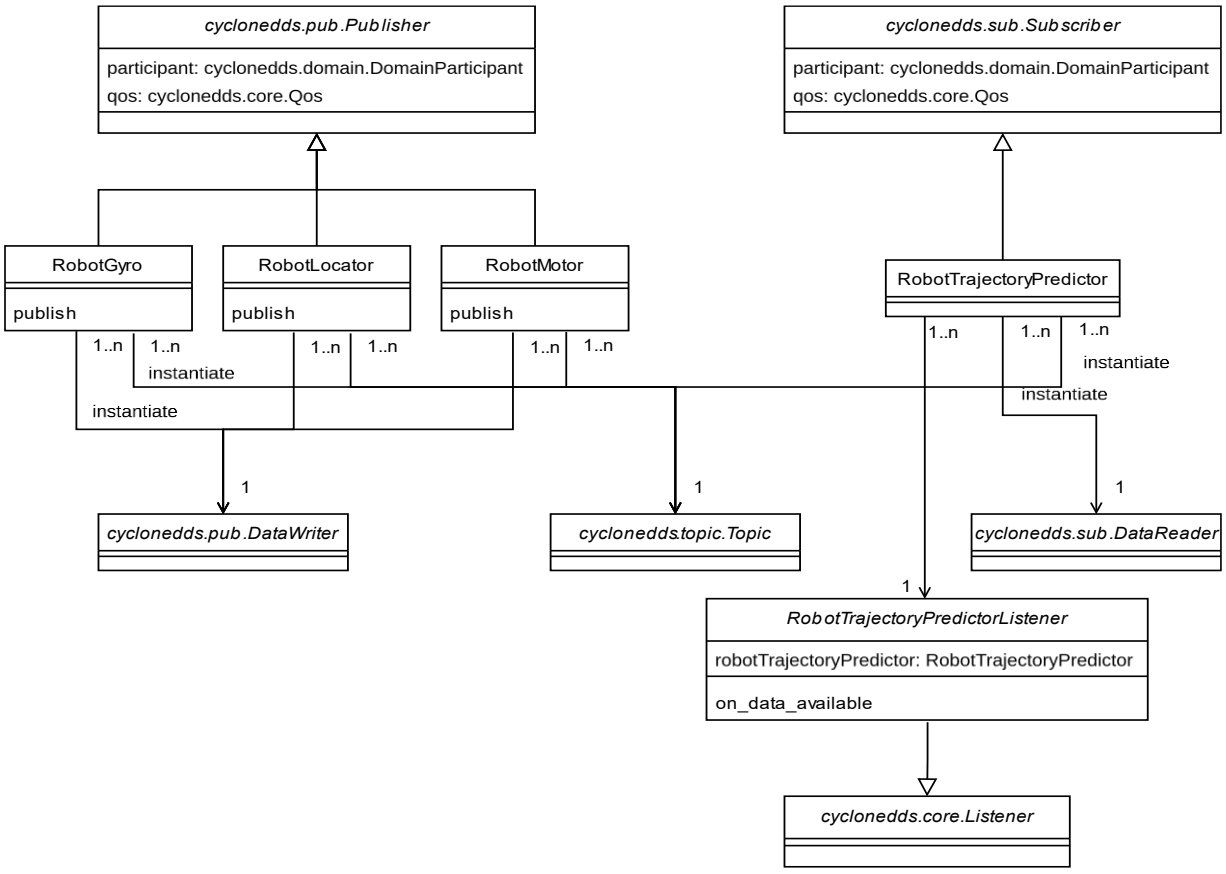


Figure 37. UML class diagram of generated CycloneDDS python code

The CycloneDDS generator creates a python data class per AsyncAPI message and associates each DDS topic with a unique tuple of AsyncAPI channel and message. For each AsyncAPI send/receive operation, the generator instantiates a data writer/reader and associates each data writer/reader with a publisher/subscriber using the operations DDS binding. In addition, for each data reader of a subscriber, the generator sets up a DDS listener. The DDS listener implements a callback that triggers when new data is available on the topic of the corresponding data reader. Optionally, the CycloneDDS generator associates receive operations' reply objects with a data writer that enables data readers to send a response on a defined topic.

A functional python CycloneDDS application from the robot trajectory example has been generated. The resulting application publishes a robots position, orientation, and speed on corresponding topics and a trajectory predictor component subscribes to the topics to use the received messages to calculate and publish the predicted trajectory of the robot. All send and receive operations are specified to be reliable, messages

13 https://github.com/asyncapi/parser-api/blob/master/docs/api.md

being volatile, and occurring periodically with a deadline. The corresponding QoS policies are automatically added to the publisher and subscriber and applied to the data writers and data readers. Figure 37 showcases a UML class diagram of the resulting DDS publishers and the RobotTrajectoryPredictor DDS subscriber and its DDS Listener that is activated when a data reader has messages ready.

When the CycloneDDS application executes it creates a thread per publisher and subscriber and reads/writes data as specified in the QoS policies. The resulting output can be seen in Figure 38.

```
> npm run test:clean && npm run test:generate && npm run test:run


> python-cyclonedds-template@0.0.1 test:clean
> rimraf test/project/client.py


> python-cyclonedds-template@0.0.1 test:generate
> asyncapi generate fromTemplate test/fixtures/asyncapi.yaml ./ --output test/ --force-write --param pythonModuleName=t
est_module

   AsyncAPI Generator

◑  Generation in progress. Keep calm and wait a bit
Created directory structure:
test/test_module
test/test_module/dataclasses
test/test_module/publishers
test/test_module/subscribers

◇  Check out your shiny new generated files at test/.


> python-cyclonedds-template@0.0.1 test:run
> python3 test/main.py

Running robotLocator
Running robotMotor
Running robotGyro
Running robotTrajectoryPredictor
Received '[RobotStateSpeed(value=0.4981216817406924)]' on topic 'robot/state/speed'
Replying on topic 'robot/trajectory'
Received '[RobotStatePosition(x=-198659610, y=405437461)]' on topic 'robot/state/position'
Replying on topic 'robot/trajectory'
Received '[RobotStateSpeed(value=0.9963081962663275)]' on topic 'robot/state/speed'
Replying on topic 'robot/trajectory'
Received '[RobotStateOrientation(value=323)]' on topic 'robot/state/orientation'
Replying on topic 'robot/trajectory'
Received '[RobotStateSpeed(value=0.7841747195525831)]' on topic 'robot/state/speed'
Replying on topic 'robot/trajectory'
Received '[RobotStatePosition(x=320802509, y=363947182)]' on topic 'robot/state/position'
Replying on topic 'robot/trajectory'
Received '[RobotStateSpeed(value=0.19856607450562247)]' on topic 'robot/state/speed'
Replying on topic 'robot/trajectory'
Received '[RobotStatePosition(x=-863121496, y=903642757)]' on topic 'robot/state/position'
Replying on topic 'robot/trajectory'
Received '[RobotStateSpeed(value=0.01599130575464991)]' on topic 'robot/state/speed'
Replying on topic 'robot/trajectory'
Received '[RobotStateOrientation(value=25)]' on topic 'robot/state/orientation'
Replying on topic 'robot/trajectory'
Received '[RobotStateSpeed(value=0.10025642588002226)]' on topic 'robot/state/speed'
Replying on topic 'robot/trajectory'
Received '[RobotStatePosition(x=-853495138, y=-729634313)]' on topic 'robot/state/position'
```

Figure 38. Output of code generator and exemplary CycloneDDS application

Note how the RobotTrajectoryPredictor receives twice as many RobotStateSpeed and RobotStatePosition messages than RobotStateOrientation message since their periods in the Deadline QoS policy as shown in the specification differ by a factor of two.

3.2.3.3. AsyncAPI for ROS 2

Robot Operating System 2 (ROS 2) is an open-source middleware framework designed for real-time, scalable communication in robotic systems. It builds on standards like DDS and Zenoh to facilitate effective data exchange crucial for complex robotic architectures. Supported by a vibrant community, ROS2 provides access to thousands of ready-to-use, community-driven libraries, making it an indispensable tool for rapid prototyping and deployment across various sectors—from industrial automation to autonomous vehicles. Its robust features enable efficient multi-robot interactions and seamless cross-platform operations, contributing significantly to its widespread adoption.

Integrating ROS2 with AsyncAPI presents an exciting opportunity to standardize interface specifications in robotic applications. With ROS2's topics aligning naturally with AsyncAPI channels, and its publishers and subscribers corresponding to AsyncAPI's send and receive operations, there is a strong foundation for synergy. However, integration must address ROS2-specific Quality of Service (QoS) settings, which lack direct AsyncAPI equivalents. Developing an AsyncAPI-ROS2 binding would effectively encapsulate these QoS parameters, facilitating precise and reliable system configurations. This integration not only promises to streamline the specification process but also enhances interoperability and innovation within the robotics community.

3.2.3.3.1. An Experimental AsyncAPI Binding for ROS 2

We describe an experimental binding for messages, servers, channels, and operations that enables specifying event-driven applications in AsyncAPI. This binding is particularly relevant for ROS 2, which can use either DDS or Zenoh as its middleware.

Server Binding Object

In ROS 2, the server binding object contains information about the server representation. Since ROS 2 can use decentralized middleware with no central server, the host field can be set to localhost. When using Zenoh, the host field specifies the Zenoh Router IP address.

Table 13. ROS 2 Server Binding Object

Field Name	Type	ROS 2 Versions	Description
rmwImplementation	string	all	Specifies the ROS 2 middleware implementation to be used. Valid values include rmw_fastrtps_cpp (Fast DDS), rmw_cyclonedds_cpp (Cyclone DDS), rmw_connext_cpp (RTI Connext), and rmw_zenoh_cpp (Zenoh). This determines the underlying middleware implementation that handles communication.
domainId	integer	all	All ROS 2 nodes use domain ID 0 by default. To prevent interference between different groups of computers running ROS 2 on the same network, a group can be set with a unique domain ID. Must be a non-negative integer less than 232.

```yaml
servers:
  ros2:
    host: localhost
    protocol: ros2
    protocolVersion: humble
    bindings:
      ros2:
        rmwImplementation: rmw_fastrtps_cpp
        domainId: 0
```

Figure 39. Example ROS 2 server binding object

Operation and Channels Binding Object

AsyncAPI operations, with their send and receive actions, map directly to ROS 2 subscribers, publishers, actions, or services:

- send -> publisher, action_client, service_client
- receive -> subscriber, action_server, service_server

Unlike DDS, which only maps send/receive operations to publishers and subscribers, ROS 2 also includes request and response operations, encompassing services and actions. Each operation binding maps to a channel object with a ROS 2 role, node, and QoS policy object:

Table 14. ROS 2 Operation Binding Object

Field Name	Type	ROS 2 Versions	Description
role	string	all	Specifies the ROS 2 type of the node for this operation. Valid values are: publisher, subscriber, service_client service_server, action_client, action_server. This defines how the node will interact with the associated topic or action.
node	string	all	The name of the ROS 2 node that implements this operation.
qosPolicies	object	all	Quality of Service (QoS) for the topic.

Table 15. ROS 2 Quality of Service Object

Field Name	Type	ROS 2 Versions	Description
reliability	string	all	One of best_effort or reliable. More information here: ROS 2 QoS
history	string	all	One of keep_last, keep_all or unknown. More information here: ROS 2 QoS
durability	string	all	One of transient_local or volatile. More information here: ROS 2 QoS
lifespan	integer	all	The maximum amount of time between the publishing and the reception of a message without the message being considered stale or expired. -1 means infinite.
deadline	integer	all	The expected maximum amount of time between subsequent messages being published to a topic. -1 means infinite.
liveliness	string	all	One of automaticor manual. More information here: ROS 2 QoS
leaseDuration	integer	all	The maximum period of time a publisher has to indicate that it is alive before the system considers it to have lost liveliness. -1 means infinite.

```yaml
receiveCmdVel:
  action: receive
  channel:
    $ref: "#/channels/CmdVel"
  bindings:
    ros2:
      role: subscriber
      node: /turtlesim
        qosPolicies:
          history: unknown
          reliability: reliable
          durability: volatile
          lifespan: -1
          deadline: -1
          liveliness: automatic
          leaseDuration: -1
```

Figure 40. Example ROS 2 operation binding object

Message Binding Object

ROS 2 message types, defined in .msg, .srv, or .action files, are mapped to AsyncAPI message payloads. The following table describes how to map ROS 2 type to AsyncAPI types and format:

Table 16. ROS 2 type map to AsyncAPI types and format

ROS 2 Type	AsyncAPI Type	AsyncAPI Format
bool	boolean	boolean
byte	string	octet
char	integer	uint8
float32	number	float
float64	number	double
int8	integer	int8
uint8	integer	uint8
int16	integer	int16
uint16	integer	uint16
int32	integer	int32
uint32	integer	uint32
int64	integer	int64
uint64	integer	uint64
string	string	string
array	array	--

3.2.3.3.2. ROS 2 code generator for AsyncAPI

Note: The software associated to this development is not included in the compressed file since it is undergoing its own open-sourcing process (SIEMENS). It will be available in due time for the community.

The primary objective of our code generator is to utilize AsyncAPI tools equipped with ROS2 bindings to systematically transform the specified information into ROS2 interface definition files, including .msg, .srv, and .action files, as illustrated in Figure 41.

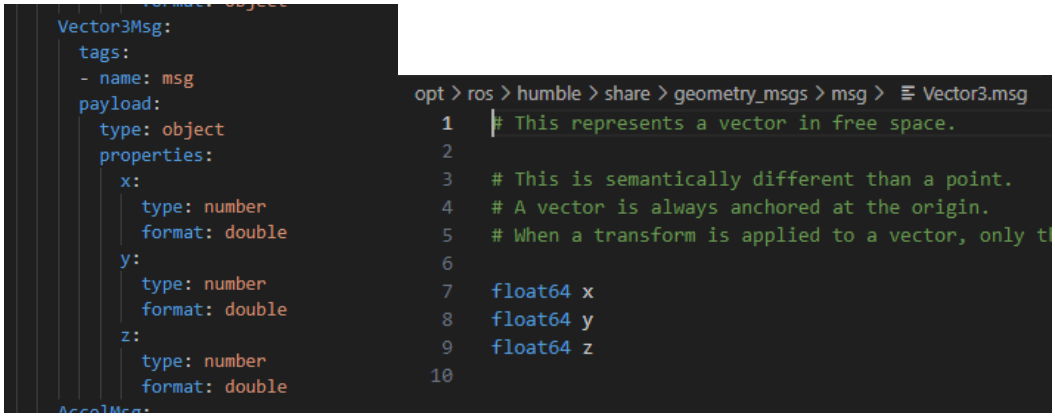


Figure 41. ROS 2 interfaces represented left as the AsyncAPI specification file format and right as a .msg ROS 2 file format

These generated ROS2 files, supplemented with additional details from AsyncAPI, serve as the foundation for deploying a comprehensive ROS2 application. The code generator is specifically optimized for integration with low-code tools, as discussed in Chapter 4.2.1.3. It extracts key elements such as topic names, interface descriptions, and Quality of Service (QoS) settings from the AsyncAPI specifications. Utilizing this information, the tool generates C++ code for ROS2 components, including subscribers, publishers, action clients, and service clients.

The application of this generated code for low code tools facilitates efficient development workflows, enabling rapid prototyping and deployment of ROS2 applications. Detailed usage and benefits of this approach are further elaborated in the subsequent sub-chapter.

3.2.4. 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, but instead function as a graphical low-code interface that triggers functionalities within already running applications with different parameters.The behaviour trees enable users to define triggers that activate specific services' functionalities, without initiating or terminating the services themselves. This approach 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.

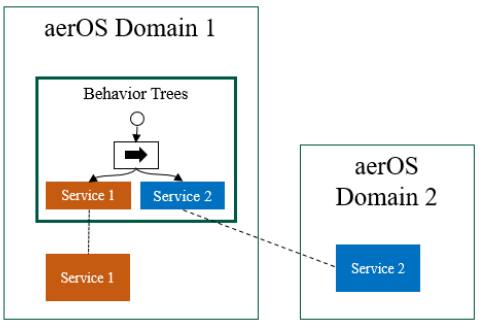


Figure 42. Behavior trees in aerOS.

In this illustrative example of the application of behavior trees within the aerOS framework, showcasing their role in triggering functionalities across different aerOS domains. The behavior tree was strategically configured to initiate specific functionalities of service 1 within aerOS domain 1, as well as trigger

functionalities in service 2, which could be executed in an external aerOS domain 2. This cross-domain interaction demonstrated the interoperable design of the aerOS system and the flexible nature of the behaviour trees.

In addition to the integration of behavior trees, Node-RED flows have been employed as another low-code tool within the aerOS project. Node-RED is a graphical programming tool that allows users to create and deploy applications through a browser-based interface. The integration of Node-RED flows into the aerOS system further enhances the system's flexibility and ease of use, complementing the capabilities provided by the behavior trees.

Users will be able to leverage the intuitive, drag-and-drop interface of Node-RED to define custom workflows and integrate various functionalities within the aerOS ecosystem.

This low-code approach will empower system administrators and users to rapidly configure and adapt the system's behavior without the need for extensive programming knowledge.

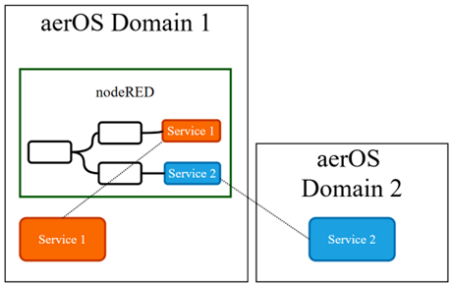


Figure 43. Node-RED in aerOS

As in the behavior tree's example, the role of Node-RED is to trigger functionalities across different aerOS domains. This tool has been used within the MVP, as discussed in Chapter 3.2.5.

The incorporation of behaviour trees and Node-RED thus represents a nuanced enhancement of the aerOS system's responsiveness and adaptability, providing users with powerful tools to influence the system behaviour dynamically while leaving the core orchestration responsibilities to the HLO and LLO.

3.2.4.1. Generate skills from AsyncAPI for Low Code tools

Within aerOS' scope, two low-code tools were employed: Node-RED and Behavior Trees. Both platforms offer intuitive GUIs that facilitate the integration of interfaces and the management of data transmission using protocols such as ROS2, REST, MQTT, and other industrial standards. Code generators can be utilized to create specific blocks within these tools, significantly enhancing usability and streamlining development processes.

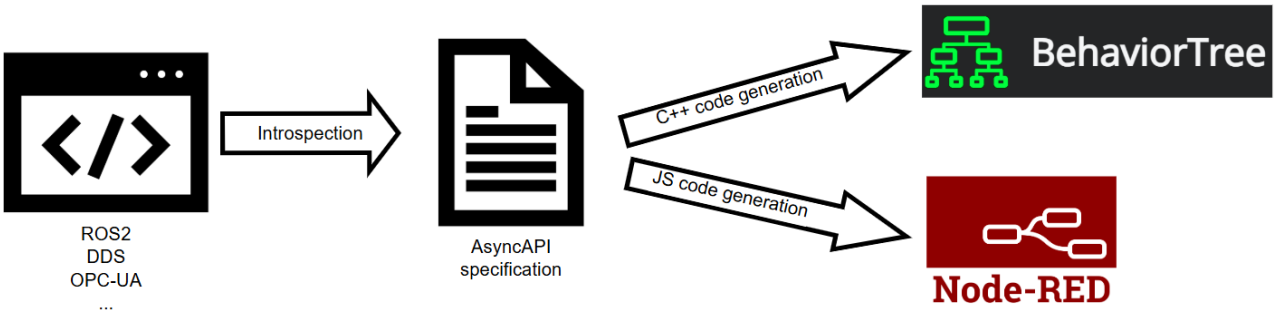


Figure 44. Flow to generate low-code skills

As can be seen in Figure 44, an AsyncAPI specification for the machine acts as the blueprint for generating these skills or blocks. This specification is transformed into C++ code for Behavior Trees or JavaScript code

for Node-RED. The generated blocks in the low-code tools represent specific robotic skills. Users can connect these skills or blocks via the GUI, efficiently deploying robust applications.

To further streamline the process, AsyncAPI specifications for machines can be auto generated, offering significant benefits, particularly in brownfield environments where manually documenting existing interfaces in an AsyncAPI YAML file is both time-consuming and labour-intensive. For instance, in machines utilizing ROS2, the AsyncAPI specification can be automatically created using ROS2 interface files (.msg, .srv, .action), as detailed in the previous subchapter. Alternatively, information can be extracted from live monitoring tools, although this approach necessitates an operational machine to accurately capture real-time interface dynamics.

3.2.5. Technologies and standards deployed in MVP

Table 17. 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.
Node-RED	Graphical programming tool that allows users to create and deploy applications through a browser-based, low-code interface.	It provides a UI to send commands to a service and automatically publish the outcome of those commands into an IOTA block.

3.3. aerOS service and resource orchestration

3.3.1. Main functionalities

3.3.1.1. 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.

3.3.1.2. High-level Orchestration components decomposition

As described in deliverable D3.1, 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.

3.3.1.3. 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, Operators watching an aerOS-specific custom resource in the Low Level Orchestrator 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.

The components constituting two types of low-level orchestrators (dockerd and K8S) have been described in previous deliverables. 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.

3.3.1.4. Connectivity and energy support orchestration in aerOS continuum ontology

The aerOS continuum ontology has provided the modelling tools for the internal functioning of the aerOS orchestration system. Nevertheless, certain models were missing but are important for the optimization of the allocations of Service Components into the IEs. In this regard, two modelling aspects were added. The first one being the connectivity, *Network Link* and *Network Port* have been added to the continuum ontology. They allow to model how Service Components in the current topology are connected, creating a network overlay that is automatically orchestrated as part of aerOS. These added concepts can also be augmented with optimization constraints such as latency and bandwidth. These allows further adaptation to the application nature of the user. The second one, related to energy ontologies have been added to accommodate two modelling needs. Energy efficiency has been integrated as part of the IE requirements and defines the required compromise between computation and energy of the running IE. The *Energy Source* ontology independently defines the percentage of green energy supplied to the infrastructure. The user can provide a minimum green energy requirement in the application blueprint.

3.3.1.5. High-level Orchestration Allocator AI algorithm

As part of the architecture of the HLO, the AI algorithms constitute the core of the smart mechanisms in the allocation of the services resources and deployed in the HLO Allocator. Any proposed AI algorithmic approach passes through different steps before final integration.

1- Design of an AI algorithm for the continuum.
2- Test and validation with simulation data.
3- Input and Output adaptation for the HLO Allocator data formats.
4- Final deployments and tests.

In the context of aerOS, different algorithms have been designed, then tested and validated. However, due to the current stage of the project, only some of them have been able to be deployed over real data (those used in the MVPv2). Others have been developed and tested over simulated data (compliant with aerOS data models and structure).

In the next pages, a summary of the algorithmic approaches developed can be found.

- **A Deep Reinforcement Learning (DRL) Allocator**

This has been the algorithm used in the MVPv2 validation of aerOS. Therefore, it has been the only one applied over a real, functional aerOS continuum (composed of three domains: CloudFerro (entrypoint), NCSRD and mobile domain).

In this case, the development of a HLO Allocator AI Algorithm utilizing a Deep Reinforcement Learning (DRL) approach based on the stable-baselines3 framework was performed. The work encompasses the creation of a synthetic dataset for training the DRL model, the integration of the allocator into the HLO, and a performance comparison with a Mixed Integer Linear Programming (MILP) approach, which has been identified as significantly slower. The development adheres strictly to Machine Learning Operations (MLOps) best practices to ensure scalability, reproducibility, and maintainability.

The decision to employ a DRL algorithm stems from the need for an efficient and scalable solution to the allocation problem within the HLO. DRL offers the ability to learn complex policies that can generalize over a wide range of scenarios, making it suitable for dynamic and diverse environments. Unlike traditional approaches such as Mixed Integer Linear Programming (MILP), which re-calculate the optimal allocation for every request, DRL shifts the computational overhead from online calculation to offline training through back-propagation. During inference, the allocation action is computed using a neural network, resulting in significantly reduced computational effort for each allocation request.

In the context of the aerOS federation, which supports large networks and distributed systems across multiple domains, scalability is a crucial requirement. DRL enables higher scalability on a per-request basis, allowing for a larger number of devices within the same domain. For example, while a MILP approach with similar objectives requires approximately one minute to compute allocations for a network with 60 devices, our DRL

approach requires less than one second. This translates to a saving of approximately 59 seconds per allocation request compared to MILP.

The HLO Allocation Engine receives as input the service component to be placed and a pre-filtered list of IE candidates from the HLO Data Aggregation and Alert System, as defined in the protobuf message format. The DRL approach utilizes the service component definition, including its constraints, and the resources of an IE, such as memory usage, to minimize the expected latency between service components and the overall power consumption. The output of the HLO Allocation Engine is an allocation mapping between the service component and an IE, and optionally, the previous IE if the service was deployed previously.

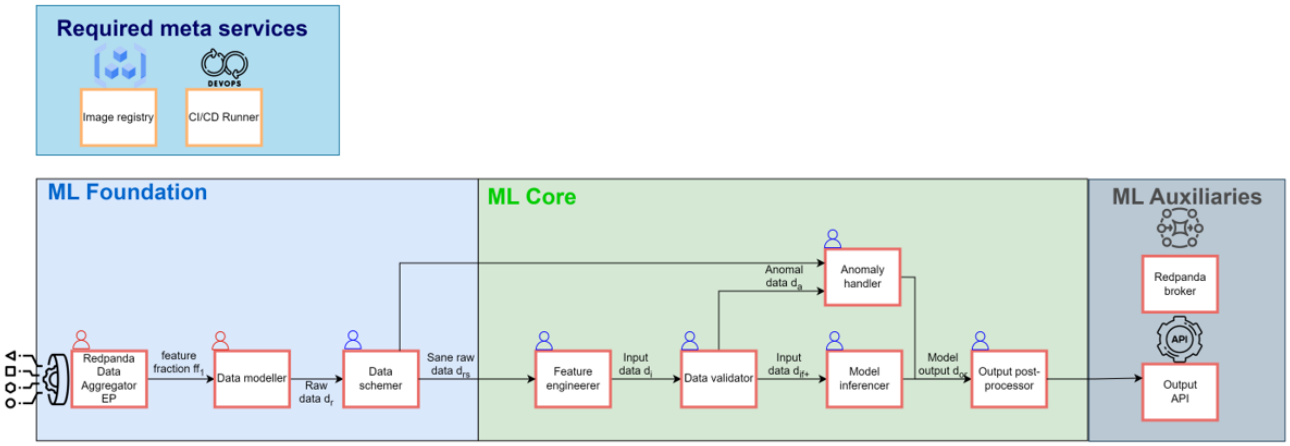


Figure 45. MLOps inference pipeline structure for the HLO Allocator AI algorithm.

As illustrated in Figure 45, the HLO Allocation Engine based on an MLOps pipeline structure to ensure future scalability and maintainability has been implemented. The approach is divided into three main parts, each subcategorized into specific functions:

- ML Foundation:
  - Redpanda Data Aggregator Endpoint: Collects data from the HLO Data Aggregation and Alert System.
  - Data Modeller: Structures and models the incoming data.
  - Data Schemer: Defines and manages the schema of the data for consistency.
- ML Core:
  - Anomaly Handler: Detects and handles anomalies in the data.
  - Feature Engineer: Processes and transforms raw data into meaningful features for the model inference.
  - Data Validator: Ensures data quality and validity.
  - Model Inferencer: Performs inference using the trained DRL model.
  - Output Post Processor: Refines and formats the model output for downstream applications.
- ML Auxiliaries:
  - Output API / Redpanda Broker: Manages communication and data exchange between HLO Allocator Engine and HLO Deployment Engine.

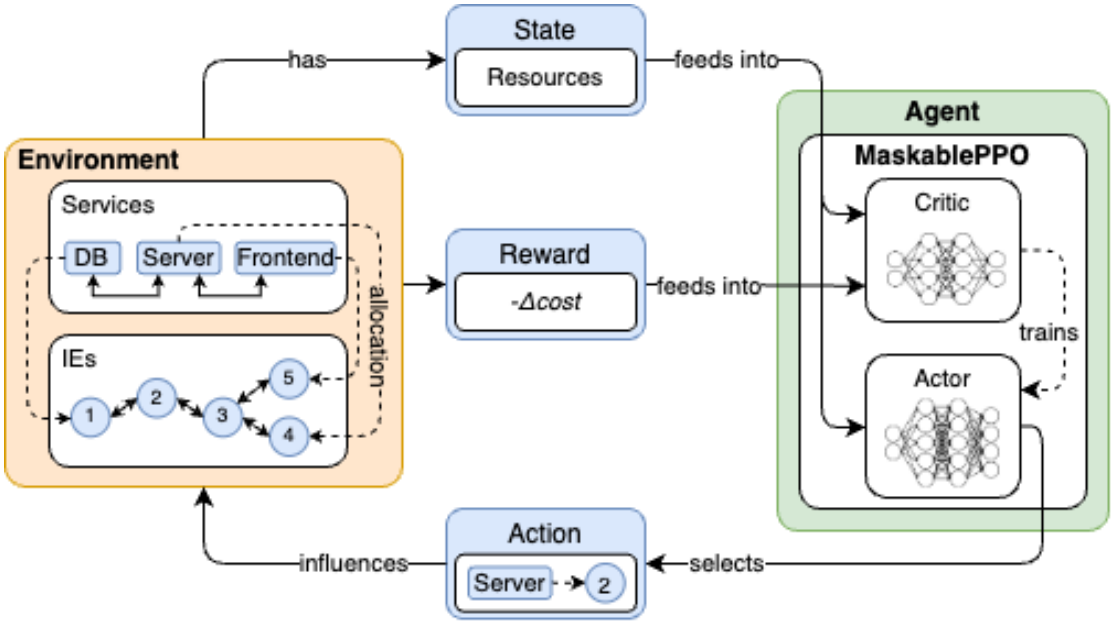


Figure 46. Deep Reinforcement Learning of the HLO Allocation Engine.

In DRL, an agent learns to make decisions in a complex environment by interacting with it and receiving rewards or penalties for its actions, as shown in Figure 46. The goal is to learn a policy that maximizes the cumulative reward over time. The MaskablePPO algorithm is employed, a variant of Proximal Policy Optimization (PPO), which is a popular model-free, on-policy DRL algorithm.

Concrete Example of DRL in HLO Allocation

In the aerOS DRL approach, the action during inference is the selection of an IE for a given service component. For instance, the agent might select "IE 2" for the service component "Server". To prevent the selection of unsuitable IEs, such as those exceeding certain resource thresholds (e.g., IEs larger than "5"), action masking to exclude these possibilities is employed. The reward function is defined as:

$$Reward = -\Delta cost$$

The cost includes factors such as:

- Latency between service components.
- Estimated power consumption of the service component on the IE.
- Penalty for CPU overload if the allocation would exceed the IE's CPU capacity.

Since DRL algorithms aim to maximize the reward, multiplying the cost by -1 effectively turns the problem into a cost minimization task. The state and reward are used by the critic to train the actor neural network, which then selects the actions. By continuously interacting with the environment and receiving feedback through rewards, the agent improves its policy over time.

Synthetic Dataset Generation for Training

Although the environment is fixed and could, in theory, generate random states, training can be improved both in efficiency and time by using a pre-defined dataset that resembles real-world scenarios. To achieve this, a synthetic dataset with different-sized networks in a hierarchical form and services with one or more interdependent service components was generated. This dataset contains:

- One hundred samples for each network size between 1 and 100.
- Zero to 200 samples for services with 1 to 30 interdependent service components.

By matching services with networks in the dataset, we can train the DRL model effectively. The pre-trained network can then be used for inference on actual allocation requests.

Multi-Tenancy Gaming approach from Slice Resource Provisioning perspective (theoretical)

aerOS consortium suggested the investigation of the maximization of total profit for all users through a Profit-aware Slicing Resource Provisioning approach with Multi-Tenancy Gaming (PS-MTG) algorithm, towards the orchestration of microservices across aerOS domains. In this context, the microservices are considered as Slicing Resource Provisioning due to its direct connection with mobile communications domain.

The results of this work have been published in the article: *"Profit-Aware Proactive Slicing Resource Provisioning with Traffic Uncertainty"*

A summary of this work is as follows:

The proposed approach consists of two main steps. First, a Slicing Request Pre-Check Algorithm is developed to verify whether the slicing satisfies predefined conditions related to anticipated bandwidth requirements, slots, and wavelengths. After the pre-check, the set of slice requests that can be served by the tenant is determined. Next, the Slice-Tenant Matching for Credible Prediction Algorithm begins, involving a loop that matches tenants with users. Each user selects the most profitable tenant. If network congestion occurs after serving a user, the algorithm updates resource information and costs for all tenants before finalizing the match and updating the related costs, profits, and paths.

The proposed profit-aware resource provisioning algorithm using a 14-node network was elevated. For this evaluation, 200 services are generated by intercepting and scaling real-world data, with the sliced data traffic constrained to a range of 0-10 Gbps. The delay for each slice is randomly selected within the range of 40 ms to 100 ms, with 10 ms intervals. AI-based (GRU-based) prediction, as discussed in (1), is used to make predictions and assess the credibility of the results. The experiment spans 24 consecutive time steps. The PS-MTG algorithm achieved an overall accuracy of 7.26%, slightly trailing behind the FIX algorithm at 7.95%. However, PS-MTG offers a marginal advantage in terms of overall user benefit accuracy. Additionally, the refusal rate of sliced services across the different schemes was assessed. To highlight the algorithm's effectiveness, the services were intentionally overloaded, which resulted in a higher rejection ratio. Notably, the PS-MTG algorithm demonstrates the lowest service rejection rate, while the FULL algorithm consistently shows a high rejection ratio. In contrast, the FIX algorithm exhibits a fluctuating rejection rate, peaking at 23%.

- **Adaptable Computing and Network Convergence algorithmic approach** (theoretical)

aerOS consortium proposed a fundamental framework called Adaptable Computing-Network Convergence (ACNC), designed to address the challenges of autonomous orchestration of cloud and network resources. ACNC is an ML-aided framework that integrates computing and networking resources to efficiently manage dynamic and voluminous user requests with stringent QoS requirements. Even though this algorithm has not been able to be deployed over aerOS infrastructure (due to several reasons), it is being further explored and has been tested over certain 6G infrastructure conditions.

The results of this work have been published as a pre-print in arXiv: *"Towards a Dynamic Future with Adaptable Computing and Network Convergence (ACNC)"*

A summary of this work is as follows:

ACNC comprises several key components:

- State Recognition and Context Detection: ACNC employs dimension reduction techniques to generate live, holistic, and abstract system states in a hierarchical structure. Continual Learning (CL) is used to classify these system states into contexts, each managed by dedicated ML agents.
- Resource Orchestration: The framework includes an end-to-end orchestrator that collaborates with domain orchestrators (network and computing) to allocate resources efficiently. The orchestration process is closed loop, ensuring that resources are dynamically adjusted to meet changing demands.

As the system size increases, ACNC has demonstrated over a simulated environment optimal performance in terms of energy consumption and total profit. The DDQL-GNN approach in ACNC, which uses Graph Neural Networks (GNNs), outperforms the standard Double-Deep Q-Learning (DDQL) approach, indicating the effectiveness of incorporating graph-shaped system states in decision-making..

Predicting Network Metrics for Managing Mobility and Reallocation (theoretical)

aerOS consortium proposed a prediction-based intelligent network analytics framework so that the allocation of microservices can be done based on the forecasted behaviour of the network. Anticipating user demand and network conditions, would enable proactive adjustments by HLO in aerOS whenever deploying services. Since this approach has not been validated over real aerOS continuums, the demonstration is supported over 6G theoretical infrastructure, including historical data.

The results of this work have been published as a scientific article: *"Network Slice Mobility for 6G Networks by Exploiting User and Network Prediction"*

A summary of this work is as follows:

The work exposed operated within a distributed Cloud-edge-IoT environment, where resource predictions guide decisions on scaling, migrating, or reallocating services. By prioritizing high-value services and leveraging complementary load profiles across servers, the approach ensures that resources are utilized efficiently while reducing the costs associated with mobility/reallocation. It assumes a time-slotted system and uses traffic prediction methods to obtain accurate prediction information. The scheme prioritizes based on their importance and uses the prediction information to decide on scaling up/down or migrating slices to different servers.

Anomaly Prediction and Resource Allocation (theoretical)

aerOS consortium proposed a framework to locate the potential microservices/slices anomalies and decide the resource allocation strategies simultaneously by predicting the users' future requests. Although departing from a slice-only perspective, the alignment with aerOS was permeated across this effort, so that the inner dynamic adjustment (e.g., slice splitting, merging, and scaling) can be applied.

Also, building on top of the work n T3.5 of aerOS (self-awareness), this approach, by monitoring the running status of physical/virtual nodes, the connectivity of physical/virtual links, and the latencies of different service function chains in the sub-slices level and slice level, slice anomaly detection can improve the users' quality of service/experience (QoS/QoE) by ensuring users' specific requests for resources, service latencies, computing capacities, and content availabilities.

The results of this work have been published as a scientific article: *"User Request Provisioning Oriented Slice Anomaly Prediction and Resource Allocation in 6G Networks"*

Markov-Decision Process as the based for joint Service Migration and Resource Allocation in Edge IoT System (theoretical)

aerOS team also proposed a comprehensive approach to address the joint optimization of service migration and resource allocation in Cloud-Edge-IoT computing continuums. Here, the approach had as main target to is to minimize access delay while maintaining service continuity for IoT users in a dynamic environment characterized by user mobility and constrained edge server resources.

The results of this work have been published as a scientific article: *"Joint Service Migration and Resource Allocation in Edge IoT System Based on Deep Reinforcement Learning"*

. The proposed methodology relies on a deep reinforcement learning (DRL)-based algorithm to dynamically adapt to changes in system conditions and user requirements. The problem of allocation is formulated as a Markov Decision Process (MDP) with defined states, actions, and a reward function. The state includes information about predicted user locations, edge server resource availability, and user-server associations. The actions consist of binary migration decisions and continuous resource allocation parameters. The reward is designed to incentivize minimizing total task processing delays. The algorithm offers a scalable and intelligent solution for optimizing service migration and resource allocation in aerOS computing continuums. Its integration of mobility prediction, hybrid action space handling, and DRL-based optimization ensures enhanced service continuity, reduced delays, and improved resource utilization.

A Priority-Aware Energy-Efficient Approach for Latency-Sensitive Applications (theoretical)

aerOS consortium proposed a priority-aware solution for the autonomous orchestration of cloud and network resources that had as basis the configuration of 6G networks. Applicability to aerOS would be demonstrated in a future initiative.

The results of this work have been published as a pre-print in arXiv: *"ORIENT: A Priority-Aware Energy-Efficient Approach for Latency-Sensitive Applications in 6G"*

A summary of this work is as follows:

The approach is designed to address the joint problem of service instance placement and assignment, path selection, and request prioritization, collectively referred to as PIRA (Placement, Instance Assignment, Request Prioritization, and Allocation). The primary objective is to maximize the system's overall profit, defined as a function of the number of concurrently supported requests, while minimizing energy consumption over time. This is achieved while considering end-to-end latency requirements and resource capacity constraints. The proposed approach leverages a combination of Double Dueling Deep Q-Learning (D3QL) and GNNs to encode the state of the system and make optimal resource allocation decisions. The solution is particularly suited for latency-sensitive applications in 6G, where stringent QoS requirements must be met efficiently, however, as mentioned, it would be for interest in the regular aerOS implementation cases.

Customizable Hybrid Isolation for Vertical Slicing approach (theoretical)

aerOS consortium proposed a novel flexible hybrid isolation model and addresses challenges in slice resource provisioning with uncertain traffic in transport networks. After this scientific work under task T3.3 of aerOS, that targets resource (mobile network slices) allocation, a dynamic programming algorithm efficiently handles grouping, and an iterative adjustment algorithm fine-tunes resource allocation based on probabilistic analysis.

The results of this work have been published as a scientific article: *"Probabilistic-Assured Resource Provisioning With Customizable Hybrid Isolation for Vertical Industrial Slicing"*

Multi-Agent Actor-Critic (MAAC) algorithm for Heterogeneous Edge Caching Learning with Attention Mechanism Aiding approach (theoretical)

aerOS consortium proposed a novel multi-agent, neighbour-aware actor-critic (NAC) framework, inspired by the Multi-Agent Actor-Critic (MAAC) algorithm was developed in order to optimize edge caching strategies. The work uses an attention-based multi-agent caching replacement strategy. Agents can learn from neighbouring Base Stations (BSs), improving caching decisions through shared knowledge. Consequently, caching states can be exchanged between BSs, facilitating better information sharing, such as content size and type. In this approach, both time and space factors were incorporated as observations to analyse the influence between BSs through the critic network, using an attention mechanism. Each BS, acting as an agent, has its own critic network and can observe the historical caching states of neighbouring BSs. This process combines distributed local training with centralized global learning.

The results of this work have been published as a scientific article: *"Heterogeneous Edge Caching Based on Actor-Critic Learning With Attention Mechanism Aiding"*

Joint Network Slicing, Routing, and In-Network Computing approach (theoretical)

aerOS consortium proposed a slicing-based solution for the autonomous orchestration of computing continuum network resources, particularised in next-generation mobile networks.. The solution involves formulating a Mixed-Integer Linear Programming (MILP) problem that considers end-to-end capacity and QoS constraints. Given the NP-hard nature of the problem, a heuristic algorithm is proposed, WF-JSRIN (Water Filling-based Joint Slicing, Routing, and In-Network Computing), which provides near-optimal solutions with significantly reduced execution times compared to optimal approaches. This makes it highly suitable for practical real-world applications, particularly in the context of autonomous resource orchestration. The goal was to align with aerOS principles and to maximize the number of accepted users while minimizing energy consumption, thereby ensuring sustainable and efficient network operations

The results of this work have been published as a scientific article: *"Joint Network Slicing, Routing, and In-Network Computing for Energy-Efficient 6G"*

3.3.2. Structure diagram

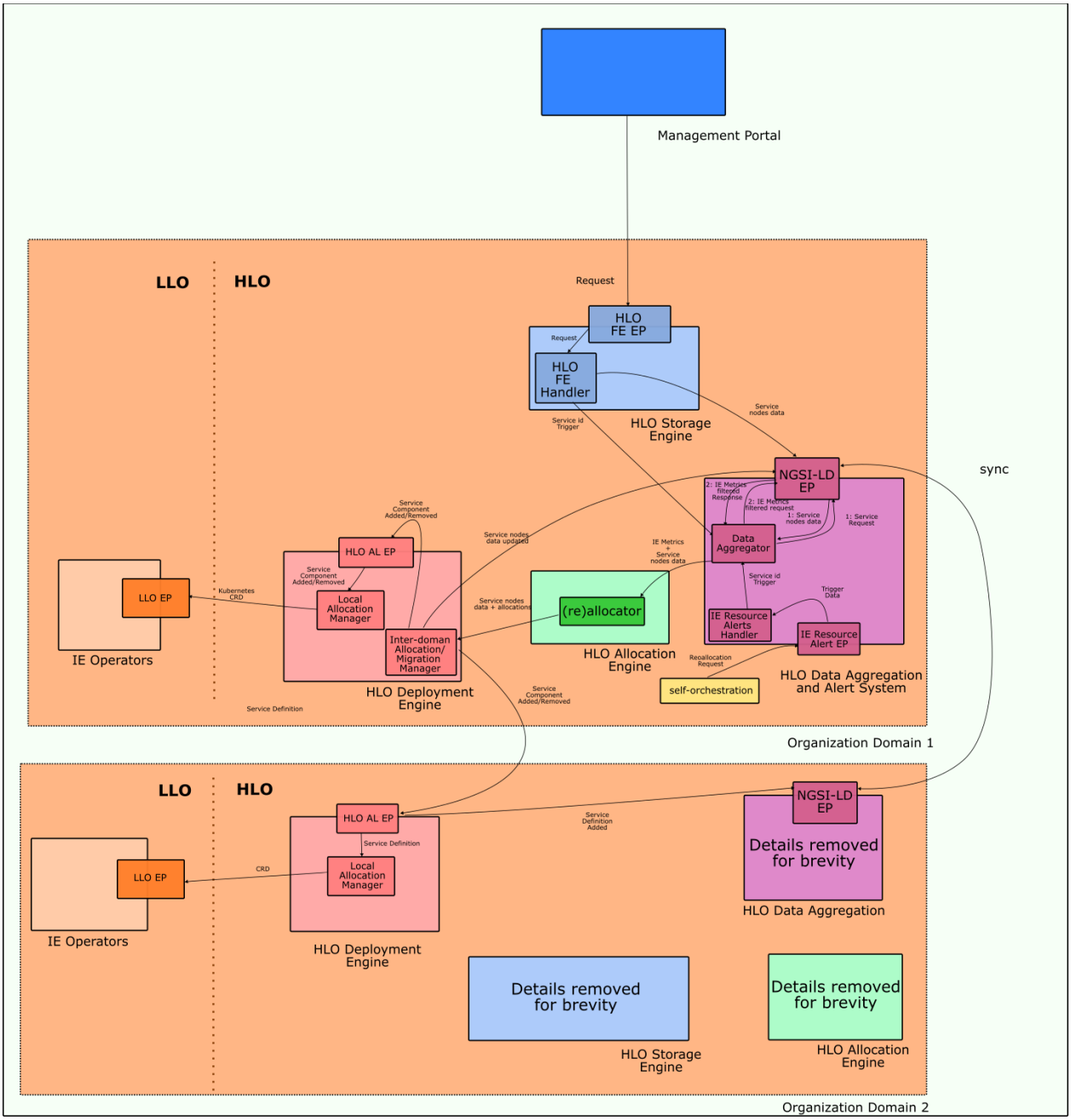


Figure 47. aerOS High-Level Orchestration Components

Table 18. 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	**Management portal,** located in entrypoint domain. Receives TOSCA descriptor through REST endpoints. **Orion CB,** to which it pushes (using ngsi-ld API) service requirements**.**

	from the aerOS entrypoint 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.	**Data Aggregator** is triggered to proceed 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	**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.

	internal components (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	**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.

	exposes Rest API for LCM 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).

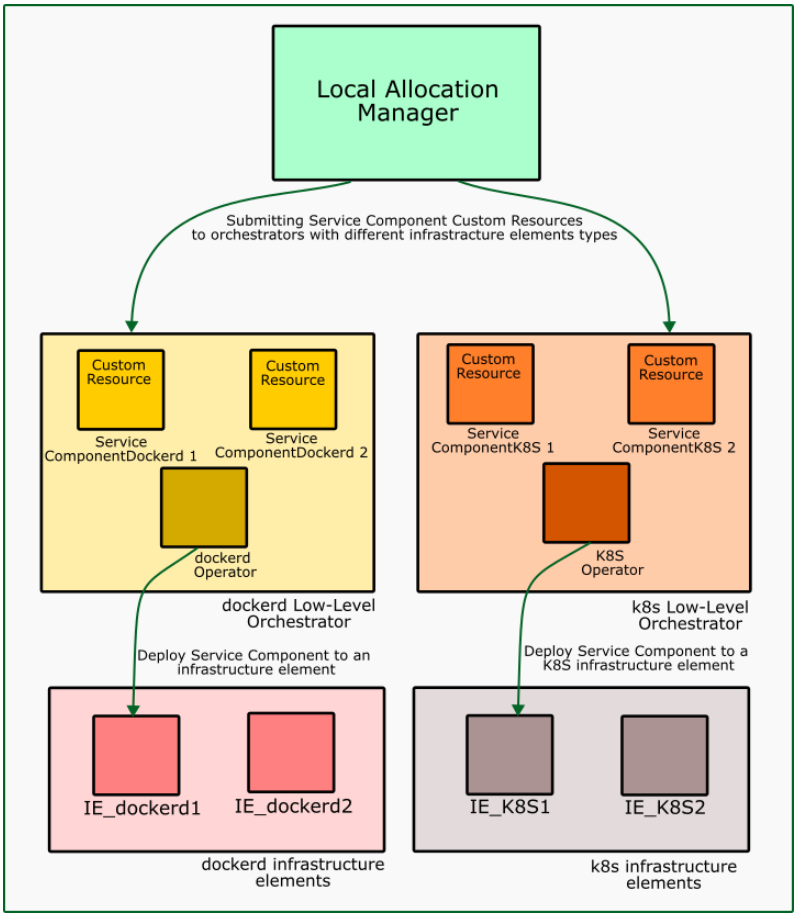


Figure 48. aerOS Multi Low-Level Orchestration Components

Table 19. aerOS Multi Low-Level Orchestration Components' description

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 allocation of the service components in a specific infrastructure element of the domain.	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 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

3.3.3. Technologies and standards deployed in MVP

Table 20. Technologies and standards deployed in MVP

Technology/Standard	Description	Justification
Protobuf	A language-agnostic data serialization format developed by Google. It's 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. 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's 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.
Operator SDK	Go is a simple and efficient programming language developed by Google, which is used in most 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)	Low level orchestrators are based on Kubernetes operators, so the most used and mature framework for developing them should be tested and used, among other alternatives. Furthermore, it uses Go, which is the most common language for building K8s native applications.

	Kubernetes Operators.	
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.

3.4. Cybersecurity components

The definition of aerOS AAA, namely Authentication, Authorisation, and Accountability, shows the importance of creating a comprehensive security framework that ensures secure access, trust and transparency in the project's resources. The embodiment of these concepts and their integration into the project was realized through the technical implementations carried out in Task 3.4. More specifically, the terms Authentication and Authorisation were covered by the aerOS IDM, while Accountability by the aerOS Secure API gateway. The combination of these three concepts through the technical implementation enhance data security and ensures operational integrity from insider threats or external attacks.

The following subsections describe the final result developed in task 3.4 since D3.2, namely during the months M19-M30. More specifically there is a thorough description of the components, an architectural diagram of aerOS AAA infrastructure which illustrates the relationships between the components, and a final subsection which describes the technologies and protocols deployed in MVP v2.

3.4.1. Main functionalities

As aforementioned, the aerOS Identity Management (IdM) and aerOS API Gateway components are essential for ensuring secure and efficient operation within the aerOS ecosystem, each serving a different purpose. The main objective of the aerOS IdM is to provide secure and dependable authentication and authorization for aerOS clients. Also, it prevents unauthorized access, by implementing advanced security mechanisms, such as token-based authentication and Single Sign-On (SSO) through the usage of OpenID Connect (OIDC). Additionally, it enforces Role-Based Access Control (RBAC), assigning users specific roles that determine their access to resources and data, thereby aligning access privileges with organizational policies.

The API Gateway is designed to provide a centralized entry point for all interactions with aerOS components. Its purpose is to streamline communication, eliminate redundancy from multiple access points, and enforce security policies. It applies access controls defined by the IdM system based on user roles and groups. Additionally, the API Gateway plays a critical role in safeguarding the aerOS Data Fabric by managing API-level security and preventing unauthorized interactions, ensuring the integrity and confidentiality of the system’s data.

The progress of aerOS IdM and aerOS API Gateway is described in the following sub-sections.

3.4.1.1. aerOS Identity and Access Management

The Identity and Access Management (IAM) of aerOS, as discussed in D3.1, has been based on Keycloak[14], while the authentication and authorization has been performed using the OpenID Connect (OIDC) protocol and access has been granted to aerOS users based on their roles (i.e., Role-based Access Control). Keycloak,

14 https://www.keycloak.org/

OIDC, and RBAC thoroughly presented in D3.1; hence, in D3.3 there is no further elaboration on these tools and protocols.

In this deliverable, the advances in IAM are discussed presenting the intermediate implementation of Keycloak, OIDC, and RBAC. Furthermore, enhancing the IAM of aerOS the consortium decided to implement Keycloak with OpenLDAP[15] 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 49.

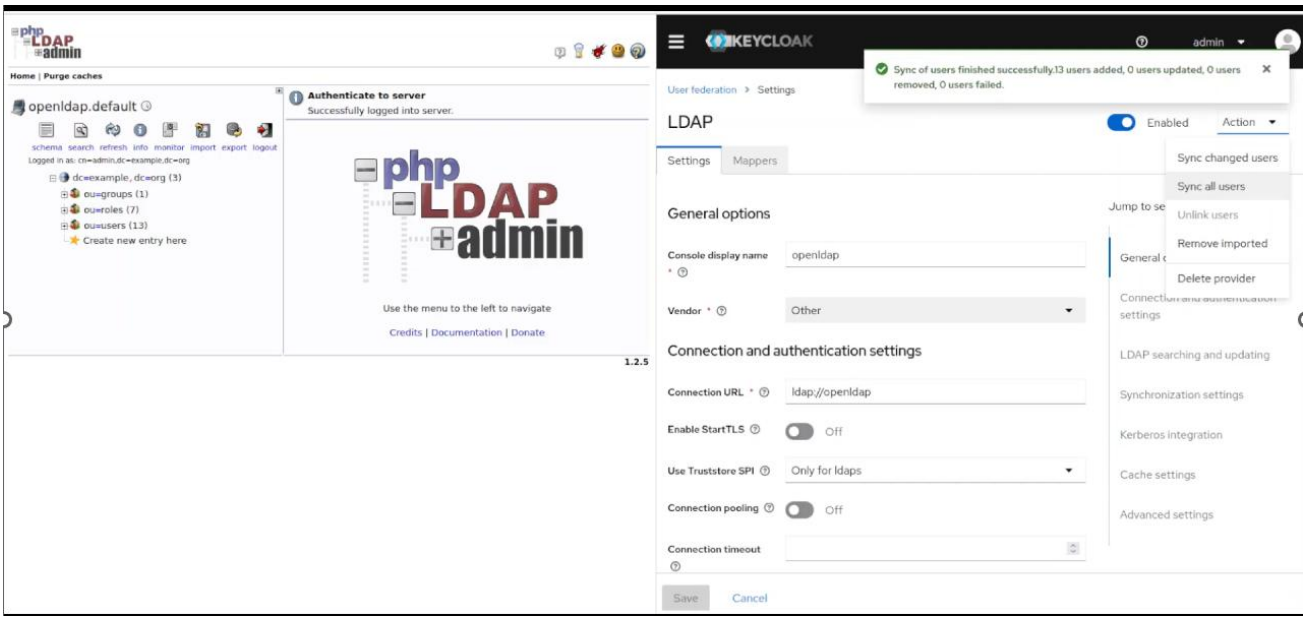


Figure 49. Synchronisation of OpenLDAP users in Keycloak.

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 (after analysing the needs of the project, the users have been updated):

- Continuum administrator: can access all the aerOS services, generate new deployments, access all the data (read only) and is able to generate new users.
- Data product owner: can create new data sources that will be integrated in the domain.
- Vertical deployer: can deploy new services in his domain.
- aerOS user: can consume the data of his domain, but it has not any permissions to create new data or change the configuration of the domain.

In order to facilitate the installation of the aerOS IdM, a new version of OpenLDAP has been packaged in a Helm chart where the default users defined in the project have been created. Once the first Keycloak connection to the OpenLDAP has been set up, the new users/groups/roles are managed directly from the Management Portal. Figure 50 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. As the groups are generated using the Management Portal, it has been decided to only generate a Default group in the first OpenLDAP installation.

15 https://www.openldap.org/

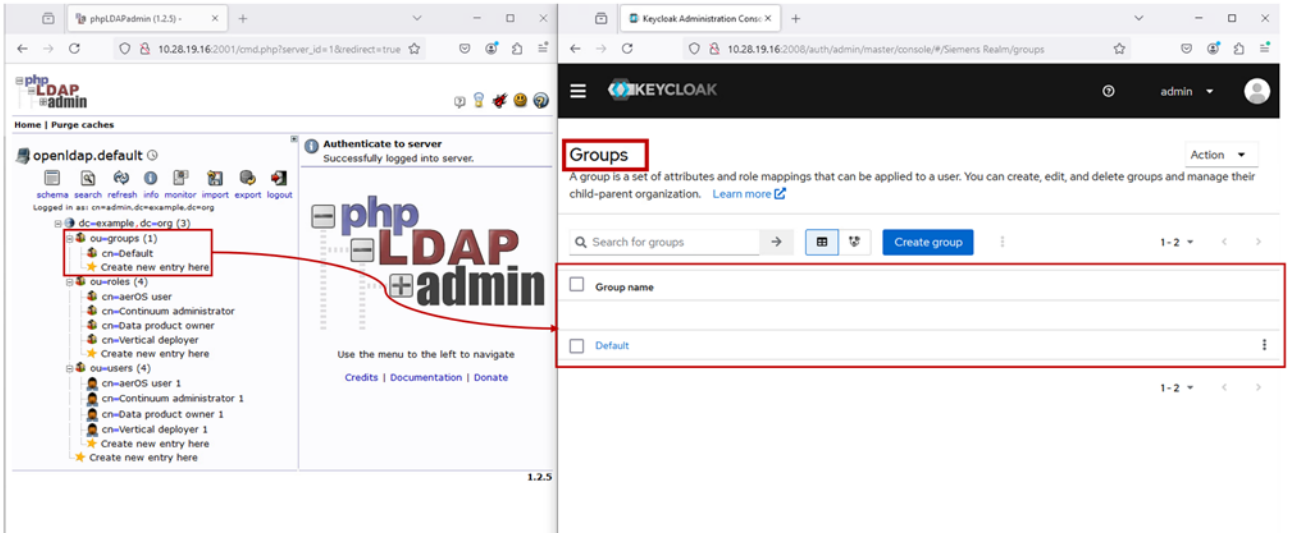


Figure 50. Groups generated for 2nd MVP in OpenLDAP (and federated in Keycloak)

Figure 51 shows the roles generated by default in the OpenLDAP image and the federation of the OpenLDAP image in the Keycloak. In Figure 14 the same can be seen with the default generated users.

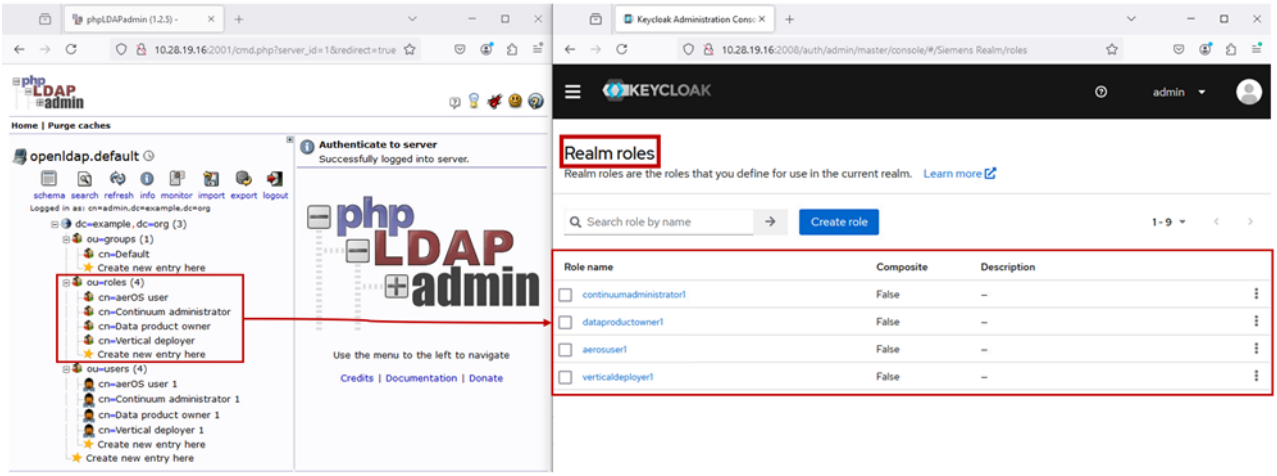


Figure 51. Roles generated for 2nd MVP in OpenLDAP (and federated in Keycloak)

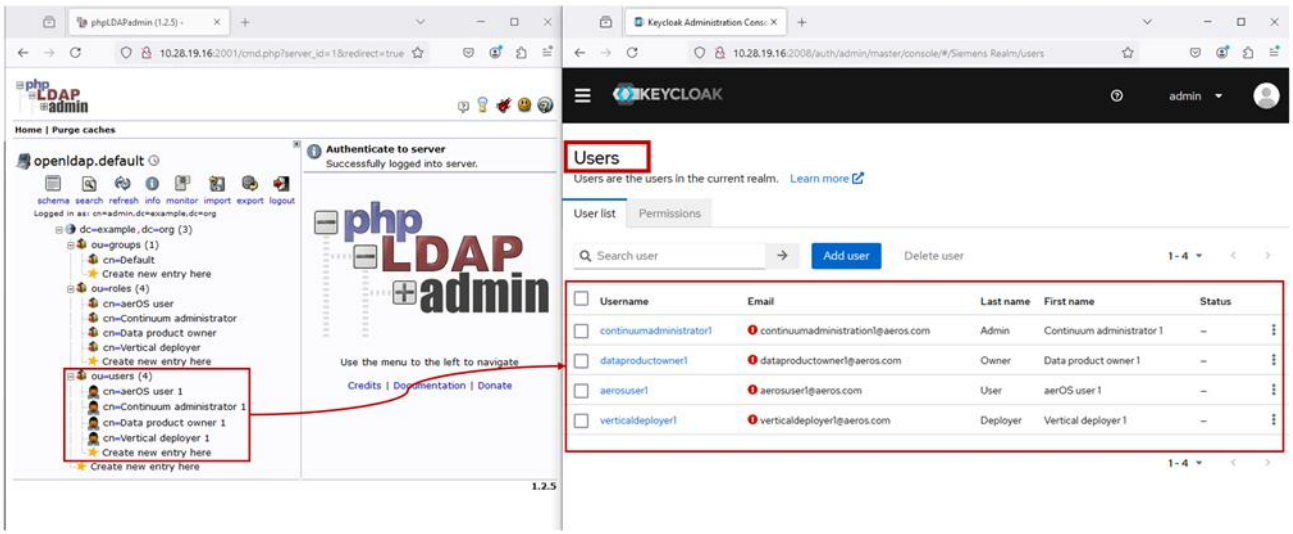


Figure 52. Users generated for MVPv2 in OpenLDAP (and federated in Keycloak)

3.4.1.2. 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 GO that is used to fill the architecture gap about the gateway. It is 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 is 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[16] showcases these features:

16 https://www.krakend.io/docs/overview/

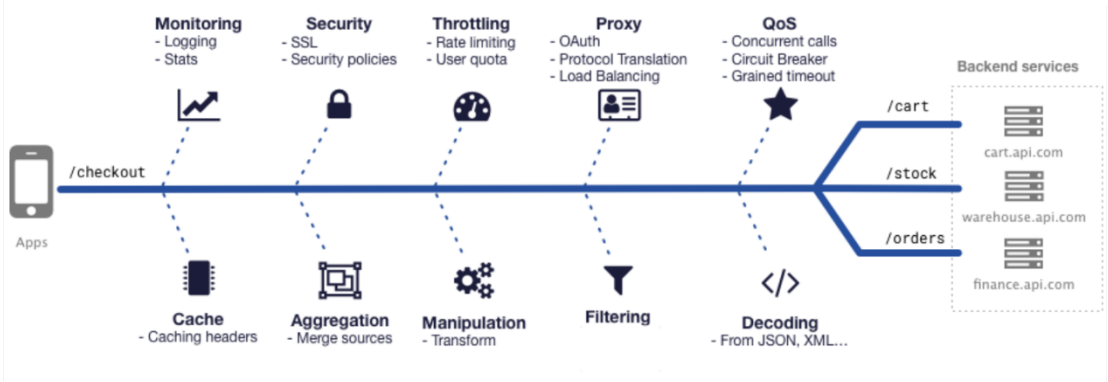


Figure 53. KrakenD and its capabilities

KrakenD has been successfully deployed in the entrypoint domain of the MVPv2, 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. Additional functionalities were added as time went on and the different components required extra endpoints such as the implementation with the federator, the aerOS Portal and IOTA. A simplified installation process via helm charts has been integrated alongside the new installation instructions. Connection to the Keycloak to retrieve user tokens is still necessary although token caching has been implemented to reduce latency.

3.4.2. Structure diagram

Figure 16 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.

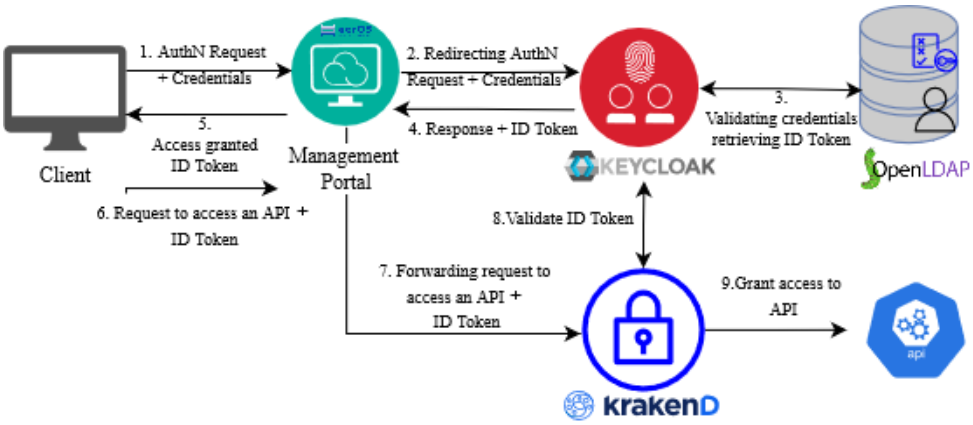


Figure 54. aerOS Authentication, authorization, and access control

Table 21. 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/group/role 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.

3.4.3. Technologies and standards deployed in MVP

The aerOS AAA components have been deployed in the MVPv2 in order to demonstrate the cybersecurity capabilities and protection mechanisms for a user that wants to access the OrionLD API. In this set up, 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 55, KrakenD retrieves the access token from the IAM, using its public IP, as well as the special API endpoint created to get the tokens.

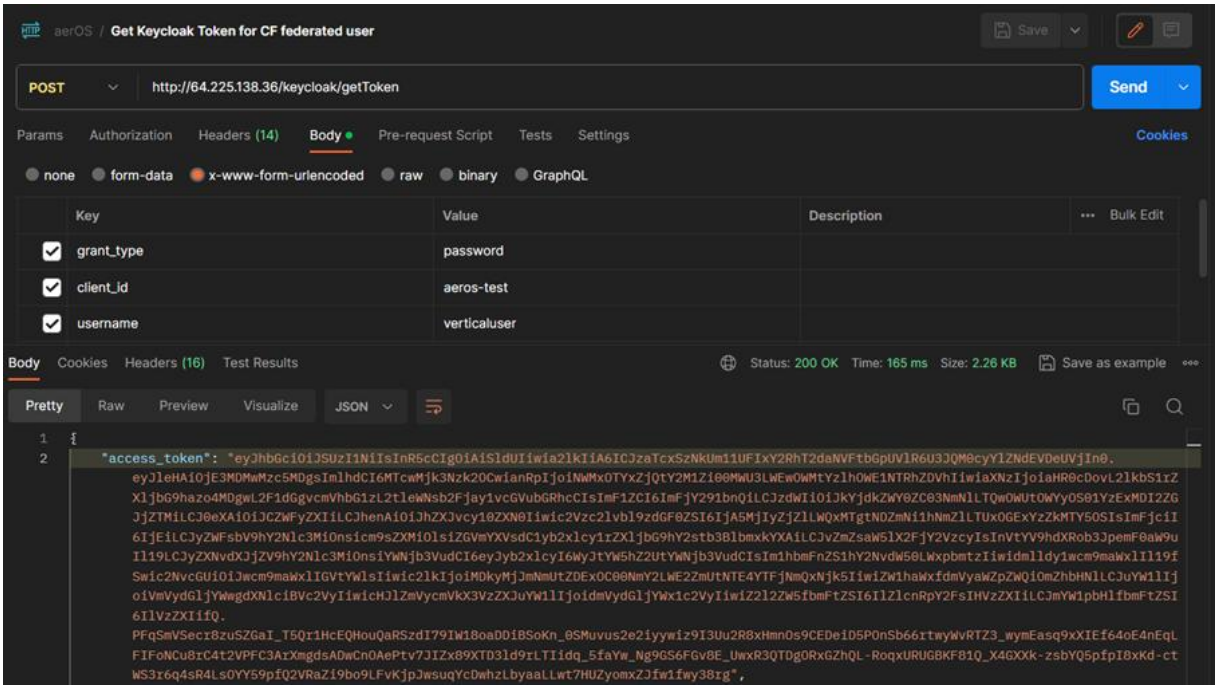


Figure 55. KrakenD retrieving access token from Keycloak

Afterwards, as depicted in Figure 56, the access token could be used to access an aerOS API, such as Orion-LD endpoint.

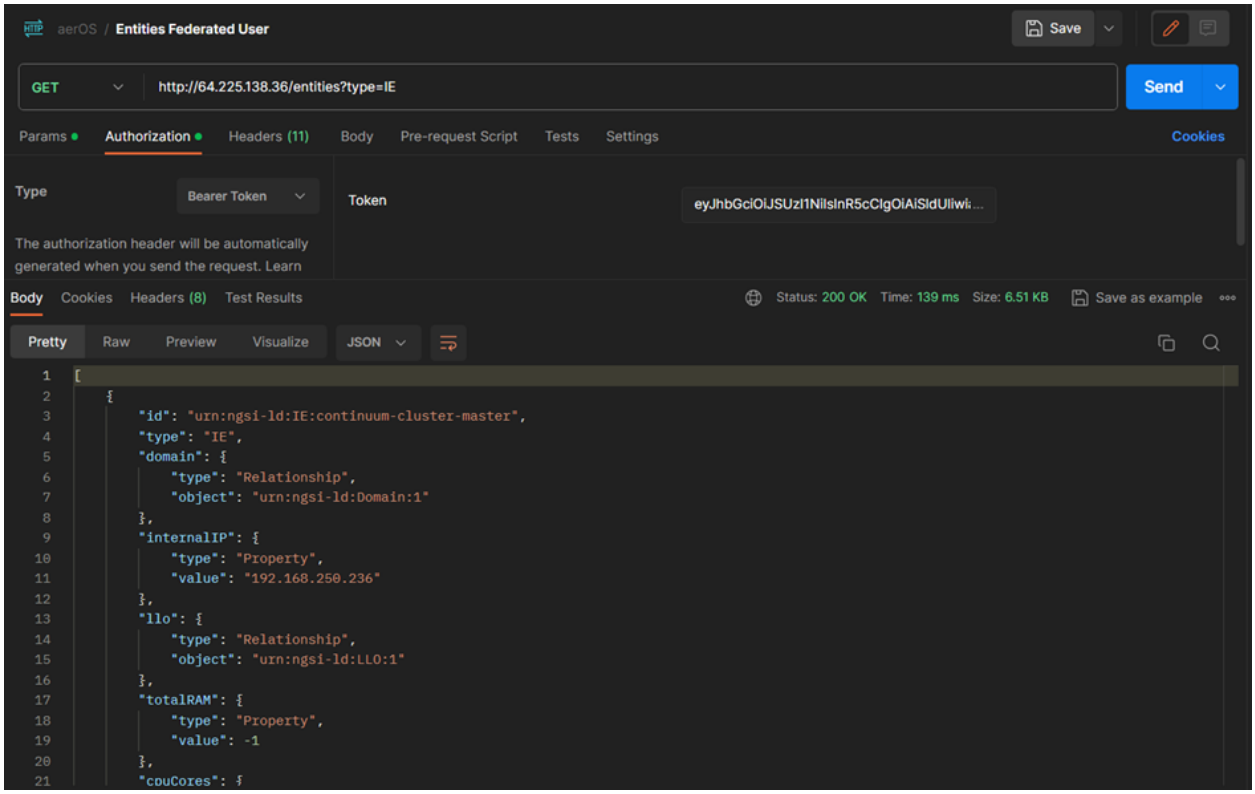


Figure 56. Deploying token to access an aerOS API

Table 22. 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.

3.5. Node's self-x 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 autonomously, 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 shall 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.

3.5.1. 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, 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 (*power_consumption*) is in charge of obtaining the energy consumption of the node, which requires more computing time. The other (*hardware_info*) is responsible for obtaining the rest of the parameters. The component that obtains the power consumption needs an average of 20-25 seconds per execution to obtain new valid values and the other only needs about 3-15 seconds to update its information, depending on the amount of information to be collected by the sub-module. This amount is specified by environment variables in the sub-module deployment files. 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. Currently, this self-* capability is able to obtain the following information from each node: hostname, addresses (internal IP and MAC), CPU (architecture, number of cores, max frequency and current usage), RAM (total capacity, available capacity and current usage), disk (type, total capacity, available capacity and current usage), network (speed up, speed down, traffic up, traffic down and lost packages), power consumption (current and average), capability to execute workloads in real-time and operating system. To obtain all these parameters, both sub-modules use external packages and libraries such as *PowerTOP*, *iproute2*, *psutil*, *getmac* or *speedtest-cli*, as detailed below. On the other hand, each sub-module has a REST API that allows the sampling frequency of the information in each node to be set independently. This makes it possible to optimise the operation of the node within the domain to which it belongs and reduce the consumption of resources. Below are screenshots of the two sub-modules running on the continuous development and integration cloud infrastructure of the project (provided by partner CF).

```
Name:             self-awareness-hardwareinfo-9bfnp
Namespace:        default
Priority:         0
Service Account:  default
Node:             aeros-2-jms6qnflylil-node-1/10.0.0.238
Start Time:       Thu, 13 Feb 2025 11:20:53 +0100
Labels:           app.kubernetes.io/component=hardwareinfo
                  app.kubernetes.io/instance=self-awareness
                  app.kubernetes.io/managed-by=Helm
                  app.kubernetes.io/name=self-awareness
                  app.kubernetes.io/version=1.4.0
                  controller-revision-hash=5dc98b87c5
                  helm.sh/chart=self-awareness-1.4.0
                  isMainInterface=yes
                  pod-template-generation=1
                  tier=external
Annotations:      kubernetes.io/psp: magnum.privileged
Status:           Running
IP:               10.0.0.238
IPs:
  IP:           10.0.0.238
Controlled By:  DaemonSet/self-awareness-hardwareinfo
Containers:
  hardwareinfo:
    Container ID:   docker://7c2431e9acb19cf71b5fa936a806517dab3815028a70118681bba40f44913957
    Image:          registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-awareness/hardware_info:1.4.0
    Image ID:       docker-pullable://registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-awareness/ha
56:53de2f47fa19dd0ba283ce8c9382ea6a4658b69768eadd0f8b5a46e2ab6399ed
    Port:           8002/TCP
    Host Port:      8002/TCP
    State:          Running
      Started:      Thu, 13 Feb 2025 11:20:56 +0100
    Ready:          True
    Restart Count:  0
    Environment:
```

Figure 57. Hardware info sub-module running on a test cluster of infrastructure

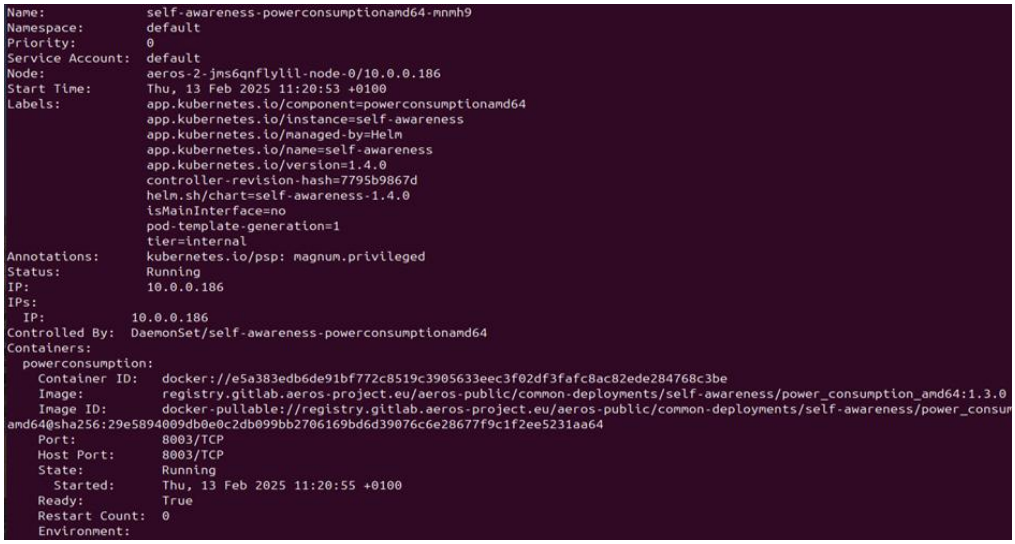


Figure 58. Power consumption sub-module running on a test cluster of infrastructure

- **Self-orchestrator:** considered one of the main self-* capabilities of an autonomous system, this component is composed by a *Rules Engine*, a *Facts Generator*, a *Trigger* and wrapped by a *REST API*. It is capable of managing facts, rules and alerts, obtaining information from the self-awareness, self-realtimeness, self-healing and self-optimisation and adaptation modules to send warnings about problems in the IE to the aerOS EAT (Embedded Analytics Tool) of the domain, 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. This self-* capability uses libraries such as *json-rules-engine* or *jsonschema*, as detailed below.

  In order to be able to manage the rules for detecting faults, malfunctions or anomalous situations within a node, the module exposes a *REST API* that allows CRUD (Create, Read, Update and Delete) operations to be performed on these rules. Moreover, by means of persistent storage in the node, this module maintains an updated backup copy of the rules to restore them to their last state in case of failure in the IE.

  On the other hand, this *REST API* also allows to receive alerts in a predefined format coming from other self-* modules in order to carry out the necessary corrective measures through the aerOS EAT. In addition, when a rule is triggered or an alert is received at the corresponding *REST API* endpoint, a message is sent to the domain's IOTA hornet node to register the event.

  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-orchestrator-b6dcv
Namespace:        default
Priority:         0
Service Account:  default
Node:             aeros-2-jms6qnflylil-node-0/10.0.0.186
Start Time:       Thu, 06 Feb 2025 10:51:54 +0100
Labels:           app.kubernetes.io/component=orchestrator
                  app.kubernetes.io/instance=self-orchestrator
                  app.kubernetes.io/managed-by=Helm
                  app.kubernetes.io/name=self-orchestrator
                  app.kubernetes.io/version=1.2.0
                  controller-revision-hash=657c67c45c
                  helm.sh/chart=self-orchestrator-1.2.0
                  isMainInterface=yes
                  pod-template-generation=4
                  tier=self
Annotations:      kubernetes.io/psp: magnum.privileged
Status:           Running
IP:               10.0.0.186
IPs:
  IP:           10.0.0.186
Controlled By:  DaemonSet/self-orchestrator-orchestrator
Containers:
  orchestrator:
    Container ID:   docker://27010fc4d52d8fd479f753a4bc79a9c05437a9e153c46a1a869fbd8906bd0767
    Image:          registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-orchestrator:1.2.0
    Image ID:       docker-pullable://registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-o
a9be96591684151ebdde18e40ef720a6a662181259c4d0b2a2788496
    Port:           8001/TCP
    Host Port:      8001/TCP
    State:          Running
      Started:      Thu, 06 Feb 2025 10:51:55 +0100
    Ready:          True
    Restart Count:  0
    Environment:
```

Figure 59. Self-orchestrator module running on a test cluster of infrastructure

- **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 end-point (Trust Manager). These alerts allow to discover different types of network attacks to detect vulnerabilities and threats at the IE level. An API has also been created so that the Trust Manager can make requests each week, with the intention of collecting the alerts for the whole week and saving them in the history. The alerts generated by self-security are deleted once a week by a new service with the intention of minimising the space that this component occupies on disk.

  Below is an example of the alerts generated by the self-security running on the continuous development and integration cloud infrastructure of the project (provided by partner CF).

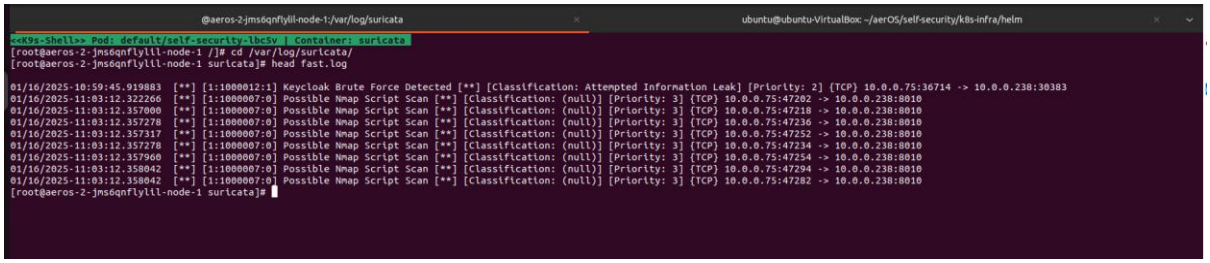


Figure 60. Self-security alert example

- **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.

  Below is a screenshot of the self-API module running on the project's continuous development and integration cloud infrastructure (provided by partner CF):

```
Name:             self-api-api-6nc9s
Namespace:        default
Priority:         0
Service Account:  default
Node:             aeros-2-jms6qnflylil-node-8/10.0.0.158
Start Time:       Wed, 26 Feb 2025 15:46:05 +0100
Labels:           app.kubernetes.io/component=api
                  app.kubernetes.io/instance=self-api
                  app.kubernetes.io/managed-by=Helm
                  app.kubernetes.io/name=self-api
                  app.kubernetes.io/version=1.0.0
                  controller-revision-hash=5cdd9cd5fd
                  helm.sh/chart=self-api-1.0.0
                  isMainInterface=yes
                  pod-template-generation=1
                  tier=self
Annotations:      kubernetes.io/psp: magnum.privileged
Status:           Running
IP:               10.0.0.158
IPs:
  IP:           10.0.0.158
Controlled By:  DaemonSet/self-api-api
Containers:
  api:
    Container ID:   docker://44a9de63e9d3aeee0de295dfad45d63c0ac55e99a1e854a6d4399da84274f265
    Image:          registry.gitlab.aeros-project.eu/aeros-public/common-deployments/self-api:latest
    Image ID:       docker-pullable://registry.gitlab.aeros-project.eu/aeros-public/common-deployments
    Port:           8600/TCP
    Host Port:      8600/TCP
    State:          Running
      Started:      Wed, 26 Feb 2025 15:46:19 +0100
    Ready:          True
    Restart Count:  0
    Environment:
      PORT:                    8600
      URL:                     0.0.0.0
      SELF_ORCHESTRATOR_IP:     (v1:status.podIP)
      SELF_ORCHESTRATOR_PORT:  8001
      SELF_DIAGNOSE_IP:         (v1:status.podIP)
      SELF_DIAGNOSE_PORT:      8002
      SELF_SECURITY_IP:         (v1:status.podIP)
      SELF_SECURITY_PORT:      8000
      SELF_OPTIMIZATION_IP:     (v1:status.podIP)
      SELF_OPTIMIZATION_PORT:  8090
      SELF_HEALING_IP:          (v1:status.podIP)
      SELF_HEALING_PORT:       8500
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-nknlv (ro)
```

Figure 61. Self-API module running on node-8 of test K8s cluster-2 of infrastructure

- **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. This module detects and remedies abnormal states of the network, outlier values of sensors connected to the IE, and issues with the IE's power level. Since the self-healing module detects abnormal states or outlier values, it generates a JSON alert message and sends this message to the Trust Manager component for the health score calculation algorithm. The following is an example of a POST JSON message:

```
{
    "timestamp": "2025-02-04T11:00:00",
    "scenario": "Sensor Failure",
    "message": "Sensor measurement detected as an outlier. Exclude the sensor from the set of those that provide input to the system",
    "mac_address": "fa:16:3e:5e:25:ef"
}
```

Figure 62. Example of JSON alert from self-healing to Trust Manager

Furthermore, the module collects these alert messages generated from scenarios and exposes a GET API endpoint, which is consumed by self-API component. An example of these JSON alerts is shown below:

```
{
  "alerts": [
    {
      "timestamp": "2025-02-04T11:00:00",
      "scenario": "sensor_failure",
      "message": "Sensor failure detected",
      "mac_address": "fa:16:3e:5e:25:ef"
    },
    {
      "timestamp": "2025-02-04T11:00:00",
      "scenario": "network_violation",
      "message": "Network duty cycle exceeded",
      "mac_address": "fa:16:3e:5e:25:ef"
    }
  ]
}
```


Figure 63. Example of JSON alerts from self-healing to self-API

- **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 has focused on evolving an existing open source tool (originated in H2020 project AS-SIST-IoT), by integrating the innovative needs of aerOS and incorporating automated configuration.
- **Self-optimisation and adaptation:** ability to optimise the dissemination of the data and control the performance of IE. On the one hand side, by using dynamic sampling techniques and the current metric streams (i.e. the IE's operational data obtained from self-awareness), the component suggests optimal sampling periods, allowing control over the frequency of data monitoring by the self-awareness component. On the other hand, it incorporates an estimation model monitoring the shifts in the metric streams to detect data points with significant differences that may indicate potential anomalies, aiming to prevent the overutilization and underutilization of IE resources.
- **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 re-orchestration request as illustrated in the following figure:

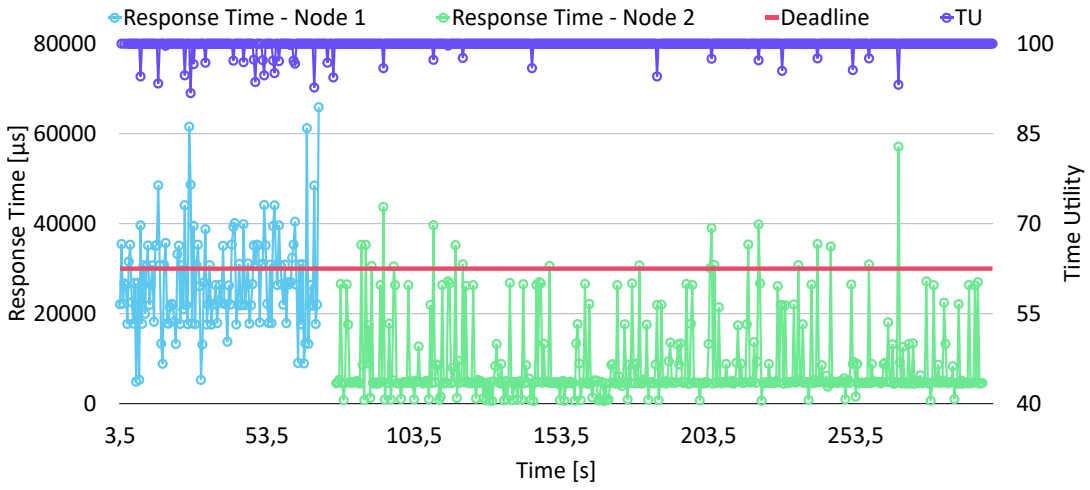


Figure 64. Self-realtimeness relocating real-time workload with bad time-utility from node 1 to node 2 (2)

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

3.5.2. Structure diagram

The aerOS self-* capabilities set is comprised of 9 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-security and self-API. The non-core modules are self-configuration, self-healing, self-optimisation and adaptation, self-realtimeness and self-scaling. To offer a clearer vision of the set, a diagram has been created that represents the interactions between the different components and tools of the set.

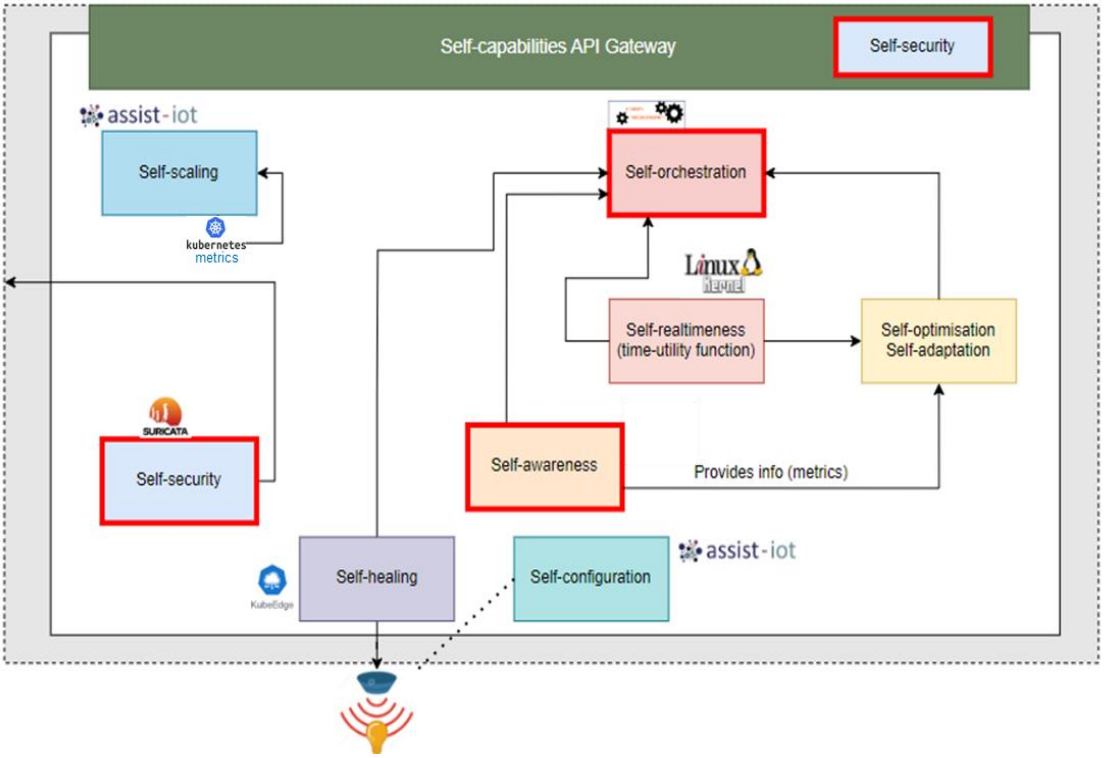
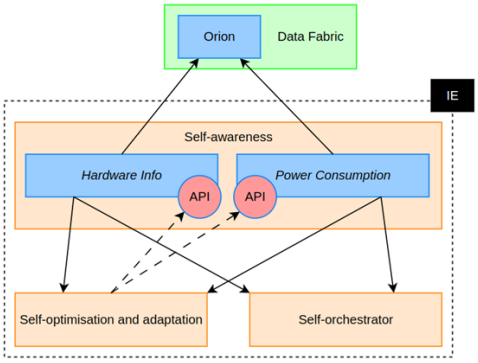


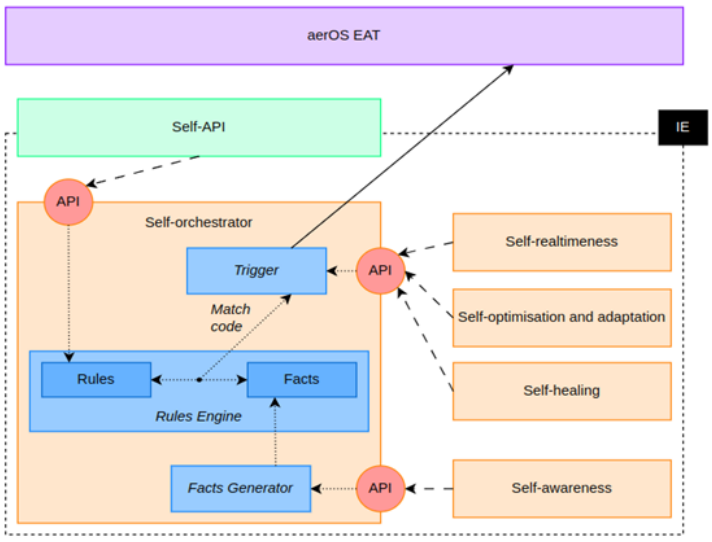
Figure 65. Relationships between the different self- capabilities of an IE*

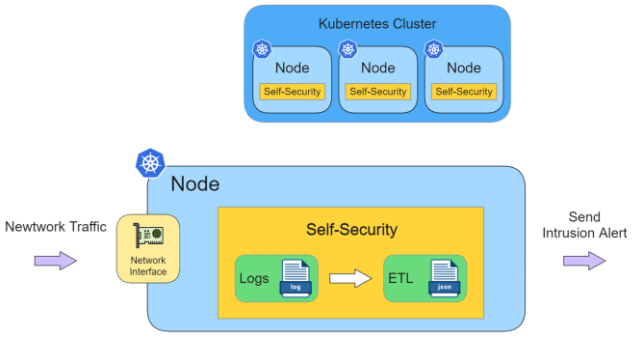
When an aerOS computing continuum node has IoT peripheral devices connected, self-configuration and self-healing modules can be installed in the IE. These systems continuously analyse the health status of these devices, sending alerts to the self-orchestrator module in case of failure or malfunction so that it communicates with the aerOS EAT in order to improve the management and coordination of the node's workloads. 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-healing and self-optimisation and adaptation, determines whether to send alerts to the aerOS EAT. 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, and 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. 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.

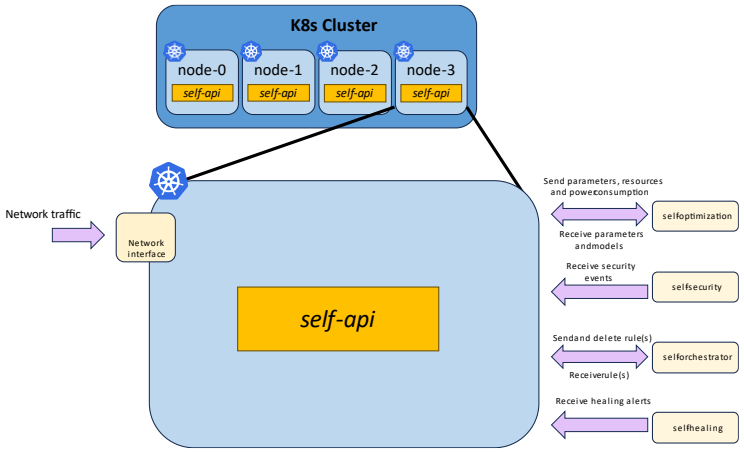
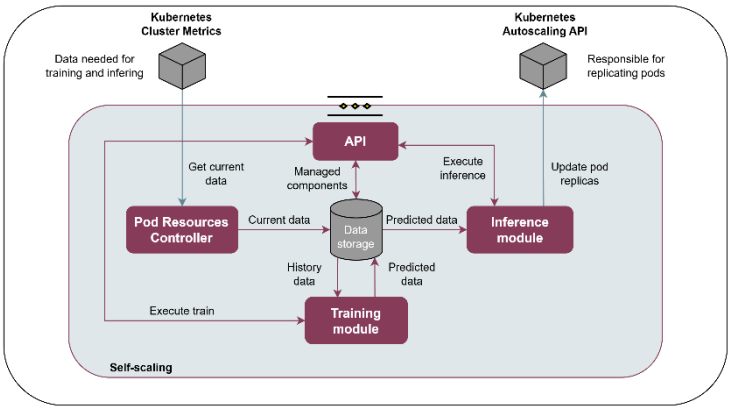
In the next table, the specific functionalities, details and interactions are further described:

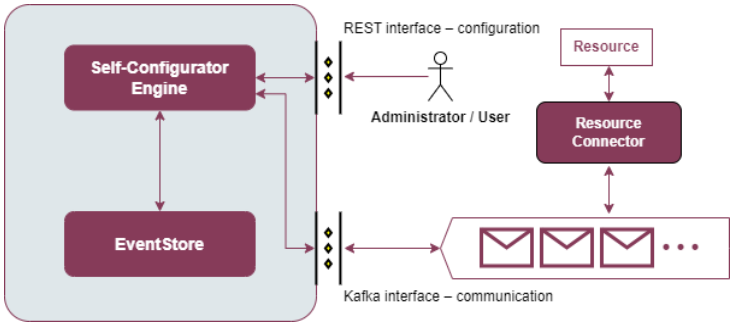
Table 23. Self- capabilities components, description and interactions*

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 self-* components, and is divided into two sub-modules. This module can: • Obtain parameters such as hostname, addresses (internal IP and MAC), CPU (architecture, number of cores, max frequency and current usage), RAM (total capacity, available capacity and current usage), disk (type, total capacity, available capacity and current usage), network (speed up, speed down, traffic up, traffic down and lost packages), power consumption (current and average), capability to execute workloads in real-time and operating system. • Define custom parameters such as Infrastructure Element ID, Infrastructure Element Tier and Infrastructure Element Status. • Works on Kubernetes clusters and Docker, on AMD64 and ARM64 architectures and physical or virtual machines. • It is capable of inform about their health status in "real-time". • Sends data periodically (the sampling period may vary through its API). This is the current schema of the development carried out in the module: From the last iteration of this deliverable until M30 of the project, the connection with the self-optimisation and  *Figure 66. Self-awareness schema* adaptation module has been carried out and tested, the amount and type of information that the module is capable of capturing in each installed IE has been extended, the data model used to define an IE has been refined and adapted, and the option to modify the sampling period through an exposed endpoint has been added.	It obtains information about the state of the node and directly feeds the self-orchestrator and the self-optimisation and adaptation modules. Additionally, it provides context information to the Context Broker associated with that IE.

	Moreover, it has been tested in more types of different IE and it has been deployed in several continuums (including those of the Pilots of the project) to check the reliability and performance of the module, as well as to determine that it is free of faults. Currently, the module is finalised, unless bugs are detected and need to be corrected due to the tests carried out. The next steps are to continue integrating the module into the Pilots' domains to enable full synergy between self-awareness and the other components of each of their domains.	
Self-orchestrator	This self-* capability allows to interact with aerOS Embedded Analytics Tool. This module is composed of: • *Rules Engine*: contains the rules and facts (rule activation thresholds) to be evaluated. These rules represent the situations in which it is necessary to send an alert to the aerOS EAT of the domain where the node is located. The facts represent the current state of the IE and the network, and are fed directly from 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. • *REST API*: allows to execute CRUD (Create, Read, Update and Delete) actions dynamically on the rules stored in the rules engine, insert facts and receive alerts from other self-* modules. • *Trigger*: generates alerts from the self-orchestrator and sends them to the aerOS EAT of the domain where the node is located. This is the current schema of the development carried out in the module:  *Figure 67. Self-orchestrator schema*	It obtains information directly from four components: (1) self-awareness (values to generate the facts), (2) self-healing, (3) self-realtimeness (to 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 corrective or compensatory actions in advance).

	From the last iteration of this deliverable until M29 of the project, the last pending functionalities of this module have been carried out to complete its development. The *REST API* has been refined to allow a common endpoint that can be used by the rest of the self-* modules to send alerts to the domain's aerOS EAT and the *Facts Generator* has been modified to adapt it to the new data model of the Context Broker's Infrastructure Element entity. In addition, work has been done on the interactions between the different self-* modules and the self-orchestrator, as well as the integration and testing in the domains of the Pilots.	
Self-security	Adapted to work in Kubernetes and non-Kubernetes environments, the three components that compose the module (Log Monitoring, ETL processing and the API) generate cyber-intrusion alerts that are sent to the Trust-Manager. Self-security is able to detect 3 types of network attacks: port scan attack, denial of service attack (DoS) and brute force attack. It is expected to expand the portfolio of detected attacks with specific attacks that can be performed on the services installed in the Pilots. This is the schema of the development carried out in the module:  *Figure 68. Self-security schema*	Gets information from the network card and sends alerts to the Trust-Manager via the ETL module (for real-time alerts) and API (for weekly alerts).
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. This is the current structure of the development carried out in the self-API module:	It will interact with all the rest of self-* capabilities in order to manage their configuration / parameters / data.

	 *Figure 69. Self-API schema*	
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 progress up to M30 has been the complete adaptation of the previously developed component (TRL3, from the ASSIST-IoT project) to the data model and the deployment configuration of the aerOS meta-operating system through configurations according to services and service components, in addition to an API to execute processes in a non-sequential way and modify the default parameters according to the needs of the user and the system. This is the current schema of the development carried out in the module:  *Figure 70. Self-scaling schema*	It interacts mainly with the Kubernetes metrics service and it is possible to interact with its API through the Self-API component.
Self-configuration	The self-configuration component can be used for reactive configuration management of heterogeneous resources within an aerOS deployment. Using the REST interface the administrator/user is able to define a multi-stage configuration structure using abstract concepts, such as resource, functionality, action, and reaction, and provide fallback configurations. Actions represent external events and can trigger predefined reactions that may induce configuration evolution, as well as communication with	This component is one of the few that are be able to operate autonomously, without the need to interact with other aerOS self-* functions. It only has to interact with external resources.

	resources.  *Figure 71. Self-configuration schema* Internally, the configuration is represented via a Directed Acyclic Graph (DAG). Its vertices can be of two kinds: resource and functionality, whereas edges represent the "requires-to-function" relationship. Different labels can be associated with each vertex, allowing categorization/grouping of vertices, without changing the overall graph structure. Taking numerical values – called "weights" – into account, the self-configurator can autonomously decide which fallback configurations should be applied to the system in case of an error in any of the system components. For the self-configurator to function, it must be able to communicate with (external) resources. The communication is done through connectors. Their duty is to perform direct manipulation on the resource and inform the self-configurator of the resource's status.	
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.	In the case of detecting abnormal states, or healing actions to be taken, this will feed the health score of the IE. The self-healing module interacts with the self-API and the Trust-Manager modules.

	◦ *Healing*: Alert messages to exclude sensor from the set of those that provide input to the system. 2. Device Power Alert: ◦ *Scenario*: Power level of the device drops 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-orchestrator, 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).	

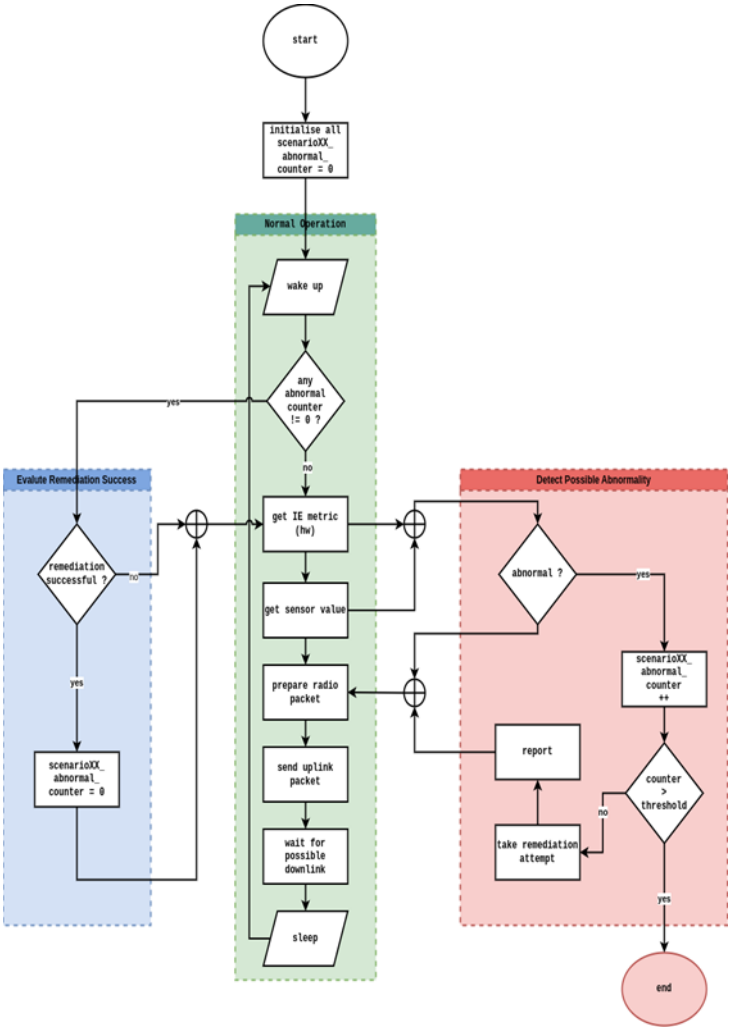


Figure 72. Self-healing schema

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, enhanced versions of all defined scenarios have been completed, and the custom software has been developed and tested in more refined scenarios. The self-healing module has been evaluated in an experimental environment and the test cases of all scenarios have been replicated using DHT22 Digital Humidity & Temperature sensors and Raspberry Pi IEs, demonstrating the improved capabilities and the resilience of the module. Also, the module has been deployed in the aerOS environment and the interactions with self-API and Trust-Manager components have been tested.

The following figures show the network related scenarios in action, within this local experimental environment, with the self-healing module successfully detecting network abnormal states.

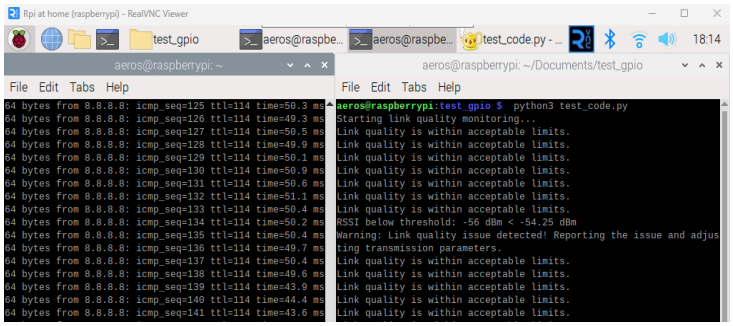


Figure 73. Link Quality Issue scenario

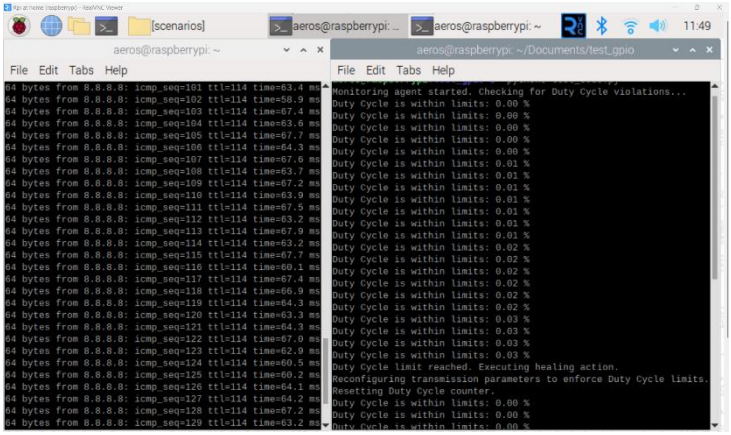


Figure 74. Network Protocol Violation scenario

Also, preparations are in progress for utilizing self-healing in the appropriate aerOS' pilots.

Self-optimisation /adaptation	The goal of this module is two-fold. First, it aims to react in advance to potential scenarios when the IE would like to act upon (e.g., overload, network down, demand peak…). Second, it dynamically adjusts the sampling frequency of the self-awareness to optimize the monitoring and data dissemination. Ultimately, self-optimisation brings the smart/predictive/proactive fashion to the self-* capabilities of an IE in aerOS. From the technical perspective, the module consists of the following components: • Collector/Parser: monitoring the metric streams obtained from the self-awareness service and optionally parsing them to the format acceptable by analytics models. • Sampling Model: model that computes the next optimal sampling period. • Shift/Anomaly Detection Model: model that returns information indicating when the significant change in a metric stream is detected and what type of anomalies it may indicate. • Recommender: component exposing computed information for self-orchestrator and self-awareness. There below is a summary of the flow that this module	Fed with data on the state of the node via the self-awareness module, it sends alerts to the (1) self-orchestrator module about detected anomalies and (2) self-awareness about new optimal sampling period.

follows:

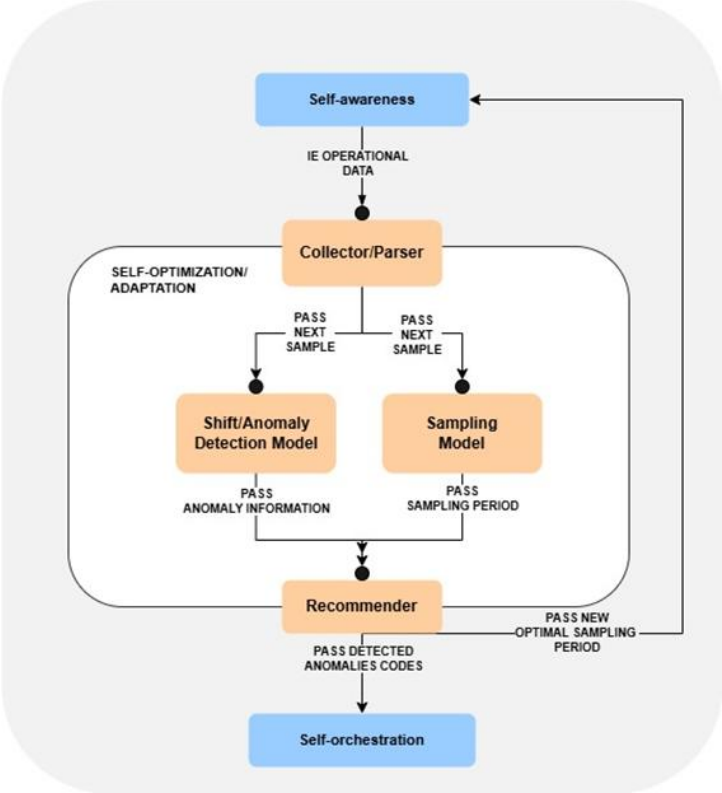


Figure 75. Self-optimisation and adaptation components schema

Its most important components are the Shift/Anomaly Detection Model and Sampling Model.

The Shift/Anomaly Detection Model's internal structure is presented on the following diagram:

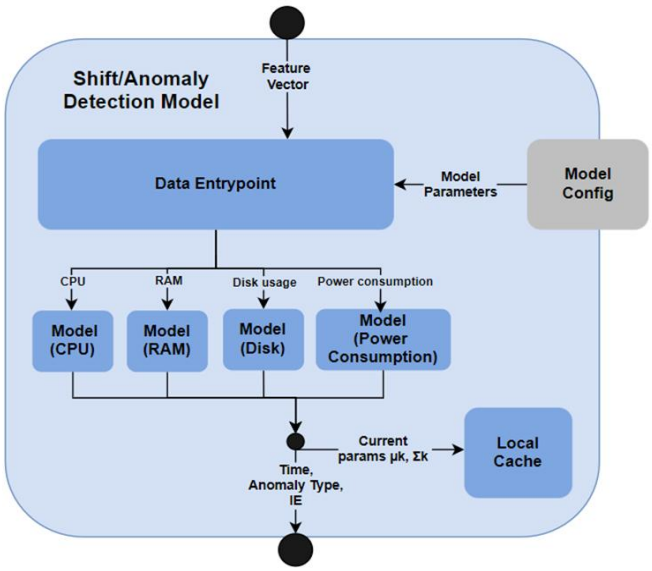


Figure 76. Schema of Anomaly Detection Model

The component separates the detection of anomalies per each type of IE's operational data. Initially, only the types detecting CPU, RAM and Disk usage-related anomalies have been implemented and tested. However, due to the modular

structure, it is simple to extend this component to detect more types of anomalies as well (e.g. power consumption-related). Internally, each model uses a statistical-based (density-based) approach to detect shifts in the metric stream obtained from self-awareness. The sensitivity of the detection for each model type is specified using configuration parameters that can be seamlessly modified by the user on runtime. Before passing the information of anomalies to the self-orchestrator, it is a role of Recommender module to map them into their respective codes recognizable by self-orchestrator.

The next component is the Sampling Model, which internal structure is presented in the following diagram:

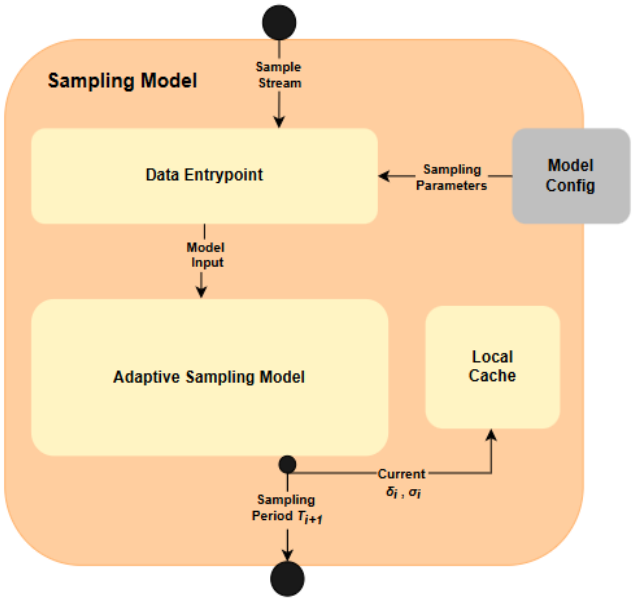


Figure 77. Schema of Sampling Model

This Sampling Model performs all computations in the Adaptive Sampling Model internal component. It accepts the IE's operational data and for each relevant metric (CPU, RAM, Disk usage) computes the optimal sampling period. The computation of sampling period is done by estimating the evolution of the metric stream using the PEWMA calculation. Then, among different proposed sampling periods, the one with the smallest value (i.e. signalling the need of the most frequent monitoring) is selected as the optimal recommendation. Similarly to the Shift/Anomaly Detection Model, the performance of the module is controlled through configuration parameters (e.g. maximal or minimal sampling period) that can also be modified using exposed API endpoints.

Both of the aforementioned components are resource-efficient since they do not require performing complex operations and need to store only individual variables in the local cache (no need for storing historical data).

All of the presented components of self-optimisation and

	adaptation have been implemented, successfully deployed and tested with the integration of the remaining relevant self-* modules. Moreover, internal experimental testing of the performance of individual models was also completed on both real IE operational data and external synthetic data set.	
Self-realtimeness	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 time utility and tardiness of their tasks. **User space component:** Calculates each real-time service's time utility from its tardiness and issues a reorchestration if the tardiness drops below a user-configurable threshold. The self-realtimeness component relies on a patched Linux kernel version (v5.10) that enables hierarchical container-based scheduling (HCBS). We have evaluated the self-realtimeness component in an experimental environment. Figure 43 above shows the real-time performance of a workload by means of its response time (left axis in light blue and light green) and its derived time-utility (right axis in dark blue). The time-utility is at 100 if a soft real-time workload's response time falls within its deadline (red) or degrades (linearly, exponentially, or as a step function) with its tardiness. We can see that the workload running on node 1 exhibits poor response times (light blue) and accordingly a degraded time-utility. As a result, the self-realtimeness component on node 1 issues a re-orchestration request so that the workload is relocated to node 2. We observe an improved response time and time-utility on node 1 (light green). This highlights how the self-realtimeness component effectively detects poor real-time performance of a workload on a node and issues a relocation request resulting in improved real-time performance as a result of a relocation of the affected containerized workload. In addition, we will evaluate the functionality, interoperability, and effectiveness of the self-realtimeness components within the controlled and closed environment of Pilot 3.	**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.

3.5.3. Technologies and standards deployed in MVP

Table 24. Self- capabilities technologies/standards, descriptions and justifications deployed in MVP*

Technology/Standard	Description	Justification
iproute2	Set of utilities for managing network connections and	It allows to obtain the desired information about all physical interfaces of the aerOS

(self-awareness – hardware info)	controlling incoming and outgoing traffic.	nodes. In addition, its small size after installation makes the final Docker image a contained size.
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.
speedtest-cli (self-awareness – hardware info)	Cross-platform library for measuring the upload and download speeds of a node's Internet connection.	It allows measurements to be carried out very simply and efficiently with only a few lines of code.
getmac (self-awareness – both hardware info and power consumption)	Cross-platform library to obtain the MAC address of a node.	It allows to easily obtain the network interfaces of the node and its MAC address with a single line of code.
quart (self-awareness – both hardware info and power consumption)	Cross-platform framework for creating asynchronous web applications. It is an asynchronous reimplementation of Flask.	It allows a REST API to be executed asynchronously in the same thread as the rest of the module's functions. It also allows extensions to be added for more specific needs and functions.
requests (self-awareness – both hardware info and power consumption, self-healing and self-API)	Cross-platform library that allows HTTP requests to be executed.	Allows HTTP requests to be executed in a simple way, with error handling, response codes, headers, data sending, etc.
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.
pandas (self-awareness – power consumption)	Cross-platform library that allows to analyse and manipulate structures and datasets easily and quickly.	Used to analyse the results of *PowerTOP*, it allows the management of possible missing data in the resulting report, the use of column sets to extract information or the analysis of CSV files.
axios (self-orchestrator)	JavaScript library to perform HTTP requests (client side). It can be considered the equivalent of *requests* in Python.	It allows to execute HTTP requests with few lines of code, use Node.js promises, automatically transform JSON data, configure the HTTP request, easy response and error handling, etc.
express (self-orchestrator and self-API)	A flexible minimalist web application suite that provides functions for developing web applications.	It allows to create powerful, lightweight and simple REST APIs. Its small size after installation allows the final Docker image to have a contained size.
fs-extra (self-orchestrator)	JavaScript library that allows extra functionality to be added to the standard *fs* library.	It allows to delete all files in a single directory with a single line of code, allowing to quickly and efficiently complete the functionality of the *DELETE /rules* self-orchestrator endpoint.

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.
jsonschema (self-orchestrator)	Library that allows easy validation of JavaScript objects using JSON schemas.	It allows in the self-orchestrator to create a validator to validate the body of the requests received in JSON format, add sub-schemas, create recursive schemas, determine if there are errors in the JSON received, etc.
Suricata (self-security)	High performance, open-source network analysis and threat detection software.	It has been integrated with Kubernetes to provide real-time security, efficiently processes and analyses data, and enhances network security and incident detection.
FastAPI (self-healing)	Lightweight Python framework for building APIs.	Enables fast and asynchronous API development with automatic validation and OpenAPI support.
getmac (self-healing)	Cross-platform python library to obtain the MAC address of a node.	Provides a simple and efficient way to obtain network interface details of the node and its MAC address with minimal code.
Swagger / OAS (self-API)	Widely adopted framework for designing, building, documenting, testing and consuming RESTful APIs. It allows to define API endpoints, request/response models, and more in a structured and standardised format using a YAML or JSON specification.	Swagger aids in the automation of testing, deployment, and monitoring of APIs. It accelerates development cycles and reduces human error, ensuring that any changes to aerOS self-API are quickly validated and deployed. OAS is also supported by a rich ecosystem of tools.
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.

4. Conclusions

The WP3 in aerOS project has made significant advances in developing a unified, scalable, and secure distributed computing infrastructure that seamlessly integrates IoT, edge, and cloud resources. The journey from the initial Minimum Viable Product (MVPv1) to the final implementation (MVPv2) has been marked by continuous refinements in networking, orchestration, cybersecurity, and self-monitoring.

Key Outcomes and Advancements in WP3:

1. From MVP to MVPv2 – A Refined Execution Environment
    a. The MVP, delivered at M18, provided the foundation for a Meta-Operating System (Meta-OS), ensuring interoperability across diverse computing environments.
    b. By M30, the MVPv2 introduced smarter networking, AI-driven orchestration, and stronger security, making aerOS a robust and industry-ready platform.
2. Smart Networking & Communication Services
    a. The final implementation refined network programmability, enhancing cross-domain connectivity with technologies like WireGuard, ONOS SDN, and OpenCAPIF.
    b. Standardized APIs (OpenAPI, AsyncAPI) ensured easy integration with telecom and IoT ecosystems, enabling interoperability across diverse platforms.
3. AI-Driven Orchestration & Automated Resource Management
    a. The orchestration engine introduced energy-aware workload selection, federated orchestration, and real-time monitoring, optimizing resource efficiency.
    b. AI-powered self-* capabilities (self-awareness, self-healing) reduced manual intervention and improved system resilience.
4. Cybersecurity & Access Control Enhancements
    a. Stronger IAM, RBAC, and secure API gateways ensured controlled access to aerOS resources.
    b. Security mechanisms were tested and validated, making aerOS WP3 components reliable for deployment in real-world scenarios.
5. Deployment in Pilots
    a. The aerOS framework components of WP3 are now ready for large-scale deployment, with real-world pilots validating its efficiency, adaptability, and security.
    b. Continuous work will focus on integrating AI-driven orchestration, and further refining security protocols showcasing in different pilots as part of WP5.

The final implementation of aerOS in WP3 transforms the initial MVP into a fully operational, federated computing platform in MVPv2 capable of dynamically managing networking, orchestration, security, and monitoring across distributed IoT-edge-cloud environments.

With AI-driven automation, secure APIs, intelligent networking, and self-adaptive capabilities, aerOS emerges as a scalable, efficient, and deployment-ready solution for IoT, smart cities, industrial automation, and cloud computing.

Having completed final testing and validation of components developed in WP3, it enables WP5 for deploying them in aerOS pilots, driving digital transformation across various industries.