

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



D3.1 – Initial distributed compute infrastructure specification and implementation

Deliverable No.	D3.1	Due Date	31-AUG-2023
Type	Other	Dissemination Level	Public
Version	1.0	WP	WP3
Description	Initial software components, relationships, building blocks in relationship with the architecture		



Copyright

Copyright © 2022 the aerOS Consortium. All rights reserved.

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 (<i>third linked party</i>)	AT
SIEMENS AKTIENGESELLSCHAFT	DE
FIWARE FOUNDATION EV	DE
TELEFONICA INVESTIGACION Y DESARROLLO SA	ES
COSMOTE KINITES TILEPIKOINONIES AE	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

This document contains material, which is the copyright of certain aerOS consortium parties, and may not be reproduced or copied without permission. This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

The information contained in this document is the proprietary confidential information of the aerOS Consortium (including the Commission Services) and may not be disclosed except in accordance with the Consortium Agreement. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the Project Consortium as a whole nor a certain party of the Consortium warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, and accepts no liability for loss or damage suffered by any person using this information.

The information in this document is subject to change without notice.

The content of this report reflects only the authors' view. The Directorate-General for Communications Networks, Content and Technology, Resources and Support, Administration and Finance (DG-CONNECT) is not responsible for any use that may be made of the information it contains.

Authors

Name	Partner	e-mail
Ignacio Lacalle	P01 UPV	iglaub@upv.es
Raúl San Julián	P01 UPV	rausanga@upv.es
Dr. Harilaos Koumaras	P02 NCSR	koumaras@iit.demokritos.gr
Vasilis Pitsilis	P02 NCSR	vpitsilis@iit.demokritos.gr
Anastasios Gogos	P02 NCSR	angogos@iit.demokritos.gr
Constantinos Vasilakis	P02 NCSR	cvassilakis@iit.demokritos.gr
Giorgos Makropoulos	P02 NCSR	gmakropoulos@ii.demokritos.gr
Thanos Papakyriakou	P02 NCSR	thpap@iit.demokritos.gr
Anna Ryabokon, ,	P04 TTC, P04.1 TCAG	anna.ryabokon@tttech.com
Edin Arnautovic	P04 TTC, P04.1 TCAG	edin.arnautovic@tttech.com
Jan Ruh	P04 TTC, P04.1 TCAG	jan.ruh@tttech.com
Renzo Bazan	P05 Siemens	renzo.bazan.ext@siemens.com
Florian Gramß	P05 Siemens	florian.gramss@siemens.com
Jose Eduardo Fontalvo Hernandez	P05 Siemens	jose-eduardo.fontalvo-hernandez@siemens.com
Philippe Buschmann	P05 Siemens	philippe.buschmann@siemens.com
Korbinian Pfab	P05 Siemens	korbinian.pfab@siemens.com
Ignacio Dominguez	P07 TID	ignacio.dominguezmartinez@telefonica.com
Ioannis Makropodis	P10 IQB	giannis.makropodis@inqbit.io
Vasiliki Maria Sampazioti	P10 IQB	vasiliki.maria.sampazioti@inqbit.io
Konstantinos Kefalas	P10 IQB	konstantinos.kefalas@inqbit.io
Panagiotis Bountakas	P10 IQB	panagiotis.bpountakas@inqbit.io
Tarik Taleb	P14 ICTFI	tarik.taleb@ictficial.com
Tarik Zakaria Benmerar	P14 ICTFI	tarik.benmerar@ictficial.com
Amine Taleb	P14 ICTFI	amine.taleb@ictficial.com
Nikolaos Gkatzios	P15 INF	ngkatzios@infolysis.gr
Vaios Koumaras	P15 INF	vkoumaras@infolysis.gr
Aggeliki Papaioannou	P15 INF	apapaioannou@infolysis.gr
Michele Mondelli	P19 DST	m.mondelli@dstech.it
Jon Egaña	P20 S21SEC	jegana@s21sec.com

History

Date	Version	Change
23-MAY-2023	0.1	ToC prepared for partner contributions
5-JUL-2023	0.2	Version with first content and ToC updated after Plenary Meeting
21-JUL-2023	0.4	Version with 4 sections completed – sent to IR
28-JUL-2023	0.5	Internal Review over most sections is received and tackled
7-AUG-2023	0.9	Version submitted to PC review
15-AUG-2023	1.0	Final Review by Editor
15-AUG-2023	1.0	Official final version of the document is ready

Key Data

Keywords	Decentralized orchestration, smart networking, security, edge-cloud continuum, self-*, Monitoring, Common API, and Identity and Access Managements.
Lead Editor	P10 ICT-FI – Tarik Taleb
Internal Reviewer(s)	P09 8BELLS – Angelos Constantinides, P17 ECTL Petros Dias

Executive Summary

The document is contextualized to the works in aerOS' WP3: *aerOS secure, scalable and decentralized compute infrastructure*. The present deliverable is the first and initial version of three WP3 deliverables planned for M12, M18 and M30. The deliverable is based on the assumptions and boundaries defined in D2.1 (State-of-the-Art and market analysis report) and D2.2 (Use cases manual, requirements, legal and regulatory analysis 1), and consistent with the architectural choices in D2.6 (aerOS architecture definition 1) which result from the following tasks:

- T2.1 state of the art.
- T2.2 use cases and requirements.
- T2.5 architecture.

This deliverable is the first materialised result of WP3 activities detailing the relevant components of the architecture of the aerOS decentralised infrastructure composed of 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.

The document is then structured in a way to provide for each task, the methodological and technological choices that are specified in the context of the aerOS decentralized infrastructure.

From smart-networking perspective, the aerOS project will focus on K8s-based VPN as well as intra-domain services mesh to achieve smart inter-domain networking. Cilium and Ligo are foreseen to be the technological focus on this part.

In the communication services and API part, multi-protocol tools based on Fiware IoT agents will be used to provide bridging capabilities with NGSI-LD context brokers. To achieve efficient communication, DDS and Zenoh will be the technological pillars to implement non-blocking, predictable, and timed execution of operations.

Multi-domain smart services and resources orchestration systems are hard to implement. Multi-level Orchestration approach will be used to achieve a scalable and solid orchestration system. From the technological perspectives, Kopf will be the one of the key enablers for the operator development representing a relevant instantiation of the low-level orchestrator. For inter-services communication, Kafka will be the potential open source tool for the Events Bus. The machine-learning operations in the high-level orchestration will be based on key open-source tools, namely: Kubeflow, MLFlow and Alibi/Seldon, respectively.

From cybersecurity perspective, KrakenD -which is an API Gateway- will allow the decoupling between the cybersecurity-related IAM (Identity and Access Management) and backend-operations. For the IAM operations in aerOS, Keycloak will be the base technology. To provide identity management, OpenID will be the main technological enabler.

To enable nodes autonomous operations, the open source tool Prometheus will be used for self-awareness, json-rules-engine for self-orchestration and KubeEdge for self-healing. Resulting tools from European projects such as ASSIST-IoT and RAINBOW EU will be harnessed for nodes autonomy in aerOS.

With each part distilled with its own components, software implementation of aerOS was already initiated. Further refinements will be described in the upcoming deliverables in D3.2 planned for M18 and D3.3 planned for M30, resulting from activities across all WP3 tasks.

Table of contents

Table of contents	6
List of tables	7
List of figures	7
List of acronyms	8
1. About this document.....	10
1.1. Deliverable context	10
1.2. The rationale behind the structure.....	11
1.3. Outcomes of the deliverable.....	11
2. Introduction	12
3. Preliminary proposal of software solutions.....	13
3.1. Smart networking for Infrastructure Element connectivity.....	13
3.1.1. Research lines and structure diagrams	14
3.1.2. Related requirements	23
3.1.3. Candidate technologies and standards	24
3.2. Communication services and APIs	25
3.2.1. Research lines	25
3.2.2. Related requirements	26
3.2.3. Structure diagram.....	27
3.2.4. Candidate technologies and standards	28
3.3. aerOS service and resource orchestration	29
3.3.1. Research lines and structure diagrams	29
3.3.2. Related requirements	45
3.3.3. Candidate technologies and standards	46
3.4. Cybersecurity components.....	47
3.4.1. Research lines	47
3.4.2. Related requirements	49
3.4.3. Structure diagram.....	50
3.4.4. Candidate technologies and standards	50
3.5. Node’s self and monitoring tools	51
3.5.1. Research lines	51
3.5.2. Related requirements	52
3.5.3. Structure diagram.....	52
3.5.4. Candidate technologies and standards	57
4. Conclusions and future work.....	58
References	60

List of tables

Table 1. Candidate technologies smart networking in aerOS.....	24
Table 2. Communication services and APIs Components description.....	27
Table 3. Candidate technologies for Communication and Service APIs.....	28
Table 4. Decentralized High-Level Orchestration components	35
Table 5. Low-Level Orchestration Components in the case of K8s framework	38
Table 6. aerOS resource orchestration components	45
Table 7. Candidate technologies and standards for Resource Orchestration.....	46
Table 8.: Identity and Access Management and Secure API Gateway components	50
Table 9: Identity and Access Management and Secure API Gateway candidate technologies	50
Table 10. Self-capabilities components, description and interactions.....	54
Table 11. Self-features and monitoring candidate technologies.....	57

List of figures

Figure 1. WP3 components in the aerOS stack	12
Figure 2. Flat Layer-3 Pods Networking.....	15
Figure 3. Network service mesh in K8s environment	16
Figure 4. Network service endpoints.....	16
Figure 5. Network service mesh.....	17
Figure 6. Control-Data plane.....	18
Figure 7: Cross domain network for service mesh in NSM	18
Figure 8. VPN Concentrator functioning for inter-domain communication by NSM.....	19
Figure 9. Cross domain network for service mesh by NSM.....	19
Figure 10. Network service mesh using Istio for layers 4 to 7.....	20
Figure 11: SDN controller in a NSM scenario	21
Figure 12. Joint usage of SDN and NSM to provide QoE services in a K8s cluster.....	21
Figure 13. Cloud native solution for NFV: CNCFV	22
Figure 14. Example of interworking of aerOS with TSN infrastructure	23
Figure 15. Time aware communication.....	26
Figure 16. Communication services and APIs architecture.....	27
Figure 17. Multi-level Orchestration approach as a source of inspiration to aerOS orchestrator [1].....	31
Figure 18. aerOS Federative architecture	32
Figure 19. Aeros Services Orchestration.....	33
Figure 20. Decentralized High-Level Orchestration	34
Figure 21. Detailed AI-powered Decision-Making Process.....	36
Figure 22: aerOS services Low-Level Orchestration in the case of K8s framework	37
Figure 23. Operators are custom controllers watching a custom resource [2]	39
Figure 24. KNF VNFD Example [12].....	41
Figure 25. NSD Example [12].....	41
Figure 26. Primitive actions implementation in a Helm Charts-based execution environment [14].....	42
Figure 27. Helm-chart based primitives inside the VNFD [13]	42
Figure 28. High-Level/Low-Level Orchestration Integration	43
Figure 29. aerOS resource orchestration architecture.....	44
Figure 30. RBAC representation	48
Figure 31: RBAC elements	49
Figure 32: API Gateway architecture	50
Figure 33. Interaction diagram of self-features in IEs of aerOS.....	53
Figure 34. Inner structure of self-orchestration feature in an IE	55
Figure 35. Inner schema of abnormal state detection in self-healing	56

List of acronyms

Acronym	Explanation
API	Application Programming Interface
BLE	Bluetooth Low Energy
CBAC	Context-based access control
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CRD	Custom Resource Definition
DDS	Data Distribution Service
DevOps	Development and Operations
DevPrivSecOps	Development, Privacy, Security and Operations
ETSI	European Telecommunications Standards Institute
FaaS	Function-as-a-Service
FOM	Federated Orchestration Module
HLO	High Level Orchestrator
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IdM	Identity Management
IE	Infrastructure Element
IoT	Internet of Things
LLO	Low Level Orchestrator
K8s	Kubernetes
KDU	Kubernetes Deployment Unit
KNF	Kubernetes-based Virtual Network Function
MANO	Management and Orchestration
ML	Machine Learning
MQTT	MQ Telemetry Transport
NFV	Network Function Virtualization
NGSI-LD	Next Generation Service Interface – Lined Data
NS	Network Service
NSD	Network Service Descriptor
NSM	Network Service Manager
OSM	Open Source MANO
QoE	Quality of Experience
QoS	Quality of Service

RBAC	Role-based access control
SDN	Software Defined Network
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

The present document describes the deliverable D3.1 which provides the initial formal methodological specification and technological implementation of the components building up the aerOS decentralised infrastructure, which conforms an essential part of the Meta-OS. It builds up on the architectural choices in WP2 and describes the composing components and their interactions in detail.

Two other deliverables in M18 and M30 will follow to refine the choices and adapt to other related work packages activities, most notable WP2 and the architectural choices. This document is the initial blueprint for building up the aerOS infrastructure and its components integrated in aerOS use cases as detailed in WP5.

1.1. Deliverable context

Item	Description
Objectives	<p>O1 (Design, implementation and validation of aerOS for optimal orchestration): Methodological specification and technological implementation of the orchestration components.</p> <p>O2 (Intelligent realisation of smart network functions for aerOS): Methodological specification and technological implementation of the smart-networking components.</p> <p>O3 (Definition and implementation of decentralised security, privacy and trust): Methodological specification and technological implementation of the methodological choices of the cybersecurity components.</p> <p>O5 (Specification and implementation of a Data Autonomy strategy for the IoT edge-cloud continuum): Specification and implementation of the NGSi-LD integration with communication services and API.</p>
Work plan	<p>D3.1 content is based on the final assumptions and boundaries defined by the tasks:</p> <ul style="list-style-type: none"> T2.1 state of the art. Approaches and solutions defined in this deliverable are based on the studied state of the art. T2.2 use cases and requirements. Solutions defined in these deliverable answers the requirements for the different use cases. T2.5 architecture. Components defined in this deliverable are defined within the defined aerOS architecture. <p>D3.1 content is the result of the following tasks activities:</p> <ul style="list-style-type: none"> 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. <p>The integration of components defined by WP4 tasks is detailed within the decentralized infrastructure context in this D3.1 deliverable.</p> <p>D3.1 components technological choices are the basis of the WP5 integration and use case deployments tasks.</p>
Milestones	<p>This deliverable is not directly related to any milestone but constitutes an important basis for the milestone <i>MS3 – Components defined</i> planned for M12 and more importantly for the milestone <i>MS7 – Final software release</i> planned for M30.</p>
Deliverables	<p>This deliverable content is based on the assumptions and boundaries defined in D2.1 (State-of-the-Art and market analysis report) and D2.2 (Use cases manual, requirements, legal and</p>

	regulatory analysis 1). It is consistent with the architectural choices defined in D2.6 (aerOS architecture definition 1).
--	--

1.2. The rationale behind the structure

The present deliverable provides the functional components for the five areas or tasks composing the WP3 activities. It consists thus of five sections. Before delving into the main outcomes, the document provides an outline of its content followed by a brief introduction. Each of the five task-specific sections is presented in the same formal structure. It begins with an overview of the main functionalities. It then explains the relation of the section and the specific requirements/use case scenarios as specified in D2.2. Structure diagrams and description of each component in these diagrams are following and finally the candidate technologies for building the solution are quoted. The last part of the document concludes the deliverable and provides perspectives about the future work envisioned for the next iteration D3.2.

1.3. Outcomes of the deliverable

The main goal of this deliverable is to provide the initial distilling of the aerOS infrastructure components. Components extraction will be approached from both methodological and technological standpoints, covering five specific areas that directly align with the various tasks outlined in WP3.

The smart-networking area provides the functional components needed to achieve networking efficiency, agility and performance across the aerOS infrastructure elements.

The communication services and API area provides the functional components needed for effortless, efficient and adaptable communication between aerOS services across the compute continuum.

Service and resource orchestration, on one hand, furnishes the functional elements necessary for deploying, managing, and federating services that deliver aerOS functionality. On the other hand, it supplies the functional components needed to appropriately allocate and utilize diverse resources in a balanced manner to fulfil the requirements of vertical IoT services deployed on top of aerOS.

The security components provide Identity and Access Management (IAM) services which are responsible for the process of identifying and authenticating different types of entities (individuals, groups of people or software processes). The objective is to control access to computerized resources (applications, systems or networks) by associating entity rights and restrictions with existing and established identities.

Node's self and monitoring tools provide the functional components to augment Infrastructure Elements (IEs), as nodes, with autonomy related features, which will provide the capability to minimize, to a large extent, human intervention during all their operations. To achieve such an objective, health state and other functional and runtime parameters' monitoring is crucial.

2. Introduction

WP3 together with WP4 constitute the two technical work packages of the aerOS project. The WP3 activities provide the required infrastructure components, based on the aerOS architecture, needed to enable scalable and secure IoT edge-cloud continuum. These components will support the resources and services orchestration for autonomous system operations. To achieve this end, the main objectives are:

- To propose the architectural approach and technologies for multi-domain smart networking capabilities.
- To provide the domain communication services and API frontends for smart-data aggregations.
- To propose the required architecture and tools to ensure interoperable Identity and Access Management (IAM) services with other aerOS components.
- To provide the methodology and solutions to orchestrate services and resources over the distributed infrastructure.
- To propose the tools for the functional components of the node’s self-mechanisms and monitoring operations.

WP3 will complement and build upon the outcomes of WP4 that will enable delivering intelligence at the edge of the aerOS infrastructure, by optimizing usage of data within aerOS without sacrificing control, trust and privacy over it.

Furthermore, the present document (D3.1) focuses on defining the initial software components, relationships and building blocks aligned with the aerOS architecture. aerOS will be developed following a Meta-OS approach, encompassing and addressing the project objectives. From the point of view of the aerOS architecture, the system will be designed around several main building blocks and fundamental concepts. For that reason, the deliverable, D2.6, offers a comprehensive overview of aerOS, consolidating the main concepts and building blocks. Additionally, it analyses the core elements, services offered, and functionalities that will be offered. D3.1 was prepared in parallel to D2.6 and synchronised with its content.

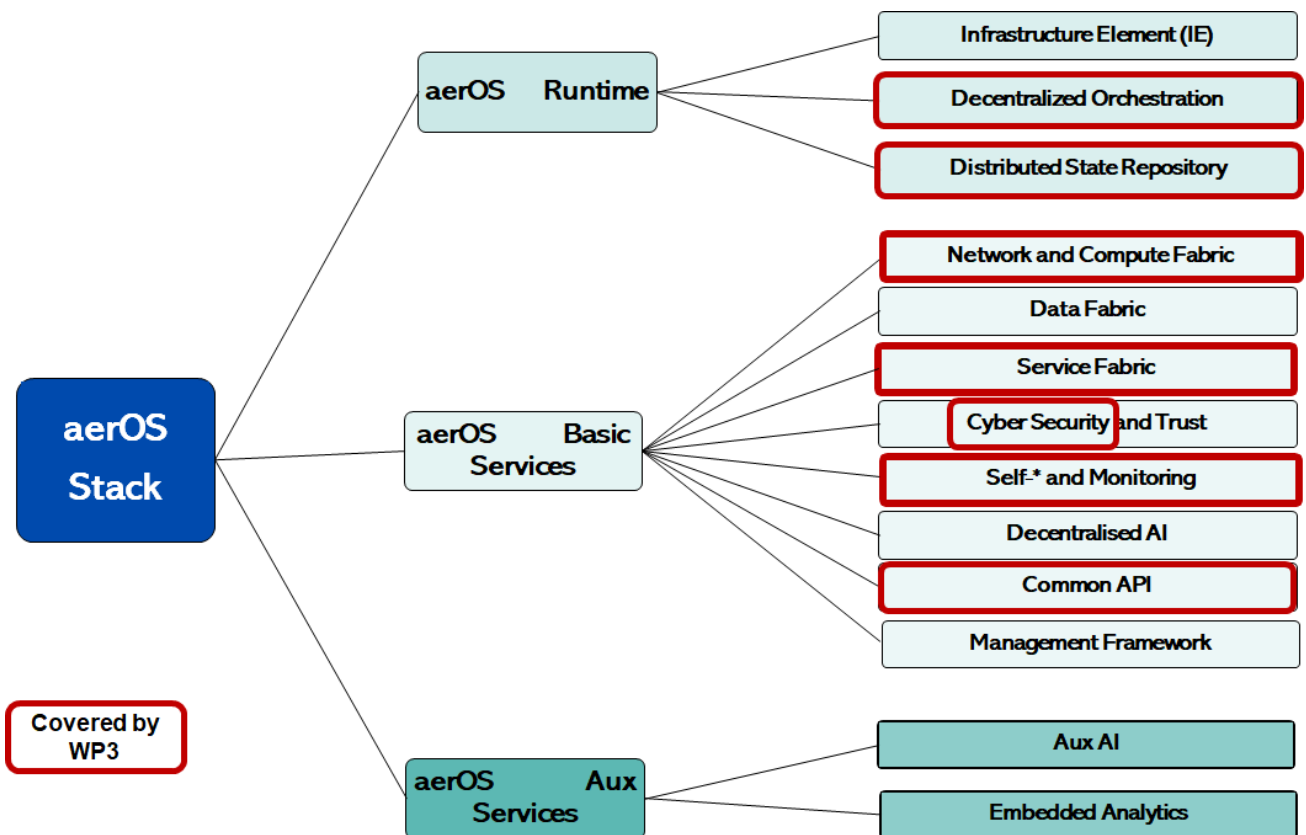


Figure 1. WP3 components in the aerOS stack

D2.6 outlines several principles that aerOS builds upon, such as the Infrastructure Element (IE) and the aerOS domain. On the one hand, the IE, as the most granular computing entity, serves as the fundamental building block. It enables the deployment and management of workloads through a flexible and adaptable physical or virtual entity, supporting containerised workloads, providing network connectivity, storage capacity, and a well-defined API to expose its state. On the other hand, the aerOS domain is formed by one or more IEs, creating a complete aerOS domain that facilitates the hosting and sharing of essential aerOS basic services across all IEs. In addition, D2.6 defines an architectural stack that provides a structured overview of runtime, basic, and auxiliary aerOS services. Figure 1 explicitly shows the location on this aerOS stack of the WP3 proposed solutions described in D3.1.

In particular, as it can be seen, D3.1 is crucial as it sets the ground for the development of the decentralised orchestration and the distributed state repository (T3.3). Also, it is pivotal for the aerOS basic services, as WP3 will output the network and compute fabric (T3.1), the service fabric and common APIs for aerOS (T3.2). Finally, it also covers (partially) the cyber-security and trust elements (T3.4) and implements the self-* and monitoring tools (T3.5).

3. Preliminary proposal of software solutions

This section contains the specification of the preliminary proposal of solutions for WP3, along with the identification of their respective technical components. Each subsection corresponds to a task within WP3 and presents the identified solutions related to it. These subsections are structured in the same way i.e., it includes: main functionalities and general description, relation to requirements and/or use case scenarios, high-level structure diagram of its components, a table with the components description, and candidate technologies for implementation. Some subsections are further divided into specific parts to provide more granular details.

3.1. Smart networking for Infrastructure Element connectivity

Smart networking refers to the use of intelligent technologies and strategies in networking infrastructure to enhance efficiency, agility, and performance. It involves leveraging advanced networking technologies, such as software-defined networking (SDN), network virtualization, automation, and analytics, to optimize network operations and deliver reliable connectivity.

In the context of Cloud, Infrastructure and equipment deployed through different regions, smart networking would involve deploying intelligent networking infrastructure, utilising cloud-based networking solutions, and strategically placing equipment in various regions to create a cohesive and efficient network architecture. Key goals of smart networking include:

- enabling seamless communication
- improving scalability
- enhancing security
- optimizing resource utilization

By leveraging smart networking principles, organisations can achieve greater control, flexibility, and visibility over their network infrastructure, enabling them to adapt quickly to changing business needs and efficiently manage their distributed network assets. Smart networking provides several key functionalities that enhance network operations, performance, and management. Here are some of the main functionalities:

- **Automation:** Smart networking leverages automation to streamline and simplify network operations. It involves automating repetitive tasks, such as device provisioning, configuration changes, and network monitoring. Automation reduces human errors, speeds up deployment, and enables more efficient use of network resources.
- **Unified Management:** Smart networking allows for unified management of network infrastructure. Through a single control plane or management interface, administrators can configure, monitor, and control network devices and services across distributed locations. Unified management improves visibility, simplifies troubleshooting, and enables consistent policy enforcement.
- **Intelligent Routing and Traffic Optimization:** Smart networking utilises intelligent routing algorithms to optimise the flow of network traffic. It dynamically selects the most efficient paths based on

real-time network conditions, application requirements, and performance metrics. Intelligent routing improves network performance, reduces latency, and ensures optimal resource utilisation.

- **Security and Threat Intelligence:** Smart networking incorporates advanced security features to protect network infrastructure and data. It includes capabilities like network segmentation, access controls, encryption, and intrusion detection/prevention systems. Additionally, it leverages threat intelligence and analytics to identify and mitigate potential security threats in real-time.
- **Scalability and Flexibility:** Smart networking enables scalability and flexibility to accommodate changing business needs and network demands. It allows for easy expansion of network capacity, seamless integration of new devices or services, and the ability to adapt to evolving technologies and protocols. Scalable and flexible networking ensures that the network can grow and evolve along with the organization's requirements.
- **Analytics and Insights:** Smart networking leverages analytics and network monitoring tools to gather data and extract actionable insights. By analysing network performance metrics, traffic patterns, and user behaviour, it helps optimise network resources, identify bottlenecks, predict capacity needs, and make informed decisions for network improvements.
- **Quality of Service (QoS) and Prioritisation:** Smart networking implements QoS mechanisms to prioritize critical applications and ensure a consistent user experience. It allows administrators to allocate bandwidth, set latency thresholds, and prioritise specific types of traffic (e.g., voice, video, real-time applications) to meet performance requirements and service-level agreements (SLAs).

These functionalities collectively contribute to a more efficient, resilient, and intelligent network infrastructure. Smart networking empowers organizations to optimize their networks for performance, security, and agility, enabling them to meet the evolving demands of modern digital environments.

This sub-section covers the research and developments done so far in Task T3.1. The challenge lies on implementing a smart, dynamic and auto-configurable network infrastructure, in which all IEs communicate seamlessly, ensuring low latency and resilient exchanges both intra- and inter-domain. It constitutes the basis for delivering aerOS' network and compute fabric.

Within this task, the efforts have been structured to advance in the following 7 research lines:

1. Smart networking within the Kubernetes¹ context
2. Intra-domain network service mesh
3. Inter-domain network service mesh
4. Network Service Mesh and Service Mesh
5. Network Service Mesh and SDN
6. Networks Service Mesh and NFV
7. TSN Support for the aerOS continuum

In the next pages, the global outcome of the task is described, and the results so far, out of the 7 research lines, are afterwards depicted. Furthermore, a list of candidate technologies that have been pre-selected is provided.

3.1.1. Research lines and structure diagrams

Before digging deep into the various research lines and diagrams, it is worthwhile to contextualise the work conducted. It has been defined in aerOS that K8s will be the most usual environment where the workload execution will take place in the Meta-OS. While the project does not constrain itself to only investigate K8s-related stuff (as, logically, other orchestration environments will exist), the research in the network field in the action is ultimately biased by the innovations in the cloud-native fields. This does not preclude that, either in T3.1 or in other tasks, further research will be conducted outside the cloud-native assumptions.

¹ Interchangeably referred to as K8s across this document.

3.1.1.1. Smart Networking within the Kubernetes context

Kubernetes (K8s) has emerged as the *de-facto* standard for cloud-native application orchestration (that is, the deployment, scaling, and administration of containerized apps), with practically all public and private clouds offering managed K8s services. K8s is an open-source container orchestration platform that enables the deployment, scaling, and management of containerised applications across distributed environments. It provides features like service discovery, load balancing, and automated container placement to ensure efficient utilisation of resources.

As the majority of applications deployed in K8s clusters are built on microservices architecture, there is a significant volume of east-west traffic among these services. To satisfy the network demands of these east-west traffics, K8s employs a flat layer 3 network architecture. Within the K8s ecosystem, there are networking components and concepts that facilitate communication between containers and enable connectivity across a cluster. These components include:

- **Pod Networking:** K8s assigns a unique IP address to each pod, allowing containers within the pod to communicate with each other over a virtual network. Various pod networking solutions, like Calico, Flannel, or Cilium, provide network connectivity and routing capabilities.
- **Service Networking:** K8s abstracts individual pods with a higher-level construct called “service.” Services provide a stable IP and DNS name, allowing other pods or external services to connect to them. K8s implements load balancing for services, distributing traffic across multiple pods to achieve scalability and high availability.
- **Ingress and Load Balancing:** K8s offers an Ingress resource that provides external access to services within the cluster. It acts as a smart load balancer, routing traffic to appropriate backend services based on rules and policies.

Different implementations of the model are possible, but all must fulfil the following fundamental requirements:

- Each pod has its own IP address.
- Each pod can connect directly with any other pod in the same cluster without the use of NAT (Network Address Translation).

If the minor intricacies of bridging and routing settings are ignored, the K8s network may be represented in a simplified way as follows:

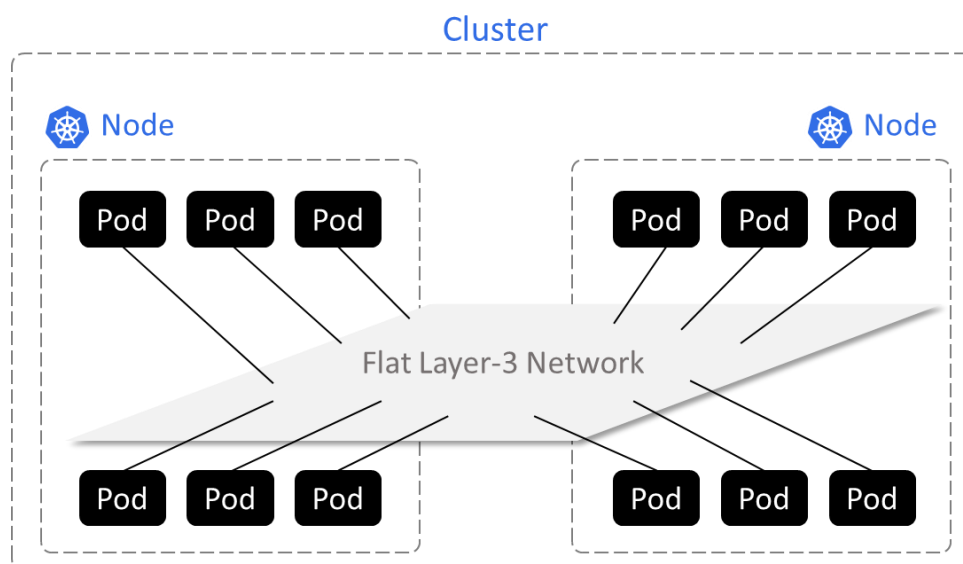


Figure 2. Flat Layer-3 Pods Networking

As shown in Figure 2. Flat Layer-3 Pods Networking, all pods in a K8s cluster may communicate with one another through a flat layer 3 network. The term "flat" refers to the fact that a pod may visit any other pod in the same cluster using just layer 3 routing and no NAT in the middle, which means that the source and destination pods view the same IP addresses in the packages transferred between them.

Of course, the layer 3 network is only "flat" from the standpoint of a pod. In practice, this L3 network might be implemented as an overlay network with complicated encapsulation.

3.1.1.2. Intra-domain Network Service Mesh

Network Service Mesh (NSM) is a CNCF project that provides sophisticated L2 / L3 networking capabilities for K8s-based applications. NSM does not interact with the K8s CNI; instead, it is a completely independent mechanism comprised of a variety of components that may be deployed in or out of a K8s cluster. It is a cloud-native network solution that operates across many clouds and hybrid clouds. When compared to the K8s service, Network Service will be easier to grasp. A K8s service may be thought of as an abstract construct that provides some type of application layer (L7) service to clients, such as HTTP or GRPC services. Network Service is defined similarly in NSM, however instead of L7, a Network service offers L2 / L3 service. The following are the distinctions between Service and Network Service (Figure 3):

- Service: It is application workload that offers services at the application layer (L7), such as web services.
- Network Service: It is a network function that delivers services at the L2 / L3 layer, which means it processes and sends packets but does not often terminate them. Examples of Network Services includes Bridge, Router, Firewall, DPI, VPN Gateway, etc.

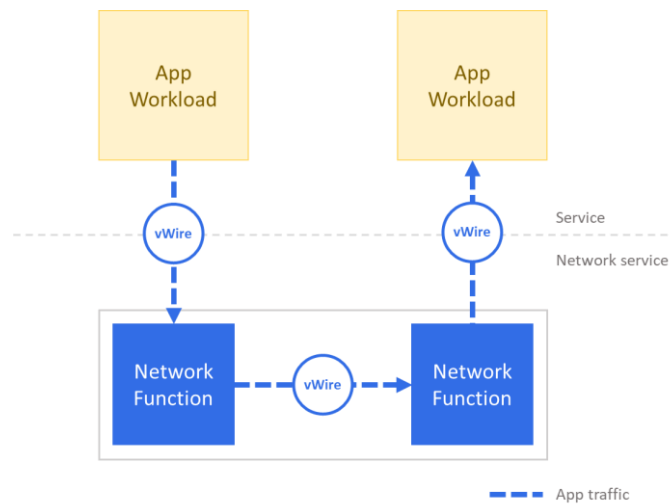


Figure 3. Network service mesh in K8s environment

To deliver service in K8s, numerous endpoints may be present behind the scenes. It is also true for NSM Network Service, where several pods/endpoints are deployed to share client loads and may be horizontally scaled to meet varying demands (Figure 4).

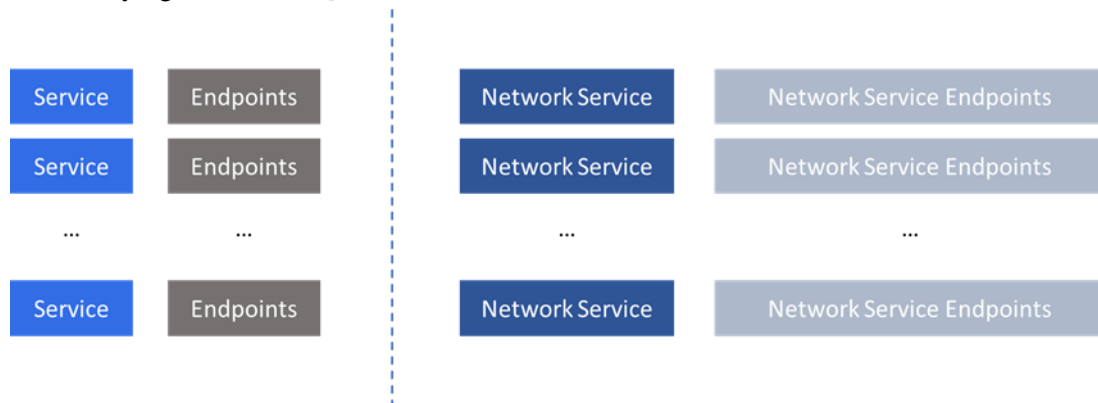


Figure 4. Network service endpoints

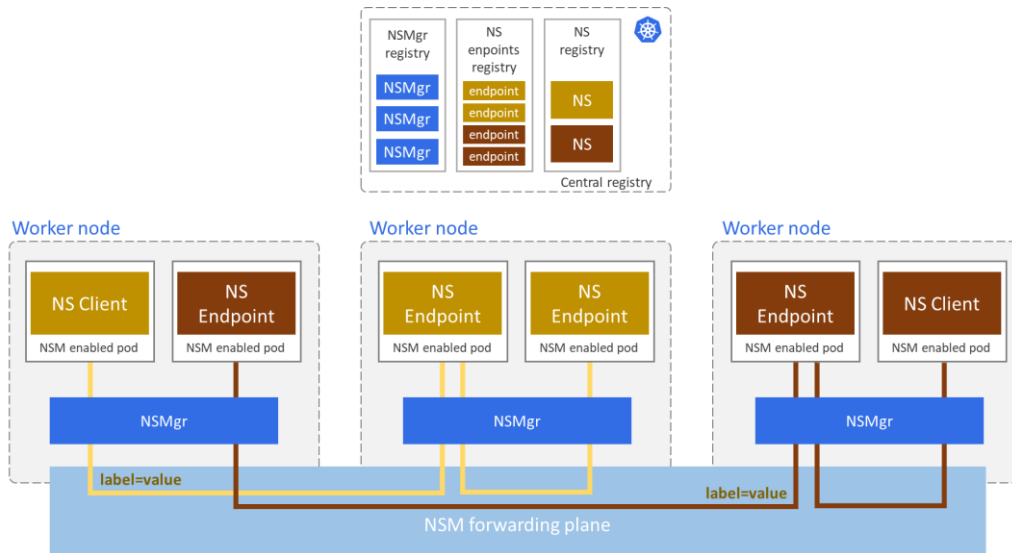


Figure 5. Network service mesh

As indicated in the diagram in Figure 5, Network Service Mesh is made up of various parts:

- Network Service Endpoint (NSE): a container, pod, virtual machine, or physical forwarder that implements Network Services. A network service endpoint receives connection requests from clients that wish to use the Network Service that the endpoint provides.
- A Network Service Client (NSC) is a person who requests or uses a Network Service.
- The network service registry (MSR) is the repository for NSM components such as NS, NSE, and NSMgr.
- Network Service Manager (NSMgr): It is the NSM's control plane. It is installed as a daemon on each node.

To establish a distributed control plane, NSMgr connects with one another. NSMgr is principally in charge of two tasks:

- It receives Network Service requests from the NSC and matches the requests with the appropriate NSE before establishing a virtual connection between the NSC and NSE (the data plane component handles the actual work).
- It registers the NSE on the NSR's node.

Network Service Mesh Forwarder: A network service's end-to-end connections, wires, mechanisms, and forwarding components are provided by this data plane component. This can be done either directly by setting up provisioning mechanisms and forwarding elements, or indirectly by sending requests to a middle control plane that can supply the four components required to implement the network service. FD.io (VPP), OvS, Kernel Networking, SRIOV, etc. are a few examples.

NSM deploys an NSMgr on each Node in the cluster. These NSMgrs talk to each other to select appropriate NSE to meet the Network Service requests from clients, and create a virtual wire between the client and the NSE. Accordingly, these NSMgrs form a mesh to provide L2/L3 network services for the applications, similar to a Service Mesh (Figure 6).

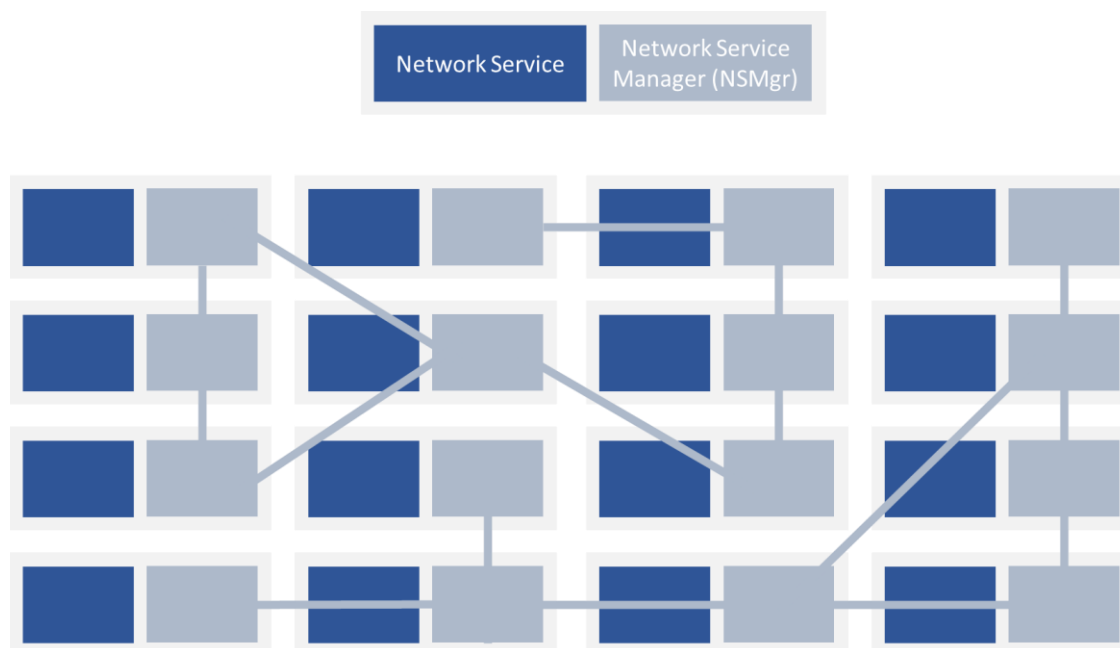


Figure 6. Control-Data plane

3.1.1.3. Inter-domain Network Service Mesh

Consider connecting a pod that is located to another domain, which, in order to access it, a VPN connection becomes required, as shown in Figure 7.

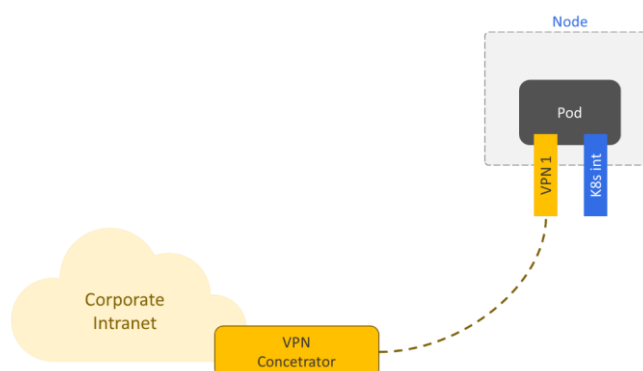


Figure 7: Cross domain network for service mesh in NSM

In order to do so, some sort of Virtual Private Network (VPN) is needed. In the conventional method, there is the need for manually configuring some network details, such as the VPN gateway address, the subnet prefix and IP address, the routes to the corporate intranet, etc., which should not be exposed to the user of the VPN Gateway service at all. This VPN gateway must be installed somewhere the pod can reach, most likely in the same cluster. In this case, the client just has to connect to the corporate intranet and performs its duties; it is not necessary for it to be concerned with the implementation's technical specifics, such as how the VPN is set up and configured.

NSM, on the other hand, provides a straightforward declarative approach to offer the VPN service to the clients. The VPN network service, network service endpoint, and network service client definitions are displayed in Figure 8.

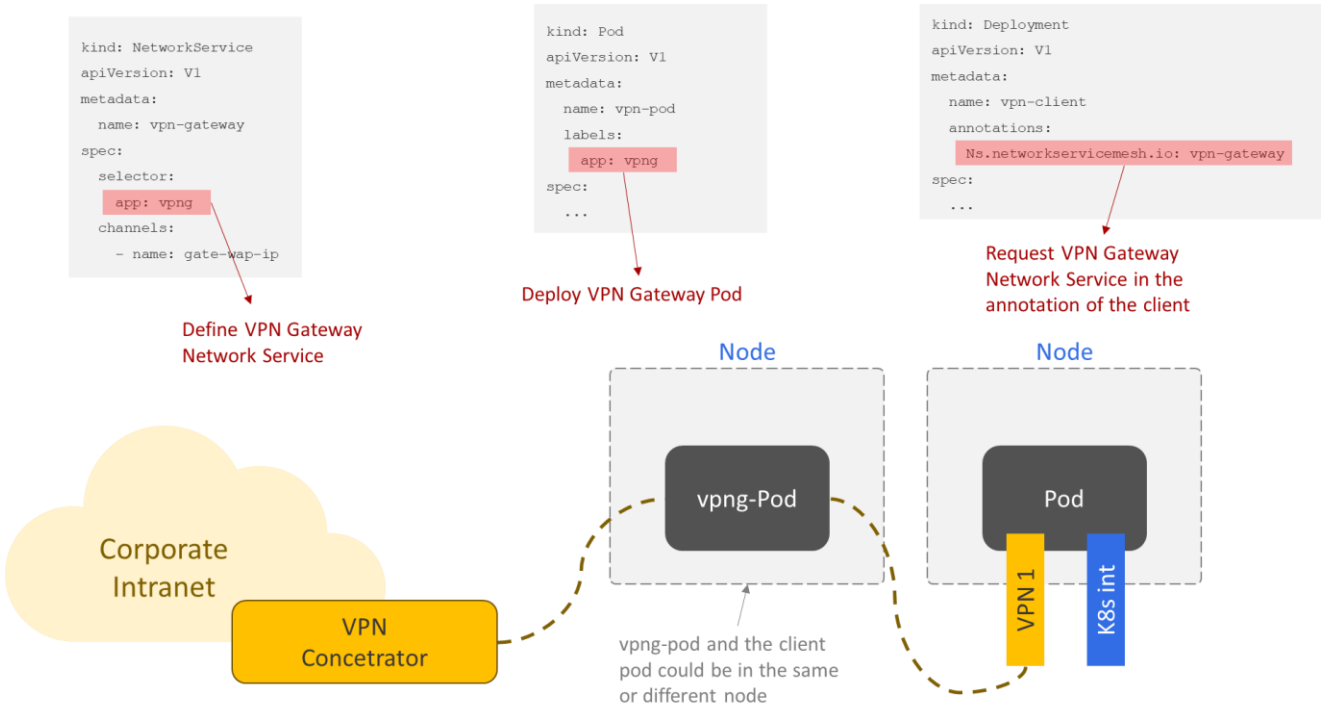


Figure 8. VPN Concentrator functioning for inter-domain communication by NSM

- Using the NetworkService CRD, the vpn-gateway Network Service can be defined. The yaml specification states that vpn-gateway NS takes IP payload and utilises a selector to match backend pods with the label "app: vpng" as the Network Service providers.
- To request the Network Service, the client uses the annotation "Ns.networkservicemesh.io:vpn-gateway".

In K8s, NSM has an admission webhook set up that injects an *init* container into the client pod. By communicating with the NSMgr in the same node, this *init* container asks the required Network Service indicated in the annotation. After the Network Service has been configured by NSM, the application container is launched in a way that is transparent to the client.

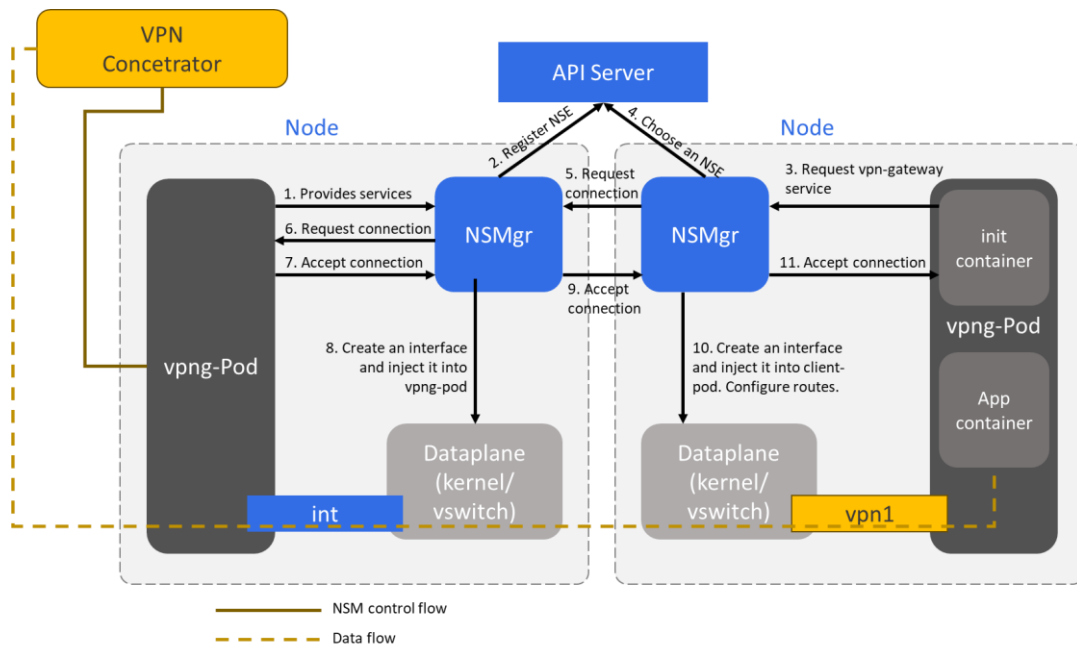


Figure 9. Cross domain network for service mesh by NSM

Figure 9 illustrates the following steps:

1. To provide VPN gateway network service, vpng-pod has been implemented.
2. The API Server's (Service Registry) API Server registers vpng-pod as an NSE.
3. The NSM init container in the client pod contacts the NSMgr on the same node to request the vpn-gateway network service.
4. To find accessible network service endpoints, NSMgr requests the API Server (Service Registry).
5. The selected NSE may be located on the same node or another. The NSMgr contacts its peer on the distant node to transmit the request if it is there.
6. On behalf of the NSC, the NSMgr on the NSE node makes a connection request.
7. If the NSE still has the resources to fulfil the request, it accepts it.
8. The network interface is created and injected into the NSE's Pod by the NSMgr on the NSE node.
9. The NSMgr on the NSE node tells the NSMgr on the NSC node that the service request has been submitted if the NSE and NSC are on separate nodes.
10. The network interface is created and injected into the NSE's Pod by the NSMgr on the NSE node.
11. The network interface is created and injected into the NSC's pod by the NSMgr on the NSC node, which also configures the routes to the corporate network.

3.1.1.4. Network Service Mesh and Service Mesh

NSM uses the Service Mesh idea. However, it operates on distinct OSI model levels. Service Mesh handles service-to-service communication (service discovery, LB, retries, circuit breakers, sophisticated routing with application layer headers), as well as offering security and insight for microservices, at Layers 4 and 7 (mainly Layer 7). The layer 2 and layer 3 network services offered by Network Service Mesh include virtual L2 and L3 networks, VPNs, firewalls, and DPI, among others. Service Mesh and Network Service Mesh can genuinely cooperate if necessary. For instance, using NSM, an overlay L3 network spanning many clouds could be established, on top of which an Istio Service Mesh could be constructed (Figure 10).

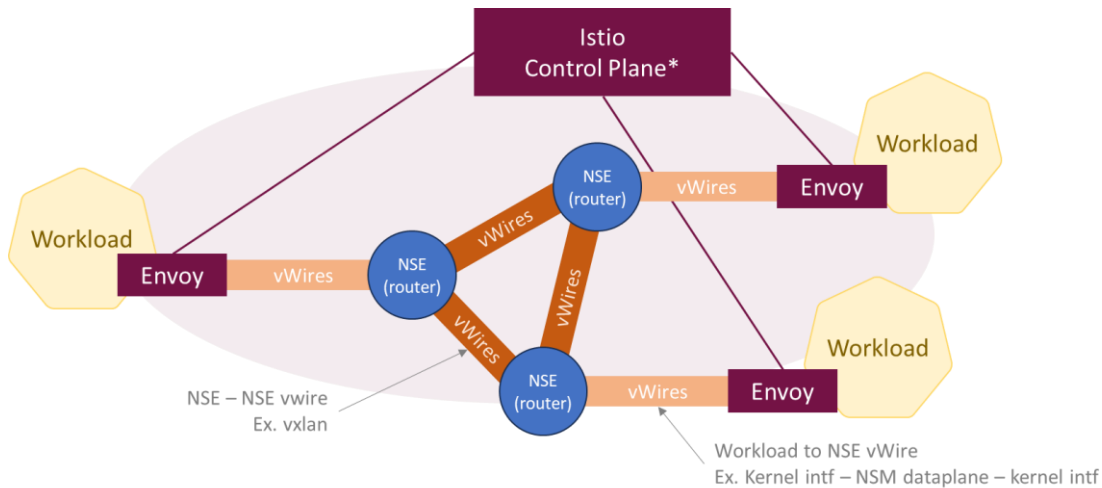


Figure 10. Network service mesh using Istio for layers 4 to 7.

3.1.1.5. Network Service Mesh and SDN

Figure 11 illustrates how SDN (Software Defined Networking) disaggregates network control and forwarding tasks from individual switches and routers and instead places them in a centralised SDN controller:

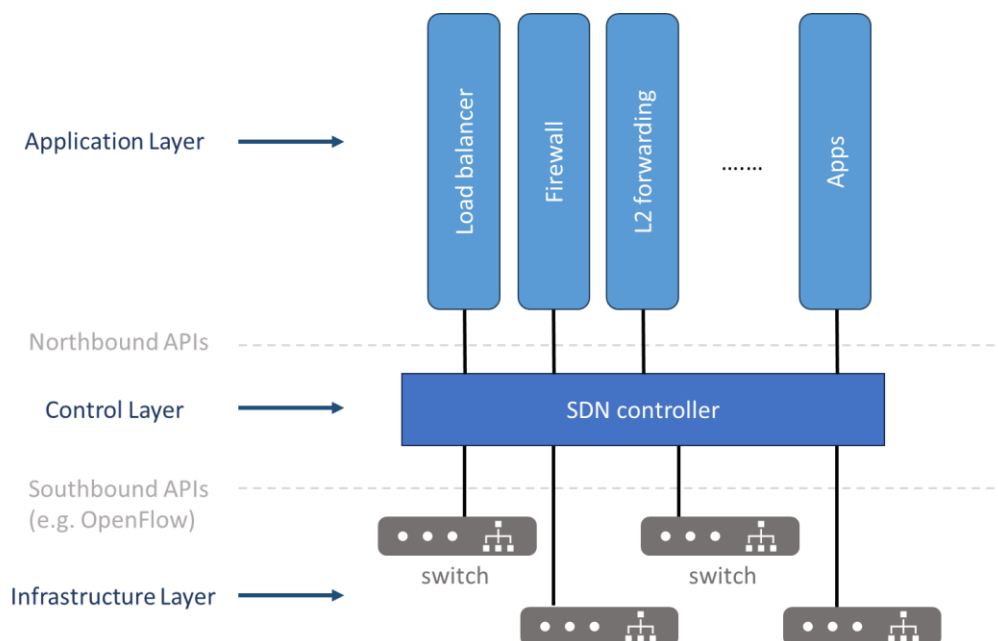


Figure 11: SDN controller in a NSM scenario

There are OSI layer overlaps between NSM and SDN. Both SDN and NSM operate on L1/L2/L3, although in separate regions. In contrast to SDN, which is primarily intended to make network equipment configuration and maintenance easier, NSM aims to offer sophisticated L2 and L3 network services in a cloud-native manner for K8s. NSM can wrap the capabilities of SDN into Network Services to be used by pods in K8s. Figure 12 shows an example of using SDN together with NSM to provide QoE (Quality of Experience) services in a K8s cluster. Network Services that are utilized by K8s pods can be wrapped in NSM to incorporate SDN features. An illustration of how SDN and NSM may be used to deliver QoE (Quality of Experience) services in a K8s cluster is shown in Figure 12. In this illustration, the SDN controller configures the network devices and implements the real QoE mechanism in the transport network while NSM offers QoE network services and the virtual wire between the client and QoE network service endpoint in K8s.

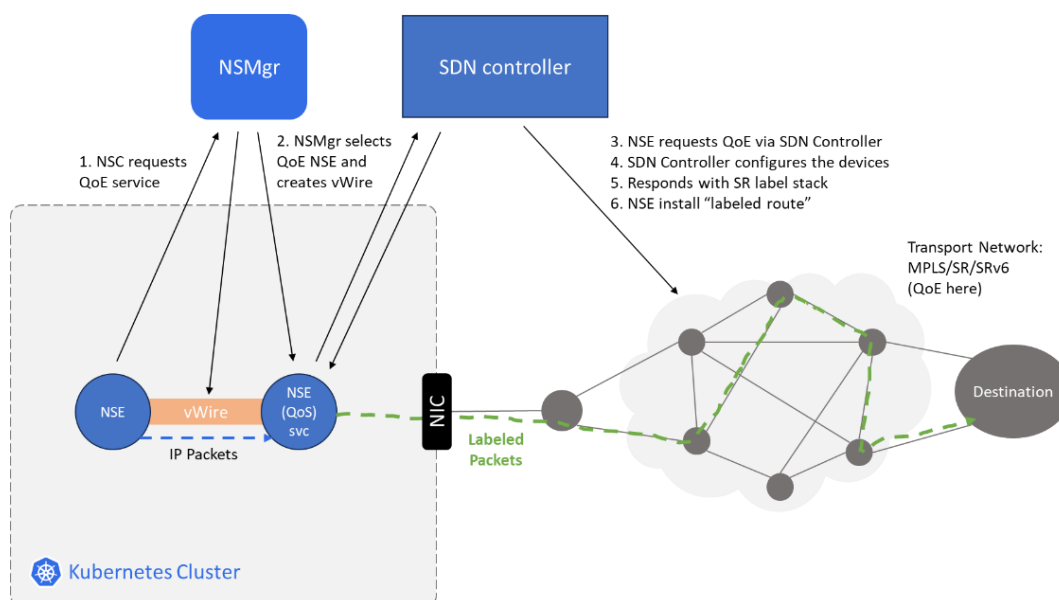


Figure 12. Joint usage of SDN and NSM to provide QoE services in a K8s cluster

3.1.1.6. Network Service Mesh and NFV

The decoupling of network operations from proprietary hardware appliances and executing them as software is known as network functions virtualization (NFV). Virtual machines (VMs) are typically used to bundle these virtualized network functions (VNFs). However, containers may use far fewer resources and be far more effective than VMs. A VM may spin up in minutes as opposed to a container's few seconds. The main issue with this method is that K8s, the container orchestrator, lacks the networking skills required for NFV. The development of NSM filled in the puzzle piece that was lacking and offers a cloud-native NFV solution. VNFs might be implemented as NSM Network Services, and by connecting these Network Services, service function chains (SFC) can be created.



Figure 13. Cloud native solution for NFV: CNCfV

Currently, telecommunications standards (such as the ETSI NFV family) are driving NFV. Telecommunication standards are important for ensuring interoperability across many manufacturers or operators, however the lengthy procedure required to create a standard makes it extremely inefficient. In the near future, NFV may experience dramatic developments because of open-source initiatives like NSM.

3.1.1.7. TSN support for the aerOS continuum

Time-Sensitive Networking (TSN) is a set of IEEE standards (e.g., IEEE 802.1Qbv, 802.1Qci, 802.1Qcc) that aim to introduce deterministic, low-latency communication over Ethernet networks. TSN is mainly used in critical real-time applications where precise timing and low latency are essential. The primary goal of TSN is to enable time synchronization and traffic scheduling mechanisms to ensure that time-critical data is delivered promptly and reliably. This is achieved through techniques such as time-aware shaper (TAS) and time synchronization protocols like IEEE 1588 (PTP).

The aerOS continuum will have the possibility to manage applications and services that require TSN functionality. However, despite being both networking technologies, there are no standard integration mechanisms or direct interworking between Network Service Mesh and Time-Sensitive Networking. NSM primarily focuses on container orchestration platforms like K8s, while TSN targets real-time communication over Ethernet networks. This calls for a novel design for an application architecture that can leverage the two technologies in parallel, intelligently using several network interfaces according to the needs of the incoming or outgoing traffic flows.

Developing an application architecture capable of harnessing the advantages of both Network Service Mesh (NSM) and Time-Sensitive Networking (TSN) is crucial in addressing the user plane connectivity challenge between aerOS and TSN. By integrating NSM and TSN, the application can achieve seamless communication and interoperability between the continuum and the devices connected to the TSN network, effectively extending the reach of that continuum. However, ensuring efficient control plane operation also becomes a pressing concern. To tackle this, a new mechanism must be devised, empowering aerOS with the capability to gain deep insights into the TSN infrastructure and effectively control and manage the network. This mechanism will empower aerOS to orchestrate its own TSN-capable services efficiently, optimizing performance, and delivering a seamless user experience across diverse interconnected environments. In this early stage of design, that mechanism is referred to as the aerOS TSN Auxiliary Service.

Figure 14 shows the potential interactions of the TSN-capable aerOS applications and the TSN Auxiliary Service in the context of an example of integrated network. In that example, the user application exchanges information via TSN with a robot arm and a camera. The information of the capabilities of the TSN network and the ways to establish TSN flows are provided by the Centralized Network Controller (CNC) to aerOS via its TSN auxiliary service.

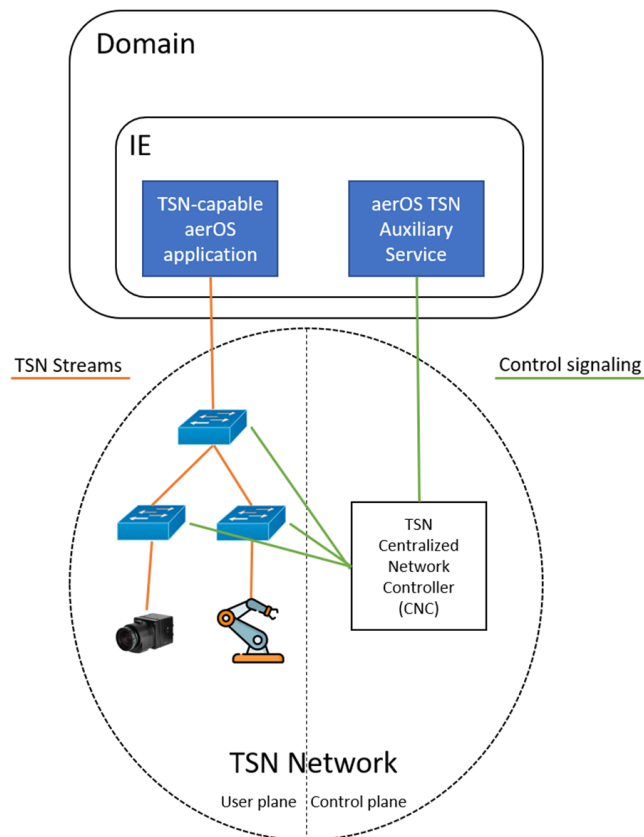


Figure 14. Example of interworking of aerOS with TSN infrastructure

3.1.2. Related requirements

Smart networking capabilities, described in this section, hold a prominent role regarding the Meta-OS deployment and operation. They address networking requirements and programmability as expressed in D2.2 and are relative to the core functionality of aerOS. Networking implementation will be supporting all pilots and scenarios and some features will be evaluated in specific cases. The list of related technical requirements related to smart networking are quoted below.

- TR-9 Network programmability in the IoT edge-cloud continuum
- TR-20 Resource availability
- TR-26 aerOS Infrastructure Monitoring
- TR-27 Infrastructure management automation
- TR-29 Services visibility across virtual network links
- TR-32 aerOS network monitoring
- TR-54 Cybersecurity tools
- R-P1-3 Low latency communication between edge devices and with cloud
- R-P1-4 Secure communications between edge devices and with the cloud
- R-P3-2 Low latency communication between system components
- R-P4-1 Develops aerOS IE that integrates data telemetry from cranes into aerOS Data continuum
- R-P4-4 Integration of IPTV camera streams in aerOS
- R-P5-4 Automatic service recovery upon system or network loss

- R-P5-8 Scalability to Support Mass Deployments

3.1.3. Candidate technologies and standards

Table 1. Candidate technologies smart networking in aerOS

Technology/Standard	Description	Justification
Cilium	Cilium, developed by Isovalent, is a technology that plays a significant role in enabling smart networking, particularly in the context of containerized environments. Cilium focuses on providing efficient and secure networking for microservices, containers, and K8s clusters.	<ul style="list-style-type: none"> • Cilium as a Data Plane Technology: Cilium acts as a powerful data plane technology that enhances network connectivity, security, and observability within containerized environments. It leverages eBPF (extended Berkeley Packet Filter) to provide high-performance networking and security capabilities at the kernel level, allowing for efficient packet processing and enforcement of fine-grained policies. • Network Security and Encryption: Cilium incorporates advanced security features, such as network-layer encryption and microsegmentation, to ensure secure communication between microservices and containers. It enables encryption at the network level, adding an additional layer of protection to data in transit. Microsegmentation allows for granular access controls and isolation of network traffic, enhancing security and mitigating lateral movement risks. • Service Mesh Integration: Cilium can seamlessly integrate with service mesh frameworks like Istio, enabling advanced traffic management, load balancing, and observability capabilities. It enhances service mesh functionality by providing efficient and scalable networking components, reducing latency and improving performance within the service mesh environment. • Observability and Network Analytics: Cilium offers comprehensive observability features, including network flow visibility, service-level observability, and distributed tracing. It collects rich network-level data and exposes it to monitoring and analytics tools, allowing for deep insights into network performance, troubleshooting, and security analysis. • Integration with K8s: Cilium is designed to work seamlessly with K8s, providing enhanced networking capabilities for containerized workloads. It integrates directly into the K8s cluster and leverages K8s APIs for efficient configuration and control. Cilium integrates with K8s network policies to enforce fine-grained network access controls.
Liqo	Liqo is an open-source project that facilitates the establishment of a decentralized cloud infrastructure by enabling the interconnection and	<ul style="list-style-type: none"> • Dynamic Network Interconnection: Liqo allows K8s clusters to establish secure, dynamic, and seamless network connectivity between different clusters. It enables the creation of virtual network overlays that span across clusters, enabling pods and services to communicate with each other as if they were within the same cluster. This dynamic interconnection enhances the flexibility and scalability of network resources and supports efficient workload migration and resource sharing across clusters. • Intelligent Traffic Routing and Load Balancing: Liqo incorporates intelligent traffic routing and load balancing mechanisms to optimize network traffic between interconnected clusters. It can intelligently route traffic based on factors such as latency, bandwidth availability, and cluster resource utilization. This capability ensures

	<p>resource sharing between different K8s clusters. In AerOS, there is no consensus yet on its adoption. The intention is to explore the capabilities of LIQO and investigate its suitability to meet the requirements of AerOS in terms of smart networking.</p>	<p>efficient utilization of network resources and enhances the performance and responsiveness of distributed applications running across federated clusters.</p> <ul style="list-style-type: none"> • Cross-Cluster Service Discovery and Access: Liqo provides mechanisms for cross-cluster service discovery and access. It allows services deployed in one cluster to be discovered and accessed by applications running in other clusters. This capability simplifies multi-cluster application deployments and enables seamless communication between services across different clusters, contributing to smart networking in distributed environments. • Network Security and Policy Enforcement: Liqo incorporates security mechanisms to ensure secure communication and policy enforcement between interconnected clusters. It supports network segmentation and access controls, allowing administrators to define and enforce fine-grained network policies across federated clusters. This enhances security and compliance within the interconnected infrastructure. • Scalability and Resilience: Liqo enables the federated infrastructure to scale horizontally by integrating resources from different clusters. It supports automated workload placement and scheduling across clusters based on resource availability and application requirements. This scalability ensures efficient resource utilisation and enhances overall system resilience.
--	---	--

3.2. Communication services and APIs

This sub-section covers the research and developments done so far in Task T3.2. The challenge lies on implementing the functionalities of seamlessly registering, communicating (APIs) and conveying the interaction between services in the continuum.

The primary objective of this task is to investigate the communication environment that facilitates the communication between services across the entire compute continuum, spanning from edge to cloud. This exploration is centred on communication services designed to enhance the connectivity of aerOS services, ensuring both efficiency and adaptability. Our solutions are structured to empower users to establish services with protocol choices tailored to their unique needs and requirements, while maintaining compatibility and performance with essential aerOS services or aerOS verticals. Furthermore, this task necessitates an examination of standardized API specifications, which serve a dual purpose: providing a technical description of the interface for developers and functioning as a foundation for code generators.

3.2.1. Research lines

In the context of communication services and APIs in aerOS, the research lines are: (i) integration services for protocol translation, (ii) service-oriented communication with timing guarantees and (iii) API specifications for documentation and starting point for code generators.

3.2.1.1. Integration services for protocol translation

A key aspect of aerOS proposal is the provision for users to select from a variety of protocols, aligning with the specific needs of their submitted services. To facilitate communication across these diverse protocols, the use of integration services capable of translating one protocol into another is proposed. Notably, both eProsima Integration Service and FIWARE IoT agents can act as a bridge between protocols such as HTTPS, DDS, MQTT, OPC UA, and other widely used industrial protocols. Specifically, FIWARE IoT agents enable communication with NGS-LD context brokers, a well-established mechanism across the aerOS continuum.

It is important to note that integration services may not deliver optimum efficiency for real-time applications. In such instances, users might require native communication without the interference of integration services. Integration services would then provide an opportunity to intercept this native communication, thereby enabling the use of other services in different protocols for data gathering and analysis. This results in a seamless edge-cloud continuum, irrespective of the protocols to be utilised.

3.2.1.2. Example for service-oriented communication with timing guarantees

As mentioned earlier, users may require native communications for efficient communication between applications. In such instances, the working team in T3.2 has identified DDS and Zenoh as viable options for implementing non-blocking, predictable, and timed execution of operations. Zenoh and DDS are two of such protocols that support data-centric communications, have support for heterogeneous networking technologies (TCP/IP, BLE, 3G, 6LowPan), and facilitate pushing data to subscribers and storage. This communication approach allows for the efficient offloading of computationally-intensive tasks to the cloud during runtime, optimizing system performance without compromising execution speed or system responsiveness. In addition, these communication protocols would also support time-aware communication, implying that applications can operate within a precisely timed environment, even when performing compute-intensive tasks in the cloud during runtime (Figure 15).

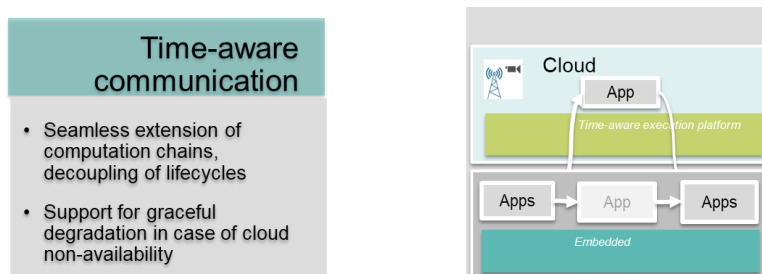


Figure 15. Time aware communication

Should the cloud-based application or component be unresponsive or unavailable, the communication platform and protocol are designed to seamlessly invoke the local component (with potential degraded functionality) operating on the embedded aerOS node (for example, an ECU). It is crucial to establish a platform that mirrors the same environment on both embedded systems and the cloud.

3.2.1.3. API specifications for documentation and starting point for code generators

A critical aspect of API development is the creation of an API specification, which serves as a comprehensive technical description of the interface and primarily acts as a reference for developers. This description must include all interface endpoints (i.e., URLs, ports, channels, topics), accessible operations and their data formats. OpenAPI already establishes a uniform, standardized format for the NGS-LD API, which is a RESTful API. In parallel to OpenAPI, AsyncAPI provides another standardized API specification for asynchronous communication. Already in use for MQTT, this standard could be extended to other protocols, thereby fostering a transparent and well-documented overview of the aerOS systems and providing developers with a convenient starting point for creating APIs. Finally, the employment of standardized API protocols, in combination with code generators, paves the way for low-code environments, such as Behavior Trees, to effortlessly define the logical flow between services. More research will be performed in this matter during the next months.

3.2.2. Related requirements

- TR-11 Data autonomy
- TR-13 Distributed data management
- TR-31 (semi) real-time data analysis

- TR-46 Data cataloguing
- TR-47 Data collection
- TR-51 Distributed data management
- TR-52 Data integration
- TR-53 Data-as-a-product

3.2.3. Structure diagram

Figure 16 illustrates the structure diagram of the Communication and Service APIs, including the relevant components that will be considered.

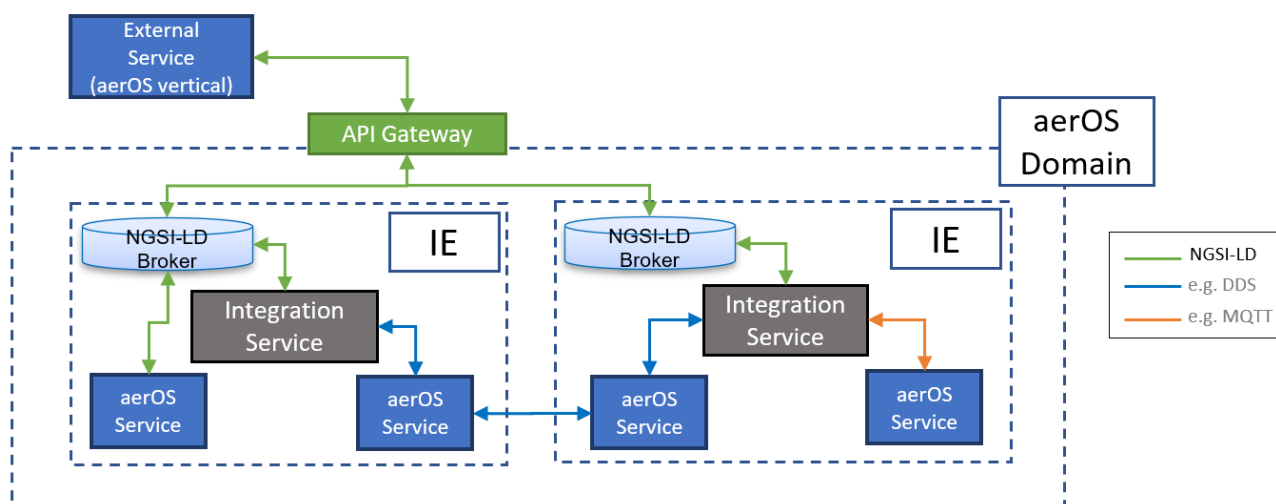


Figure 16. Communication services and APIs architecture

Such components are identified and further described in Table 2.

Table 2. Communication services and APIs Components description

Component	Description	Interactions
aerOS Service	A self-contained unit of software designed to perform a specific functionality within the aerOS domain.	Can interact with integration services, brokers, or other services, contingent on the protocol utilized.
External Service	A self-contained unit of software designed to perform a specific functionality by a third-party provider.	Can interact with aerOS services through an API Gateway.
Integration Service	An integration service is a specialised type of service that facilitates communication and data exchange between different systems or components that might use diverse data formats or communication protocols.	Functioning as a bridge for services, brokers, and gateways, translating messages between protocols.
NGSi-LD Broker	An NGSi-LD Broker is a context broker that manages context information according to the NGSi-LD standard, facilitating interoperability and the integration of various services.	Can interact with components that utilise the NGSi-LD data format
API Gateway	A server that acts as an intermediary for requests from external users who are seeking resources from aerOS services.	Can interact with external services and brokers.

3.2.4. Candidate technologies and standards

Table 3. Candidate technologies for Communication and Service APIs

Tech/Standard	Description	Justification
FIWARE IoT Agents	FIWARE IoT Agents are components enabling the integration of IoT devices with a FIWARE-based system. They provide a bridge between IoT protocols and the NGSI-LD context data interfaces.	Given their ability to translate between numerous protocols, FIWARE IoT Agents are ideal for ensuring smooth and seamless communication within the aerOS ecosystem. They effectively interface with NGSI-LD context brokers, a significant component within the aerOS continuum.
eProsima Integration Service	eProsima Integration Service is a tool designed to interconnect different systems using different communication protocols.	eProsima Integration Service can serve as a bridge between protocols like HTTPS, DDS, MQTT, OPC UA, and others. This makes it an excellent choice for facilitating communication between diverse systems, especially in the aerOS environment, which likely incorporates various protocols.
Zenoh	Zenoh is a data/content centric communication protocol, i.e. pub/sub/query protocol that unifies data in motion, data at rest and computations.	Ideal for large-scale distributed systems, Zenoh uses Named Data Networking and Content-Centric Networking paradigms, enabling data to be associated with one or more resources identified by a URL. It supports routing and is implemented in Rust, making it suitable for extensive, diverse systems.
DDS	Data Distribution Service (DDS™) is a middleware protocol and API standard for data-centric connectivity from the Object Management Group® (OMG®).	Suited for mid-scale distributed systems that require fault tolerance and QoS, DDS shares similar traits with Zenoh. The crucial difference lies in DDS's emphasis on quality of service and fault tolerance, making it appropriate for systems where reliability and continuity are paramount.
OpenAPI	OpenAPI is a specification for machine-readable interface files for describing, producing, consuming, and visualizing RESTful web services.	OpenAPI's ability to provide a standardized, uniform format for RESTful API specifications is invaluable in the aerOS context, as it ensures consistency across different interfaces. It also helps in the creation of detailed API documentation, which is crucial for developers.
AsyncAPI	AsyncAPI is a standard that allows you to define and describe asynchronous APIs in a machine-readable format.	Similar to OpenAPI, AsyncAPI can be used for standardized API specification, but specifically for asynchronous communication.
Behavior Tree	Behavior Trees are a graphical modeling language for the creation of complex, hierarchical behaviors.	Behavior Trees provide a straightforward way of defining complex logic in a visual format. When integrated with code generators, they can simplify the process of defining logical flow between services in the aerOS environment, thereby reducing the coding burden and potential for errors.

3.3. aerOS service and resource orchestration

This sub-section covers the research and developments done so far in Task T3.3. The challenge lies on providing an efficient end-to-end orchestration of resources and services in the IoT-Edge-Cloud continuum, leveraging zero-touch concept and AI techniques. There is also a great challenge for a continuous analysis of the state of the continuum so that a series of services KPIs and user requirements are enforced at any moment.

Within this task, the efforts have been structured to advance in the following two primary lines of action:

- Service orchestration (focused on allocating workloads that implement business logic across the available resources in the continuum – IEs). Here, this line addresses the global structure of the orchestration within aerOS (multi-level orchestration), the mechanisms and templates to communicate the orchestration orders and events, the AI algorithms to apply such advanced orchestration, the global interaction flow (pipeline) and how to deploy all the previous.
- Resource orchestration. This line focuses on the deployment of new resources across the continuum via the tools provided by the Meta-OS. While for the previous line, it is assumed that the resources are available (must be added to the continuum), this research line does not make such assumption, and prepares a methodology and tools to settle new infrastructure to be ready for aerOS. Although this research line is extremely interesting, it is now being demoted in favour of the first research line, that constitutes the main novelty of aerOS and that will be used during the first half of the project and by the pilots to orbit aerOS validations around. At later stages of the project, this line will be reinforced.

In the next pages, the advances and decisions so far in each of those two lines are described.

3.3.1. Research lines and structure diagrams

3.3.1.1. Services orchestration

Objectives

A key goal of this research line is to describe and inspect how the services of the aerOS continuum should be deployed and managed. Service orchestration is a key enabler for the aerOS project: it should ensure that all components work together seamlessly to provide aerOS functionality. Another main goal is to address the challenge of service federation across domains. A collaborative effort is needed for the establishment of a robust framework to ensure a successful and sustainable federation among domains so that the true realisation of a “continuum”, and not separated silos, is possible. Interoperability is a crucial requirement for service federation. A relevant description of the global strategy of aerOS orchestration can be found in Deliverable D2.6.

Functionalities provided

There will be two different layers of service orchestration. In aerOS, instead of the classic approach of relying on a single element for the allocation decision and the actual deployment of the services, a two layered structure has been envisioned. A first layer will be in charge of analysing the status of the continuum (starting with the same domain) to take an allocation decision (place service in another domain if needed), and the second layer will be enforcing the actual deployment of the service.

The traditional concept of orchestration within aerOS applies when functioning within a single domain, where both orchestration layers could be implemented by the same entity or element within the domain itself. Conversely, when it comes to connecting and aligning services across distinct aerOS domains, the concept of federated orchestration is introduced, which represents a significant advancement compared to the existing state-of-the-art approaches.

For service orchestration in aerOS, an AI Support System is used to implement optimised deployment decisions and make certain tasks in a more efficient way. Service orchestration often involves automating repetitive or complex tasks, reducing manual effort, and improving operational efficiency. By automating the coordination and execution of services, and especially by doing this smartly and consciously over all the available infrastructure, organizations can achieve faster and more reliable outcomes, while freeing up human resources for higher-value activities. aerOS takes this automation to an upper level by envisioning an AI based service

orchestration and configuration.

For federated orchestration, it is needed that each aerOS domain High Level Orchestration (HLO) is aware, any time, of all aerOS domains and IEs availability and capabilities across the whole continuum. This requires that all these capabilities are registered and can be discovered and shared. This way, HLO can address service deployment requests by considering both submitted IoT services deployment request requirements and resources availability across all aerOS domains. The registry and sharing mechanisms provided for resources makes them discoverable and thus available when needed and this can lead to a best match, of services and IEs or domains, regarding IoT services placement.

When federating services across domains, establishing trust and ensuring security are paramount. Trust frameworks, authentication mechanisms, and encryption techniques are essential to enable secure communication and protect sensitive data. Identity and access management protocols, such as OAuth, can be used to control access to federated services and will be provided by the aerOS AAA (see Section 3.4).

Specific characteristics of the solution

AI-powered decision support offers several advantages that can leverage and empower aerOS service orchestration. AI systems can analyse vast amounts of data quickly and extract valuable insights. By using machine learning algorithms, AI can uncover patterns, trends, and correlations that may not be apparent to humans. This data-driven approach enables more informed and accurate decision making, a process that is way much faster than manual decision-making processes. AI algorithms can process and analyse data in real-time or at high speeds, enabling swift decision making in critical scenarios. This speed and efficiency can lead to improved operational efficiency and responsiveness. AI can handle complex decision-making scenarios that involve numerous variables, dependencies, and constraints. It can evaluate multiple factors simultaneously and make decisions based on a holistic view of the situation. This capability is particularly beneficial in aerOS due to the complexity of its ecosystem and the needed of taking interdependent decisions based on the whole system status.

Service orchestration in aerOS is one (if not the) more relevant for achieving the goals of the project. Therefore, the required research and activities are deep and many. During the first year of the project, the following aspects have been the focus on the execution of this task. They conform a heterogeneous set of topics ranging from literature analysis, novel diagram design, decisions across the task and the project, etc.

- Multi-level orchestration (as mentioned, a characteristic of aerOS service orchestration approach).
- Definition of the elements of the aerOS orchestration (High Level Orchestration – HLO and Low Level Orchestration – LLO).
- Details, potential implementation and interactions of the High Level Orchestrator – HLO.
- Encapsulation of the services to be orchestrated.
- Details, potential implementation and interactions of the Low Level Orchestrator – LLO.
- Integration of HLO and LLO in aerOS.

Before digging deep into the research so far and the diagrams, it is worthwhile to contextualise the work conducted. Effectively, it has been defined in aerOS that K8s will be the most usual environment where the workload execution will take place in the Meta-OS. While the project does not constrain itself to only investigate K8s-related stuff (as, logically, other environments will exist), the research in the orchestration field in the action during the first year has been biased by K8s perspective. It is expected that, while this flavour will predominate, further research will be conducted outside the charts. For instance, specific LLOs will be developed that will tackle the deployment of workloads in non-K8s environments (e.g., just Docker as container management framework).

3.3.1.1.1. Multi-level Orchestration

The aerOS services orchestration system is inspired from the multi-layer orchestration approach proposed in [1]². In this approach, a scalable service orchestration architecture is proposed whereby services are defined in

² Co-authored by several aerOS researchers.

the format of blueprints and different cloud domains are considered. The high level concept beneath the approach is depicted in Figure 17. Multi-level Orchestration approach as a source of inspiration to aerOS orchestrator .

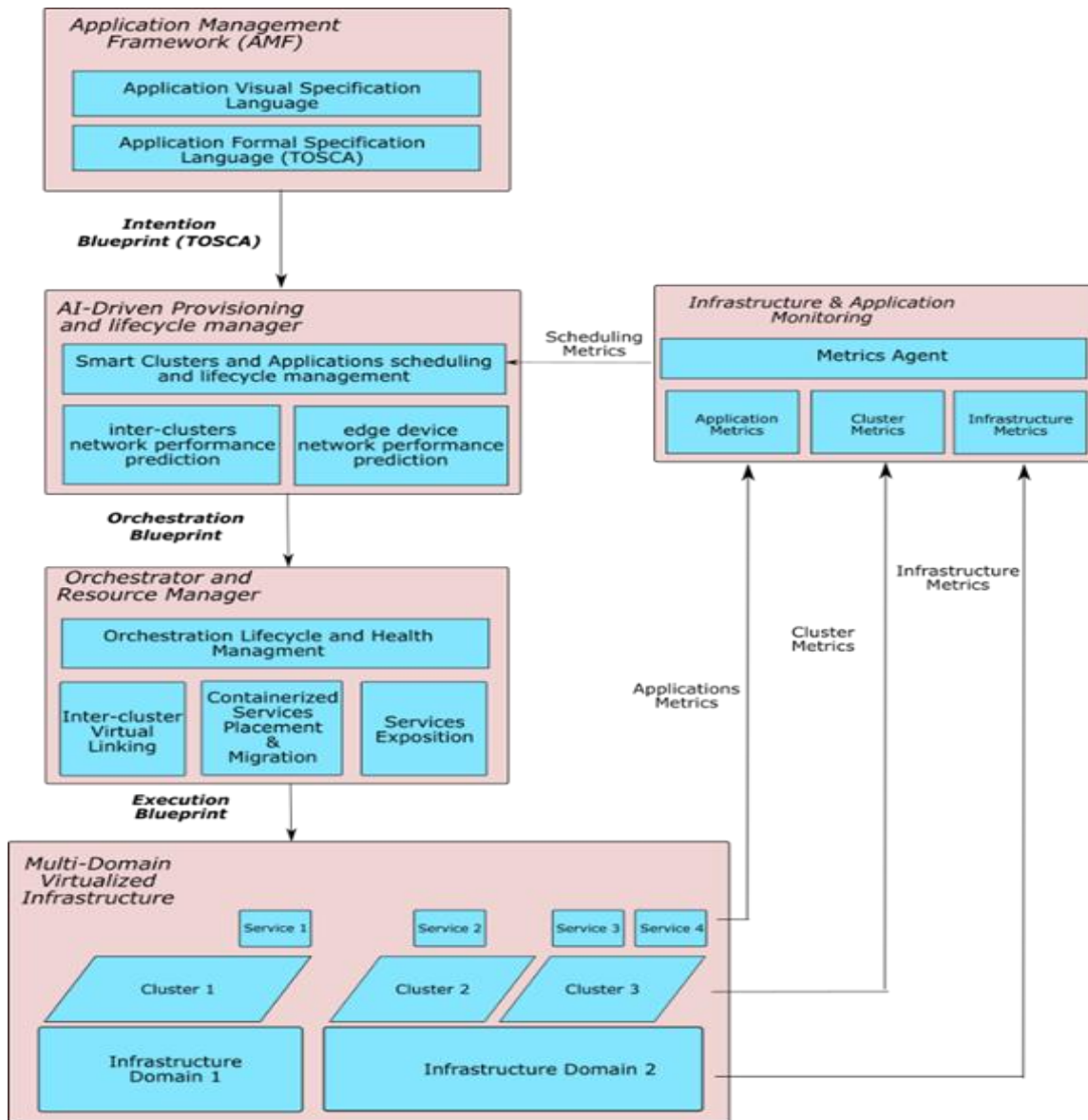


Figure 17. Multi-level Orchestration approach as a source of inspiration to aerOS orchestrator [1].

To better fit the approach with the aerOS framework, a thorough review of the designs adopted in [1] is provided below.

Application Management Framework

At this layer of the architecture, a user defines and expresses the application composition using an intuitive User Interface, independently of the infrastructure details. The visual language is then transformed to a formal Intention Blueprint using the TOSCA industry standard. It provides a high-level specification of the application while providing an opportunity for the smart layer (i.e., HLO) to tweak the infrastructure details to achieve the optimal performances. In the formal format, the modular services composing the application are defined, as well as the requirements in terms of resources constraints such as RAM and number of cores, number of replicas, and the expected Quality of Services such as bandwidth, latency or jitter.

AI-driven provisioning and lifecycle manager (High Level Orchestrator)

At this layer of the architecture, the best scheduling plans associated with the application Intention Blueprint are devised using native AI mechanisms. The plans are based on both the infrastructure capabilities (e.g., support or not for GPU) and optimization criteria such as security constraints, energy, efficiency or user-proximity. The scheduling results in a subsequent Execution Blueprint which provides the infrastructure details concerning the clusters and services provisioning. This layer harnesses monitoring metrics for predicting the resources usage to achieve a proactive scheduling, as well as for aggregating real-time feedback of the scheduling performances.

Infrastructure & Application Monitoring

At this layer of the architecture, a set of monitoring agents are put in place to gather different types of metrics to provide HLO with the required metrics to predict resource and network usage as well as performance feedbacks. The component provides three types of metrics: application, cluster, and infrastructure metrics, respectively. Each type of metrics differs in the layer at which they are aggregated. Here, in the context of aerOS, the Data Fabric (see deliverables D2.6 and D4.1) will play a fundamental role.

Orchestrator and Resource Manager (Low Level Orchestrator)

At this layer of the architecture, the orchestration decision abstracted in the Orchestration Blueprint is transformed and enforced through an actual execution plan represented by the execution Blueprint. This latter is consumed by the virtualized infrastructure. This allows building the edge-cloud continuum infrastructure in a uniform and abstracted manner. The component ensures not only that the decisions are translated into the right low-level blueprints, but also enforces that no failure resulted in the process. A remediation action is put in place in case of crashes or noticeable bugs.

3.3.1.1.2. aerOS Service Orchestration Approach

Considering the robustness and scalability of the presented orchestration approach, the aerOS Service Orchestration architecture proposes an innovated (inspired from the previous) federative multi-domain architecture (as shown in Figure 18, detailed in D2.6).

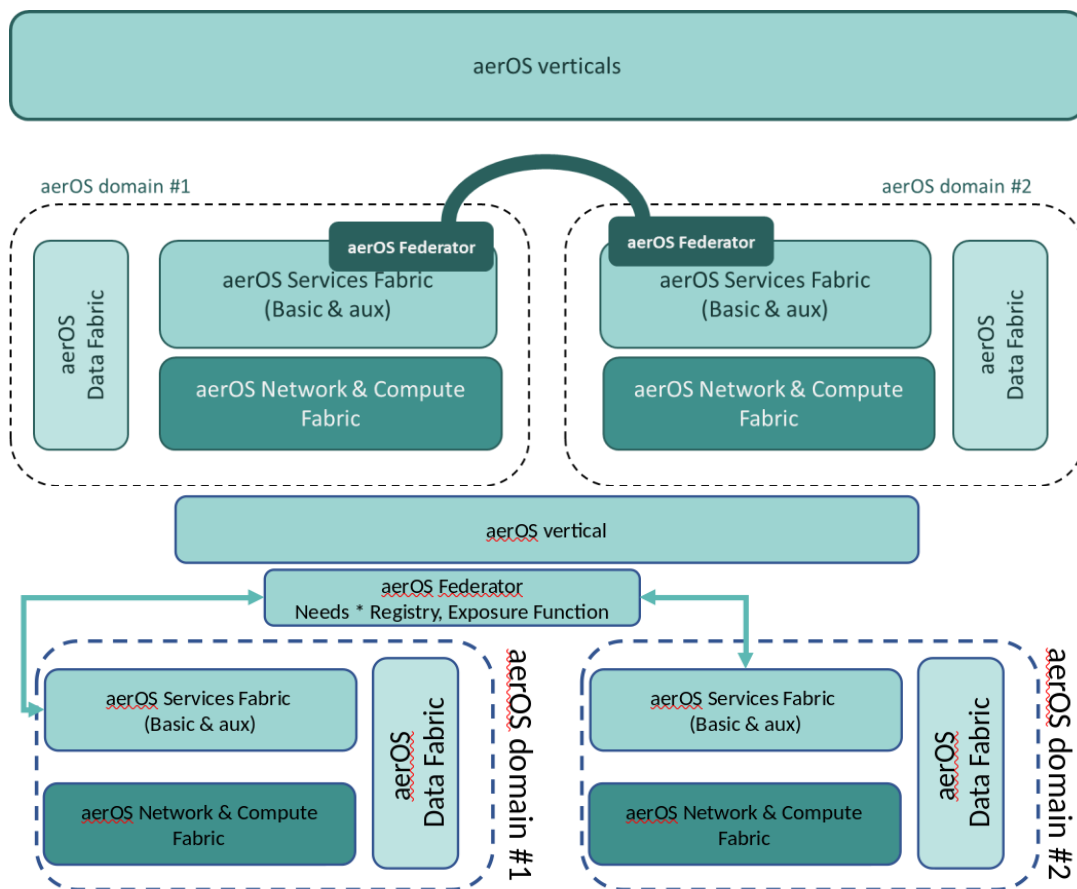


Figure 18. aerOS Federative architecture

Here, several administrative domains will exist in a continuum. Each of these domains will contain a Data Fabric, a Services Fabric and a network and compute fabric. On top of those, aerOS verticals (applications from external users, e.g., pilots) will run. In order to allow the proper functioning of the fabrics among domains, an element called aerOS Federator comes in play. The aerOS Service Orchestration system must acknowledge this structure in order to properly define its architecture and the implementation of its components. As illustrated in Figure 19 (introduced in D6.2), it consists of two main parts:

- **High Level Orchestration (HLO):** It is composed of an AI-powered Decision-Making Engine for the service orchestration. The aerOS vertical submits the services requirements blueprints for execution in the continuum.
- **Low Level Orchestration (LLO):** It enforces the decision made by the high-level orchestration. It also ensures the proper execution of the services inside the selected domain.

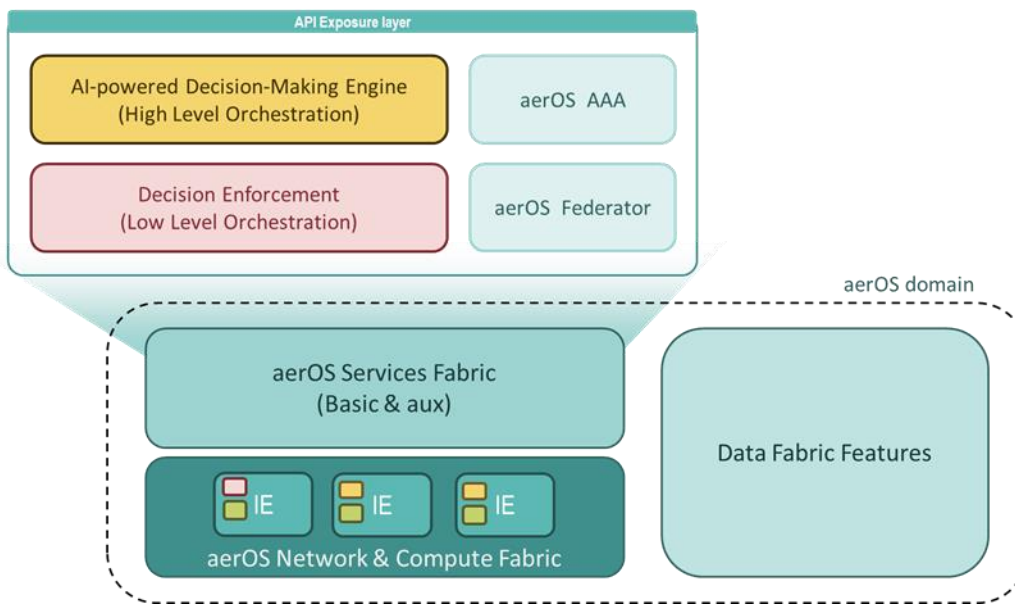


Figure 19. Aeros Services Orchestration

The remainder of this subsection provides further details about the aerOS architecture as well as the motivation behind the two types of orchestrations.

3.3.1.1.3. Decentralized High-Level Orchestration

In aerOS, there is the initial differentiation between three aerOS users for the orchestration: system administrator, IoT service developer, and IoT service provider. This is aligned with the users and narrative proposed in the global architecture of aerOS (see deliverable D2.6). Each of these users concentrates on different aspects of aerOS and, thus, has requirements that can, but not necessarily, overlap. For example, possible requirements based on the user can be:

- guarantee the execution of the service even on hardware failure (IoT service provider)
- connect to the Data Fabric and use sensor data provided by the continuum (IoT service developer)
- optimize orchestration so that the least amount of energy consumption is necessary for the execution of the service (system administrator)

To adhere to all the multi-stakeholder requirements for placement, a decision has been made to split the orchestration into high-level and low-level orchestration. While the low-level orchestration allocates resources to the services based on technologies like K8s or docker, the high-level orchestration uses machine learning to select the best IEs for running a services. Although a K8s cluster is already capable of allocating computational

resources to a service, it might not consider additional requirements such as energy consumption, and for sure it does not consider the whole continuum status in a timely and predictive manner.

Right now, some solutions have been identified that address similar problems. One has been developed in the IntelliIoT project with Mixed Integer Programming (MIP). Solving the MIP problem returns one of the most optimal placements (if multiple exist) regarding energy consumption while adhering to the available computational resources, latency, and bandwidth constraints. In addition, the solution considers the capability requirements of the service, e.g., hardware to display a Graphical User Interface (GUI). As mentioned, the current, complete solution is based on a MIP algorithm. Even though this returns the most optimal solution, the approach only scales for small networks as input. Finding a solution takes a long time, mainly if including constraints based on the network between devices, like latency and bandwidth. Furthermore, the approach needs to find an optimal solution for every network change again.

As a result, the current approach does not fit the aerOS continuum. The devices in the aerOS continuum differ depending on the aerOS vertical, and the available IEs will vary from case to case and also among them, statically and during the time. Only to mention an example, in [pilot 1](#), the aerOS continuum can be very dynamic with moving IEs on Autonomous Guided Vehicles (AGVs).

aerOS proposes a decentralised orchestration solution as depicted in Figure 20. It shows the main concept of this decentralised orchestration process limited to one domain. A rationale (more contextualized to the global aerOS view) can be found in Section 4.2.2 of deliverable D2.6.

To keep it simple, the focus is on the connectivity between IEs and the responsibilities (blue borders) of the high-level orchestration component, which is represented in yellow. The blue squared boxes represent a set of IEs within a domain that share the same LLO (or O in the diagram). This would be extended to other domains as the HLO communicates with other orchestrators of other domains with a publish-subscribe protocol (powered by NGSI-LD mechanisms) to synchronize the orchestration. It is assumed that the aerOS Management Portal has at least one user/basic/auxiliary service with requirements that can be placed on an IE.

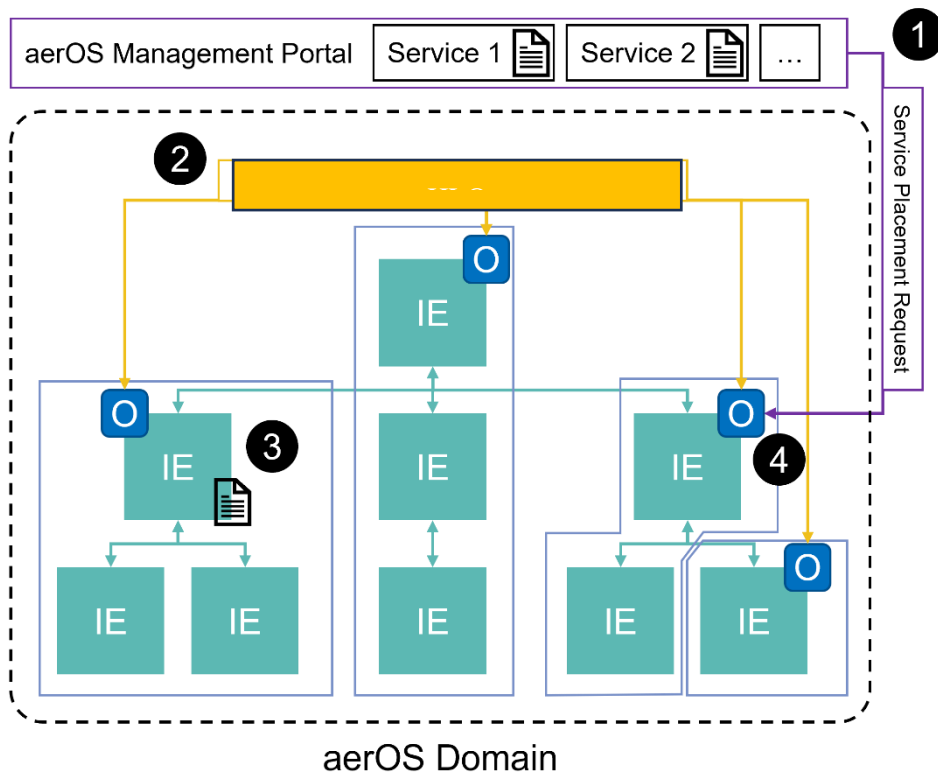


Figure 20. Decentralized High-Level Orchestration

The elements in the previous diagram can be described as follows:

Table 4. Decentralized High-Level Orchestration components

Component	Description	Interactions
aerOS Management Portal	The aerOS Management Portal is the user interface for different roles of users.	The aerOS Management Portal provides the user interface to orchestrate services in the continuum.
Service	Service is an aerOS service. This can be a user, a basic or an auxiliary service.	The high-level orchestrator should orchestrate each provided service on the continuum.
Service Placement Request	The user can request an orchestration via the aerOS Management Portal. This request is handed over to the high-level orchestration first.	The aerOS Management Portal request an orchestration from one of the distributed high-level orchestration services.
High-level orchestrator (HLO)	The high-level orchestrator is a service that returns a resource allocation and placement given specific data from the data fabric and the aerOS Management Portal as input (see Figure 21). It may use Deep Reinforcement Learning (with a deep neural network) to return the resource allocation and placement.	The high-level orchestrator interacts with the aerOS data fabric as input to return a decision on the placement of services.
Low-level orchestrator (LLO)	The low-level orchestrator is in charge of interpreting the execution order into actual workload deployment	Receives orders from the HLO (via Decision Blueprint) and interacts with IEs' container management frameworks
Inter-Orchestration Communication	High-level orchestrator can communicate with each other (other domains) to share information about orchestration requests and their possible placements.	High-level orchestrators can interact with each other (from other domains). In the future, data from a high-level orchestrator could be shared over the data fabric as well.
aerOS Domain	The decentralized high-level orchestrator concept is currently limited to one domain. Depending on the Federation Manager between domains, this concept can be expanded in the future.	The high-level orchestration focuses on one aerOS Domain.
Service Requirements	Each service has at least computational resource requirements, e.g., processing power, memory, and storage. In addition, a service can rely on latency or bandwidth requirements if it is dependent on other services.	Service requirements can be either included in the service description or gathered by data from the data fabric.
Data Fabric	The aerOS Data Fabric is	The data fabric provides necessary data for the orchestration decision of the high-level

	described in D2.6 and in D4.1.	orchestration.
Placement	A placement is a mapping between services and IEs. It is the output of the high-level orchestrator.	The placement could be used by the low-level orchestration service.

Thus, the orchestration process can be primarily described in three steps:

1. An authenticated user selects one or more services to be orchestrated in aerOS via the aerOS Management Portal. The aerOS Management Portal forwards this request to a high-level orchestrator (HLO).
2. The HLO analyses the status of the services and resources (IE) of its own, therefore trying to orchestrate the request on their area of responsibility. All possible placements are, then, evaluated by the orchestrator based also on the incoming request.
 - a. If the most optimal placement in terms of QoS requirements is within its scope, the service will be deployed in the domain, making use of the proper LLO (or “O” in the diagram) to have the workload executed on the corresponding IE.
 - b. If not possible, it analyses the state of the whole continuum through information gathered by the Data Fabric as shown in Figure 21 and then the Federation mechanism and the inter-domain connection will act and the request will be properly forwarded.
3. The process would repeat in the associated HLO of the forwarded domain.

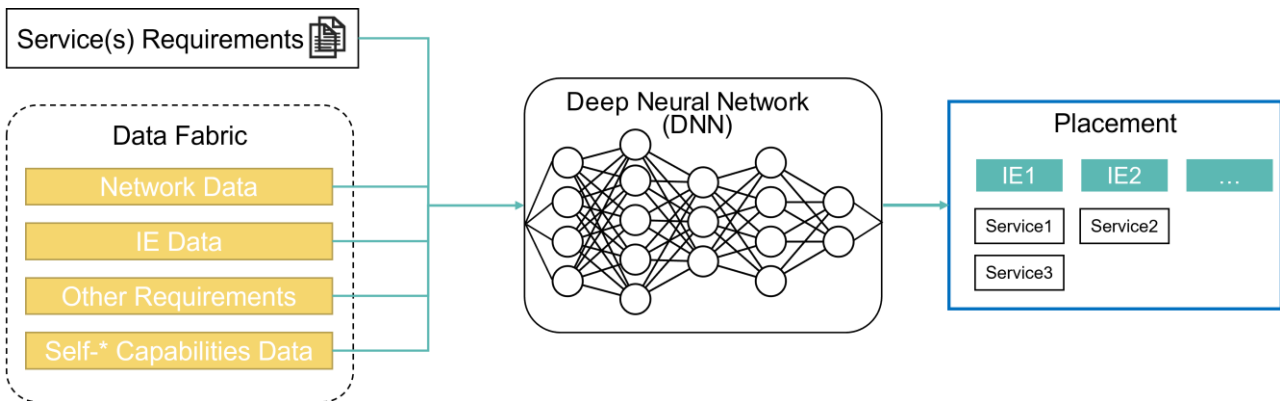


Figure 21. Detailed AI-powered Decision-Making Process

Figure 21 shows the process of the high-level orchestration component, also known as AI-powered Decision-Making Engine, in more detail. To receive a placement, data from the Data Fabric and the QoS requirements of the service(s) are used. The data needed from the aerOS Data Fabric can be split into:

- Network data e.g., available bandwidth, maximum latency
- IE data e.g., available processing power, memory, storage, the availability of a GPU or display
- Self-* capabilities data e.g., health and monitoring data
- Other requirements e.g., thresholds that are defined by the system administrator.

Requirements and data are used as input for a Deep Neural Network. The Deep Neural Network returns a mapping between IEs and services that is, then, used as the source for generating *Decision Blueprints* for the low-level orchestration. As a technology, some research has already been conducted about Deep Reinforcement Learning with the *stable-baselines3* Python library.

As it can be seen, the novel approach proposed by aerOS overcomes the issue observed about the scalability problem of MIP in Intelliot, creating a much more reliable architecture due to the redundancy of multiple high-level orchestrators (one per domain).

3.3.1.1.4. Low-Level Orchestration (and target case for Kubernetes environment)

The Low-Level part of the aerOS services orchestration receives formal services specification blueprints from the high-level orchestration and translates these blueprints into running services. Hereunder, the assumption that aerOS is working with K8s environments (and domains where only K8s exist) is made. This will not be the generic case for all domains nor IEs of aerOS. This has been only the first exercise to define, implement and deploy a LLO in aerOS. It is expected that more than one type of LLOs will be developed in the project. aerOS, as a platform-agnostic Meta-OS will need to support diverse container management frameworks underlying in the infrastructure, therefore, the efforts specified below will be replicated for other types of deployment cases.

Effectively, assuming that the underlying framework would be K8s, the implementation diagram of a LLO would look as depicted in Figure 22.

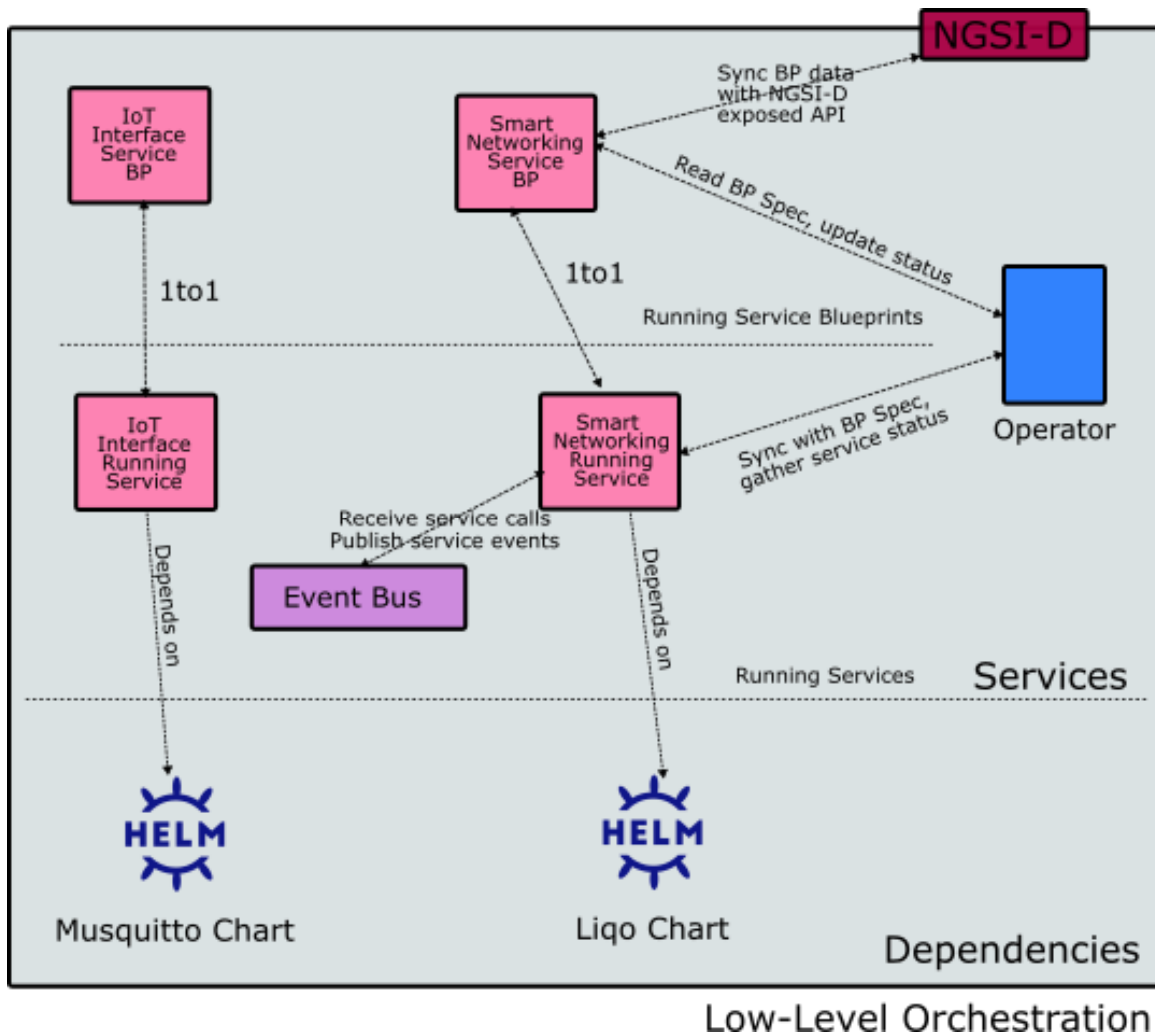


Figure 22: aerOS services Low-Level Orchestration in the case of K8s framework

As detailed in the illustrated example, the low-level orchestration in the case of K8s implementation is decomposed into two layers: services and dependencies. Table 5 explains each low-level orchestration components.

Table 5. Low-Level Orchestration Components in the case of K8s framework

Component	Description	Interactions
Low-level Orchestrator (Operator)	<p>The operator, being the low-level orchestrator, keeps in sync with the running services and the corresponding blueprints.</p> <p>The operator propagates the configuration from the specification to the running service. On the other hand, the running service status is replicated in the blueprint.</p>	<p>It interacts with both the running service and the service blueprint to keep the service configuration and status information in sync.</p> <p>It also interacts with the events bus to relay initial and updated configurations during the lifecycle of the running services.</p>
Service Blueprint	<p>During the lifecycle of the running services, the associated blueprints are created and updated.</p> <p>It is composed of two sections: the spec section containing the service specification of the running service, and the status section replicating the current running service status.</p>	N/A
Running Service	It represents a running instance of an aerOS service. Its lifecycle is managed by the low-level orchestrator (Operator).	It interacts with other services and the low-level orchestrator through events data using the Events Bus as the medium.
Events Bus	<p>The Events Bus plays a central role for the interaction of services with the low-level orchestrator and other domain services.</p> <p>A set of standard events for the requests and configurations lifecycles are exchanged between a service and other entities (low-level orchestrator or domain service) through the Events Bus.</p>	It interacts with the low-level orchestrator and the running service to relay the events data.
Third-party packages dependencies	<p>Different services may depend on third-party packages for their internal operations.</p> <p>A smart-network service for example depends on Ligo which provides inter-clusters VPN tunnelling, or an IoT interface service depends on Musquitto for providing MQTT protocol interface.</p> <p>Helm is used as the packaging system for these dependencies.</p>	N/A
NGSI-LD service	The NGSI-LD service which provides a smart data aggregation of the orchestration system keeps track of services specifications and statuses in its internal database. It will interact with the Data Fabric of the domain.	N/A

In order to develop the **LLO in the case of K8s**, certain knowledge about the functioning of it is needed. In the following, a reflection on the expression of the replicas and the concept of “operators” is provided in order to contextualize how such LLO could be implemented.

K8s provides an automated lifecycle management system of any stateless application or unit of work. All instances of this unit of work are by definition interchangeable. They are called replicas by the terminology of K8s [2]. At the infrastructure level, K8s manages a cluster. It represents a set of computerized hardware called nodes. Basic units of works are run inside those nodes called pods. Commonly, a set of Linux containers using common resources (networking, storage, shared memory) constitute a pod [2]. K8s is decomposed of two building blocks [2]:

- **The application or data plane:** It represents the nodes dedicated to running the applications pods. These nodes are called worker nodes in K8s terminology.
- **The control plane:** It represents the core K8s management system. A set of control nodes are dedicated to host the control plane pods which implement the K8s API and the orchestration logic.

The main components of the control plane are the controllers. They ensure that the cluster target state is achieved. If the target state diverges from the current state, a remediation action is triggered. This is particularly accomplished through a set of control loops. Nevertheless, an application has usually an associated state which is important to stay consistent across time to remain reliable. The internal application state management falls outside of the K8s control plane, being aware of this fact.

Hence, to extend the features offered by controllers, operators have been created inside K8s. As illustrated ([2] - Figure 23), to achieve this extension, a custom resource or a blueprint is created as an endpoint in the K8s API. The role then of the operator is to monitor and maintain the corresponding resource. The operator implements an internal control loop to orchestrate and achieve the resource target state.

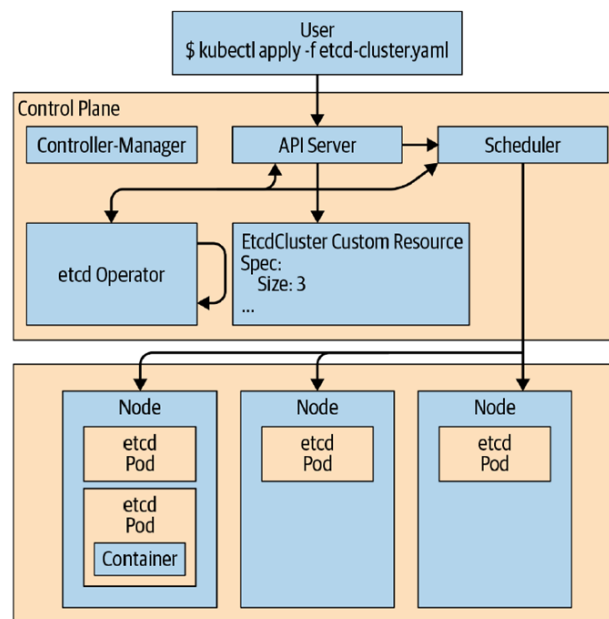


Figure 23. Operators are custom controllers watching a custom resource [2]

It is important to note that the Custom Resource is created from a Custom Resource Definition (CRD) which defines the data schema of any associated resource instance. The development of an operator involves then two main tasks [2]:

- The creation of a CRD: It provides the formal schema definition of all the resource type instances.
- The development of a **watcher**: a loop running application that watches the resource type instances and responds to changes. The kind of resources remains specific to each type of target applications the Operator is managing.

Unfortunately, the internal development of the watcher program is quite complex. An operator should handle a set of possible resource-related scenarios, named “maturities”. Different operators illustrate then distinct levels of maturities. Four levels of maturity are possible and the main development objective is then to achieve and improve the maturity level [3] [4].

- **Maturity Level 1 – Basic Install:** The operator installs and configures the workload based on the corresponding custom resource.
- **Maturity Level 2 – Seamless Upgrades:** The operator can be seamlessly upgraded to a new version. The target workload can also be upgraded based on custom resource update.
- **Maturity Level 3 – Full Lifecycle:** The operator provides the full lifecycle management features. The operator can then provide fail-over and backups at this level.
- **Maturity Level 4 – Deep Insights:** At this level, the operator exposes health and performance metrics and emits custom k8s events.
- **Maturity Level 5 – Auto Pilot:** At this level, the operator provides auto scaling, auto-healing, auto-tuning and abnormality detection.

From these observations, it is thus mandatory to start with a basic operator providing the initial building blocks of a LLO and improve it to achieve the target maturity level, at each stage.

3.3.1.1.5. Encapsulation of the services to be orchestrated

Whenever envisioning the deployment of a LLO, it is always very relevant to understand how the workloads to be deployed will be expressed. This is still an on-going discussion in aerOS, and more hints on the design and decision will be provided in successive deliverables (i.e., D3.2 and D3.3). Relevant possibilities (that would have extreme relevance for the development of the LLO for the case of underlying K8s) could be:

- **Juju charms:** Juju is a tool for topology-based applications. Based on a YAML DSL (Domain Specific Language), it allows application deployment modelling. Different cloud offerings and services are supported for the deployments [5]. At the core of Juju are charms. They define all the instructions and artifacts required for the deployment and configuration of application components. For customization purposes, configuration values can be set during the deployment step. Furthermore, charms can be grouped in bundles and provide dependency mechanisms [5]. Actions can be also defined and implemented as Unix shell scripts and triggered by runtime during the deployment. These scripts are used to install, configure, and wire software components [5] [6]. Charms require a dedicated Juju environment to run, which makes it less flexible compared to other alternatives, mainly Helm Charts. In fact, Juju provides a complete orchestration system, beyond operators packaging.
- **Helm charts:** Helm charts are the other and more widespread alternatives to Juju charms for operators packaging. Released by Cloud Native Computing Foundation (CNCF) in 2015, Helm is the defacto K8s packaging manager to ease the burden of K8s configuration [7]. Helm allows K8s configuration files to be assembled into packages. Local and remote repositories can then be used to share these reusable packages, which are called charts. Dedicated commands through CLI can then be used to install, update or uninstall them [7]. To achieve automatic package management, charts can also refer to other ones as dependencies in their configuration files. Coming back to the illustrative example of a LLO for underlying K8s environment, it is important to note that Helm charts can be used to package more than operators, but remain one of the most interesting aspects of helm packaging. Operators helm packaging are enabled by integrating the CRD into the charts configurations files as well as a pod definition of the watcher program. It remains the simplest and the most flexible way of deploying operators into K8s.
- **OSM VNF and NS packaging Approach:** Network Function Virtualization (NFV) is a set of standards proposed by ETSI to decouple network functions (e.g., firewall, NAT, and caching) from the dedicated hardware by running them in a virtualized environment [8]. Under the umbrella of ETSI, an opensource implementation of MANO is being developed called Open Source MANO (OSM). OSM provides a NFV platform to align with the NFV information models while meeting the production requirements. The opensource implementation incorporates both resource and service orchestrations [8].

OSM is here to be considered as an example for packaging VNFs (Virtualized Network Functions) and NS (Network Services) before the deployment of the associated instances. Two possible execution environments are possible for the deployments in this regard: K8s [9] or a virtualized infrastructure [10] such as OpenStack or a Cloud IaaS. In what follows, the packaging process of VNFs and NS for the K8s cluster and a helm chart KDU (Kubernetes Deployment Unit) are described. The implementation of primitives in a Helm-chart environment is also discussed.

To deploy a Kubernetes-based Virtual Network Function (KNF) in OSM, a running K8s is required after being associated to a VIM (Virtual Infrastructure Manager). Dedicated packages for the KNF need to be created and saved inside OSM before deployment [11]. In Figure 24 ([12]), a yaml definition of a VNF package is deployed inside a K8s cluster.

```
vnfd-catalog:
  schema-version: '3.0'
  vnfd:
  - connection-point:
    - name: mgmt
    description: KNF with KDU using a helm-chart for Facebook magma orc8r
    id: fb_magma_knf
    k8s-cluster:
      nets:
      - external-connection-point-ref: mgmt
        id: mgmtnet
    kdu:
    - helm-chart: magma/orc8r
      name: orc8r
    mgmt-interface:
      cp: mgmt
    name: fb_magma_knf
    short-name: fb_magma_knf
    version: '1.0'
```

Figure 24. KNF VNFD Example [12]

In the illustrative example, besides the important network configurations, the package description contains the helm-chart parameter in the KDU (Kubernetes Deployment Unit). This allows OSM to deploy the associated operator for the VNF package. Moreover, a Network Service (NS) package is also needed, which is illustrated in Figure 25 ([12]).

```
nsd-catalog:
  nsd:
  - constituent-vnfd:
    - member-vnf-index: orc8r
      vnfd-id-ref: fb_magma_knf
    description: NS consisting of a KNF fb_magma_knf connected to mgmt network
    id: fb_magma_ns
    name: fb_magma_ns
    short-name: fb_magma_ns
    version: '1.0'
    vld:
    - id: mgmtnet
      mgmt-network: true
      name: mgmtnet
      type: ELAN
      vim-network-name: mgmt
      vnfd-connection-point-ref:
      - member-vnf-index-ref: orc8r
        vnfd-connection-point-ref: mgmt
        vnfd-id-ref: fb_magma_knf
```

Figure 25. NSD Example [12]

The network service creates a virtual link (mgmtnet) that references the VNF connection point (mgmt.) in the VNFD created in the first step. OSM primitives are managed operations to allow the automatic initialization of the exposed services inside the VNF, right after instantiation [13]. Both juju charms-based and Helm Chart-based are possible, but we focus here in the later without losing in generality. Helm Chart-based executions environments can be also used to configure NF (Network Functions), starting from OSM version 8. This results in deploying additional pods inside the OSM K8s cluster namespace if integrated [13]. The first step is to implement the primitives' actions required for the NF configuration. They are defined inside a python class. The class file is created inside the helm chart folder. The implementation is illustrated in Figure 26 ([13]).

```

import asyncio
import logging
import asyncssh

from osm_ee.exceptions import VnfException

class VnfEE:

    def __init__(self, config_params):
        self.logger = logging.getLogger('osm_ee.vnf')
        self.config_params = config_params

    async def config(self, id, params):
        self.logger.debug("Execute action config params: {}".format(params))
        # Config action is special, params are merged with previous config calls
        self.config_params.update(params)
        required_params = ["ssh-hostname"]
        self._check_required_params(self.config_params, required_params)
        yield "OK", "Configured"

    async def touch(self, id, params):
        self.logger.debug("Execute action touch params: '{}', type: {}".format(params, type(params)))

        try:
            self._check_required_params(params, ["file-path"])

            # Check if filename is a single file or a list
            file_list = params["file-path"] if isinstance(params["file-path"], list) else [params["file-path"]]
            file_list_length = len(file_list)

            async with asyncssh.connect(self.config_params["ssh-hostname"],
                                       password=self.config_params.get("ssh-password"),

```

Figure 26. Primitive actions implementation in a Helm Charts-based execution environment [14]

In the illustrated example, the async methods config and touch are the primitives' actions implementations. The next step is to reference these primitives' actions inside VNFD (Figure 27; [13]).

```

vnfd:
  ...
  df:
    - ...
    lcm-operations-configuration:
      operate-vnf-op-config:
        day1-2:
          - id: simple_ee-vnf
            config-access:
              ssh-access:
                default-user: ubuntu
                required: true
            execution-environment-list:
              - external-connection-point-ref: vnf-mgmt-ext
                helm-chart: eechart
                id: monitor
            initial-config-primitive:
              - execution-environment-ref: monitor
                name: config
                parameter:
                  - name: ssh-hostname
                    value: <rw_mgmt_ip>
                  - name: ssh-username
                    value: ubuntu
                  - name: ssh-password
                    value: osm2020
                seq: '1'
              - execution-environment-ref: monitor
                name: touch
                parameter:
                  - name: file-path
                    value: /home/ubuntu/first-touch
                seq: '2'

```

Figure 27. Helm-chart based primitives inside the VNFD [13]

In the illustrated example, both config and touch primitives are referenced in the initial configuration primitives and a set of parameters are passed through.

3.3.1.1.6. Low-Level/High-Level Orchestration Integration

Figure 28 illustrates how the low-level and high-level orchestration are integrated. The high-level orchestrator receives metrics from the low-level orchestration system (potentially using other elements of the aerOS Meta-OS such as the Embedded Analytics tool – see deliverable D4.1- or via the self-functionalities of the IEs – see Section 3.5). The data are either retrieved from the blueprint status section or from the aggregation of service events data received through the events bus.

Based on the different performance metrics, the high-level orchestrator harnesses an AI Model engine for both prediction and decision-making purposes. The decision is then communicated to the low-level orchestration through the generation of a new service blueprint. The low-level orchestrator (operator) detects the change in the blueprint and propagates it to the running service.

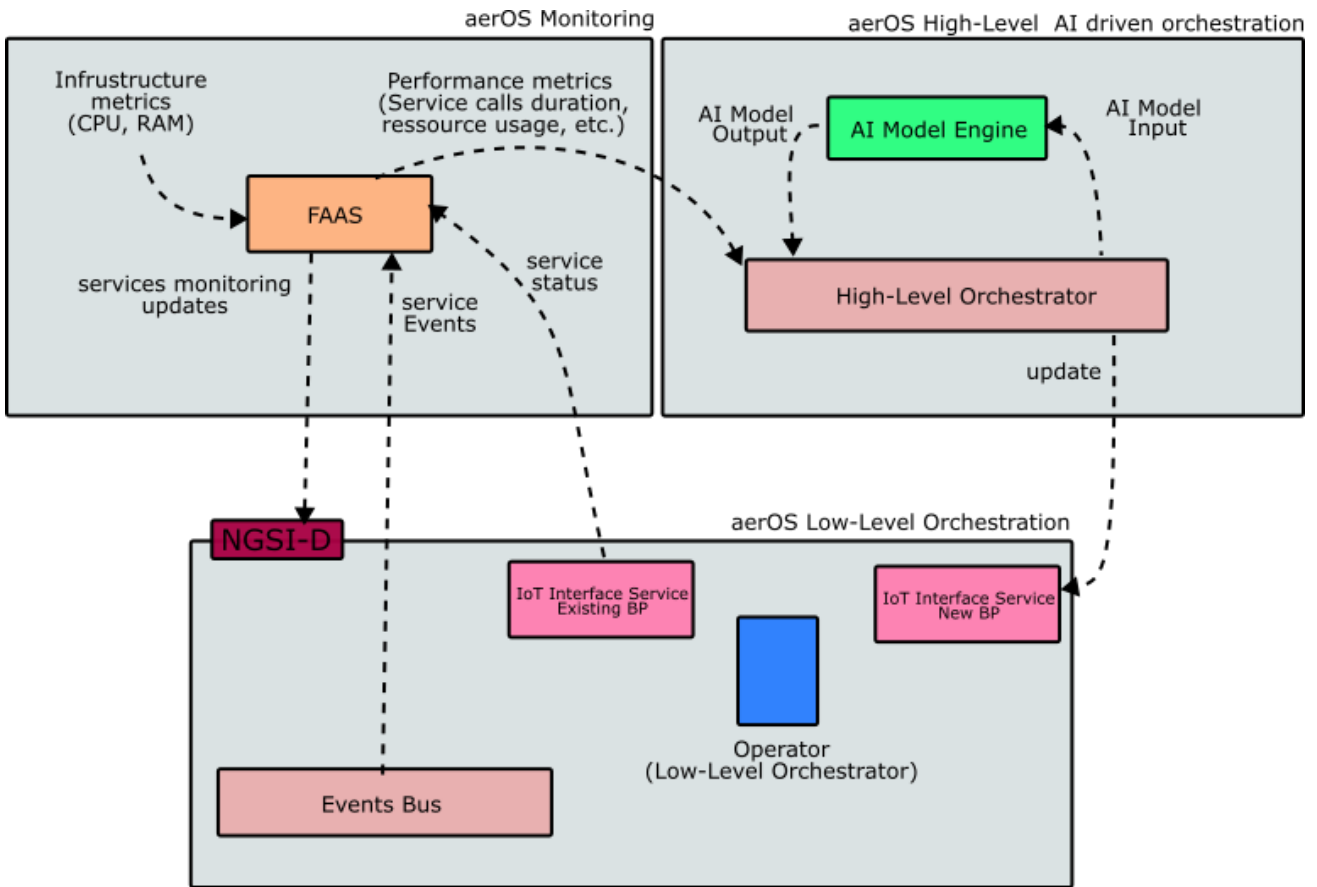


Figure 28. High-Level/Low-Level Orchestration Integration

3.3.1.2. Resources orchestration

Objectives

Resource orchestration focuses on creating infrastructure provisions that could later join the continuum. It coordinates the allocation and utilisation of various resources, such as computing power, storage, network bandwidth, and other system components, to meet the needs of aerOS services. Given the nature and the objectives of the aerOS project, this crucial step will be dealt with later (prioritising service orchestration) towards balancing resources in order to ensure the continuum experience and leverage the capabilities of every underlying hardware.

Functionalities provided

Resource orchestration for the aerOs services aims to allocate resources in an optimal manner, ensuring that the right resources are assigned to the right tasks or workloads at the right time. By intelligently taking advantage of “raw” resources based on demand, priority, and availability of the steady state of the continuum, resource orchestration may help maximise resource utilization and efficiency and may alleviate the burden of system administrators to do double work of installation and configuration. A higher level of optimization might be achieved with an AI powered decision-making process. This will be discussed further in next iterations of this document. Resource orchestration focuses on ensuring system reliability and fault-tolerance by implementing redundancy, load balancing, and failover mechanisms to maintain system availability and handle failures gracefully. Resource orchestration helps prevent single points of failure and ensures that resources are distributed to mitigate risks.

Specific characteristics of the solution

As per service orchestration, also resource orchestration is mediated by an AI based decision system. This ensures less human work associated with the definition of specific workloads or with the management of all the use cases. With AI powered resource orchestration, the resource deployment could be defined on a higher level of abstraction, leaving all the complex calculations and decisions to specific algorithms and systems.

Figure 29 illustrates the different components building the aerOS resource orchestration system and its communication channels. The system harnesses the existing service orchestration with resource-specific orchestration components.

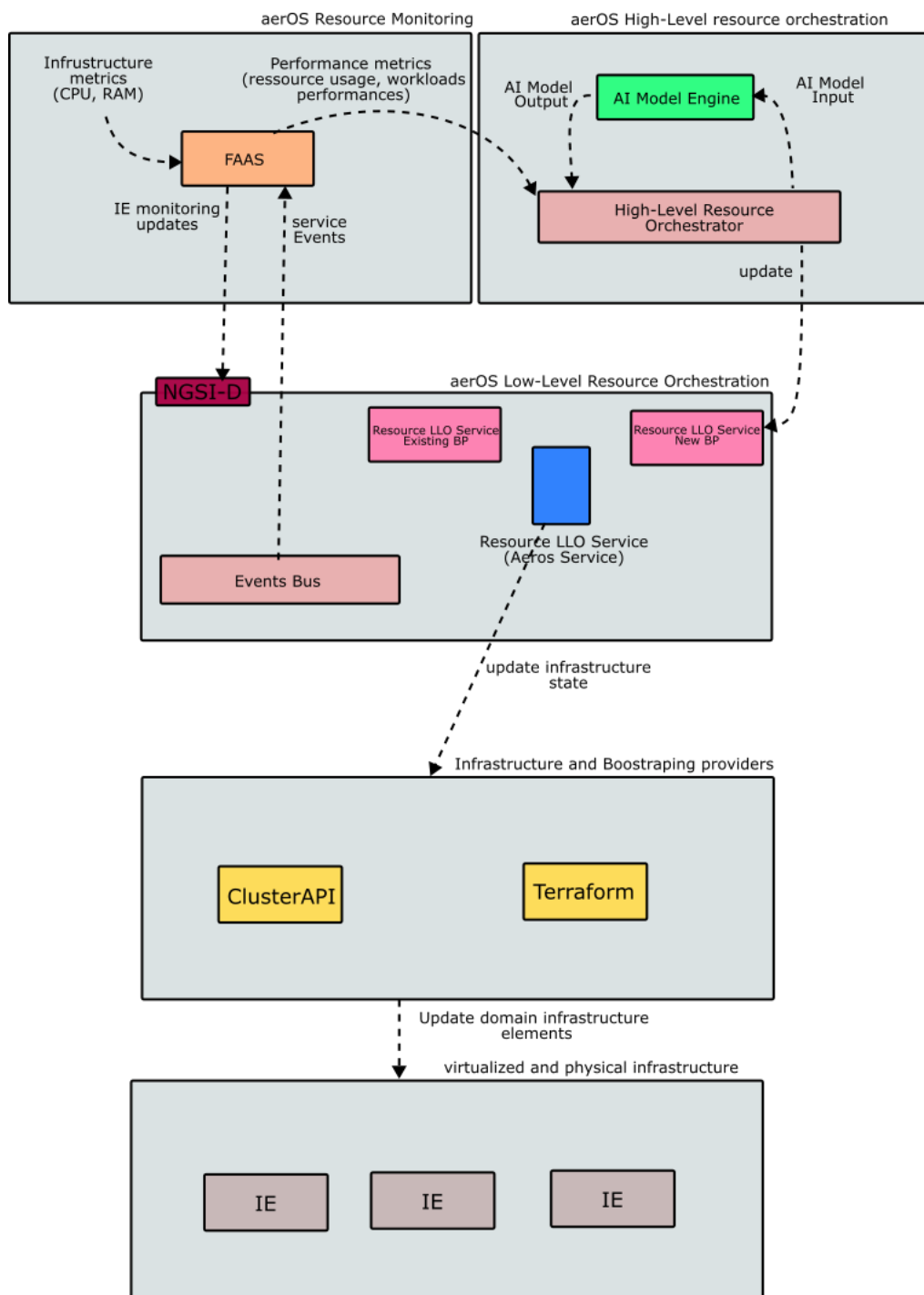


Figure 29. aerOS resource orchestration architecture

The following table contains each component details:

Table 6. aerOS resource orchestration components

Component	Description	Interactions
High-level Resource Orchestration	<p>The existing service orchestration is harnessed for the deployment of a resource-centric high-level orchestrator to overlook the decision-making part of the resource orchestration.</p> <p>The high-level orchestrator predicts the resource usage, ensures resource availability and good QoS from the aggregated metrics communicated by the monitoring component.</p>	<p>It communicates with the low-level orchestrator through its associated service blueprint.</p> <p>It gets information of resources under/over utilization and of the deployed workloads QoS compliance from monitoring metrics.</p>
Monitoring	<p>The existing aerOS monitoring component is harnessed with dedicated resource-centric metrics. Two types of metrics are used as a result. Resource usage (CPU, Memory usage etc.) metrics and services performances metrics.</p>	<p>The resource usage metrics and services performances metrics are communicated by the monitoring component to the high-level orchestrator.</p>
Low-level Resource Orchestration	<p>The existing service orchestration is harnessed for the deployment of a dedicated service for the low-level resource orchestration.</p>	<p>The orchestration service keeps in sync its blueprint with infrastructure state through the interaction with the components in the infrastructure and bootstrapping providers.</p>
Infrastructure and Bootstrapping Providers	<p>A set of providers are used to manage the deployment of the machines hosting the infrastructure elements inside the domain. There is also a set of providers to ensure the initial configuration (Bootstrapping) of the machines before their integration into the domain as resources.</p>	<p>The providers interact directly with underlying infrastructure to deploy and bootstrap the resources.</p>
Virtualized and Physical infrastructure	<p>A set of aerOS infrastructure elements builds up the domain resources of the virtualized and physical infrastructure. Depending on the current workload and the requested QoS of the deployed workloads, the infrastructure elements will be added or removed from the domain resources.</p>	N/A

3.3.2. Related requirements

- TR-7 Meta-operating system orchestration and AI enabler
- TR-10 Dynamic resources
- TR-18 Multi-domain services orchestration
- R-T43-4 User requirements for AI job
- R-T43-8 AI jobs orchestration
- R-T43-11 User requirements to resource matching
- R-T43-11 User requirements to resource matching

3.3.3. Candidate technologies and standards

Table 1. Candidate technologies and standards for Service Orchestration

Tech/Standard	Description	Justification
Deep Reinforcement Learning	We research the use of Deep Reinforcement Learning as an approach to receive a service-IE placement based on input data.	A trained deep neural network has a smaller footprint than Mixed Integer Programming to find a valid placement.
DVC	Git integrated tool for data versioning	As over the time, change of the dataset may apply due to changed feature engineering or new datapoints, it is essential for the reproducibility of any machine learning application to version the used data.
KubeFlow	Potential open-source solution for having reproducible and visual configurable training pipelines	As it is intended to use deep learning, it is necessary to create training pipelines. Most training pipelines share common components that can be reused if they are realigned. For this KubeFlow could be an existing software that could be integrated
MLFlow	Potential open-source solution for experiment tracking of machine learning experiments	Addition to KubeFlow. As machine learning application often have multiple iterations on modelling including multiple experiments, it is necessary to be able to track the results including metrics, the model, and other artifacts
Alibi/Seldom	Open-source python packages for drift detection and model monitoring	Once a model is trained, it will be deployed. To ensure long term reliability of the models, it is necessary to implement monitoring methods to alert if the data streams are changing and thus, the model performance may decay.
Kopf	an open-source framework for operator development in Python	It provides a complete set of tools to develop operators while being based on a widespread language, namely Python.
Kafka	An opensource distributed event streaming platform.	Kafka has been used by thousands of organizations. It provides a robust and a scalable solution for the Events Bus in aerOS.
Helm	Open-source solution that provides a packaging system for K8s.	Helm is the defacto packaging manager for K8s-based systems. It has been supported and released by the Cloud Native Computing Foundation (CNCF). The solution is also robust and provides access to a significant number of third-party packages.

Table 7. Candidate technologies and standards for Resource Orchestration

Tech/Stand.	Description	Justification
ClusterAPI	An open-source tool that provides declarative APIs and tooling for K8s clusters provisioning, upgrading and operating. It is used in aerOS for the deployment and bootstrapping of K8s resources.	ClusterAPI is the only active of all projects that try to simplify the management of K8s clusters lifecycle. Its declarative approach is compatible with our blueprint-based orchestration approach.
Terraform	An open-source tool to enable full-	Terraform is a mature tool used by a

	automation of infrastructure, provision, resources management. Terraform is used for the deployment and bootstrapping non-K8s resources case in aerOS.	significant number of organizations. It harnesses an infrastructure as code declarative approach, which is compatible to our blueprint-based orchestration approach.
--	--	--

3.4. Cybersecurity components

This sub-section covers the research and developments done so far in Task T3.4. The challenge lies on establishing (an aerOS-compliant) set of cybersecurity appliances, privacy techniques and policy enforcement across the continuum, aligned with the DevPrivSecOps methodology defined in the project.

Within this task, the efforts have been structured to advance in the following two lines of action:

- Identity and Access Management
- Secure API Gateway compliant with aerOS security

In conjunction, the software elements resulting from the two previous lines constitute the first two elements of **aerOS AAA** (authentication, authorisation and accountability). The third one is covered by the results of Task T4.5 of aerOS, that is initially described in the parallel deliverable D4.1. In the remainder of this subsection, the advances and decisions so far in each of those two lines are described.

3.4.1. Research lines

In the context of Cybersecurity components in aerOS, the research lines of Identity and Access Management are: (i) Identity management, (ii) Role-based access management and (iii) Secure API Gateway compliant to aerOS security.

3.4.1.1. Identity and Access Management

Identity and Access Management (IAM) is responsible for implementing the process for identifying, authenticating individuals, groups of people or software processes to have access to applications, systems or networks by associating user rights and restrictions with established identities³. IAM systems are comprised by two main components, the Identity Management (IDM) that is focused on authentication and Access Management (in case of aerOS Role-based Access Management) that is aimed at authorization⁴. The aerOS IAM will be based on Keycloak⁵ that is a popular and open-source IAM solution. Keycloak provides a single-sign-on (SSO) for individual applications, thus application developers do not need to worry about user authentication. In addition, Keycloak offers authorization services for providing secure access control for the authenticated users. The official website includes extensive documentation with configuration details, deployment setups, tutorials, and integration with external applications. Developed in Java and open sourced under Apache license. Some of its key features are:

- 1) Authentication: Integrates with external identity stores like LDAP, Active Directory, or custom stores based on relational databases. Additionally, Keycloak allows for authentication from social identity providers like Google or Facebook,
- 2) Authorisation: Keycloak supports multiple authorisation policies with different granularity. For example, access control policies can be defined based on roles (RBAC), users (UBAC) or can be context-based (CBAC), and
- 3) Standard security protocols: Keycloak is based on standard protocols like OpenID Connect, OAuth2.0 and SAML.

³ <https://searchsecurity.techtarget.com/definition/identity-management-ID-management>

⁴ http://www.cpd.iit.edu/netsecure08/KEVIN_WANG.pdf

⁵ <https://www.keycloak.org>

3.4.1.1.1. Identity management

The primary objective of identity management (IDM) is to guarantee that only authenticated users are granted access to the specific applications, systems or computing environments for which they are authorised. It includes the control of user provisioning and the on-boarding process for new users, such as administrators, internal and external users, and other interested parties. IDM also involves controlling the process of authorising system or network permissions for existing users and the removal of users who are no longer authorised to access the organisation's systems. The authentication process that is the main part of IDM is based on the OpenID Connect protocol⁶. OpenID Connect consists of an identity layer on top of the OAuth 2.0 framework⁷, which enables clients to perform identity verification, based on the authentication performed by an authorization server. OpenID Connect exploits a JSON/REST-based identity built-in functionality, alongside with JSON Web Tokens (JWT)⁸. Moreover, some basic profile information is obtained about the identified person in an interoperable REST-like manner. The principal notion of OpenID Connect is to create a lightweight API, providing seamless authentication and authorisation in applications. Finally, identity governance, i.e. the policies and processes that guide the administration of user roles and access in an organisational environment, is also an important part of IDM.

3.4.1.1.2. Role based access management

Access management (AM) is the process of identifying, tracking, controlling and managing authorised or specified users' access to a system, application or any IT instance. It is a broad concept that encompasses all policies, processes, methodologies and tools to maintain access privileges within an IT environment.

Role-based access control (RBAC) refers to the idea of assigning permissions to users based on their role within an organisation⁹. Typically, AM is used in conjunction with identity management (IM). Identity manager creates, provisions and controls different users, roles, groups and polices, whereas AM ensures that these roles and policies are followed. An example of RBAC is depicted in Figure 30.

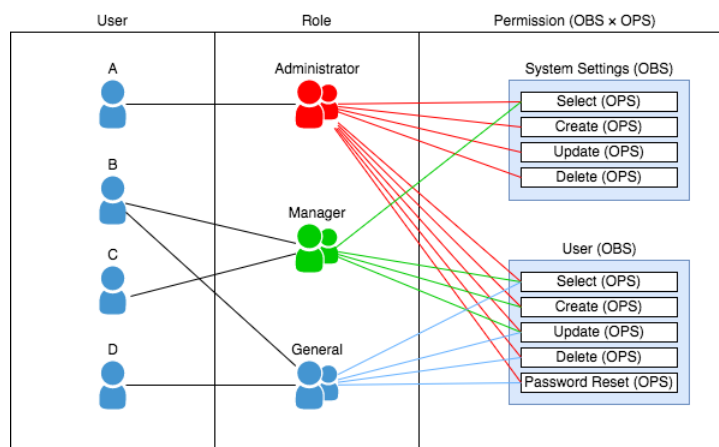


Figure 30. RBAC representation¹⁰

Figure 31 shows how RBAC works. An already authenticated user (through the IM), has a role assigned to him. This role, in turn, allows him/her to perform some limited operations and thus access some objects of the continuum. The operations that this user can perform and the objects that this user can access are described in the permissions of the role to which the user belongs.

⁶ <https://openid.net/developers/how-connect-works/>

⁷ <https://datatracker.ietf.org/doc/html/rfc6749>

⁸ <https://tools.ietf.org/html/rfc7519>

⁹ <https://auth0.com/docs/authorization/concepts/rbac>

¹⁰ <https://dsonoda.medium.com/role-based-access-control-overview-257de64534c>

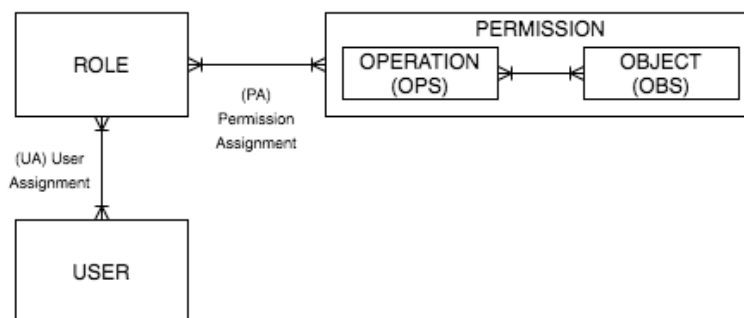


Figure 31: RBAC elements¹¹

3.4.1.2. Secure API Gateway compliant to aerOS security

An API Gateway is a critical component in any deployment that acts as an intermediary between the client applications and the backend services. It works as an entry point for all external requests and provides an invaluable service of both security and authentication, ensuring no unauthenticated user can access the backend services or data. An API Gateway is a necessary component for aerOS. This is due to the framework it offers, simplifying the process of exposing multiple APIs with a unified interface, while offering advanced features to enhance API management and performance. Without it, the endpoints are left without a way to verify any user access to any of the API's or services. The API Gateway represents an essential part of the architecture.

In order for an API Gateway to be desirable, it must have the following features:

- (i) **API Aggregation:** The API Gateway allows the aggregation and combining multiple backend services into a single API endpoint. It acts as a gateway, consolidating the functionality of various services and presenting them through a single-entry point. This simplifies client access and reduces the complexity of managing multiple APIs.
- (ii) **Load Balancing and Service Discovery:** The API Gateway helps distribute incoming requests across multiple backend services, ensuring optimal performance and high availability.
- (iii) **Caching:** With built-in caching capabilities, The API Gateway reduces the load on backend services by storing frequently accessed API responses.
- (iv) **Security and Authentication:** The API Gateway offers various security features to protect APIs. It supports authentication mechanisms such as API keys, JWT, OAuth2, and OpenID Connect. With these tools rate limiting, request throttling, and IP allowlist/blocklist can be enforced to prevent unauthorized access and ensure API security.
- (v) **Middleware Support:** The API Gateway provides a rich set of middleware options to modify and process requests and responses. Custom middleware functions can be added to implement functionality like request/response transformation, request validation, logging, metrics and error handling.

3.4.2. Related requirements

- TR-17: Cross-layer cybersecurity
- TR-64: Cybersecurity tools
- TR-65: Privacy-preserving functions
- TR-67: Cybersecurity policies

¹¹ <https://dsonoda.medium.com/role-based-access-control-overview-257de64534c>

3.4.3. Structure diagram

Figure 32 illustrates the structure diagram of the Communication and Service APIs, including the relevant components that will be considered.

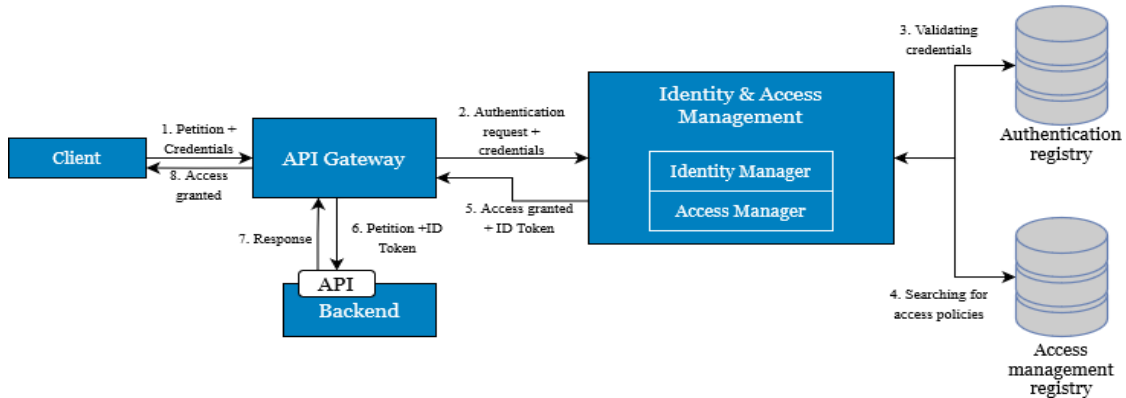


Figure 32: API Gateway architecture

Such components are identified and further described in Table 8:

Table 8: Identity and Access Management and Secure API Gateway components

Component	Description	Interactions
Identity and Access Management	Responsible for implementing the authentication and authorization of aerOS users.	KrakenD API to send the authentication tokens
Authentication registry	The place where the policies are stored	Communicates with IAM to validate the credentials
Access management registry	The place where the access policies are stored	Communicates with IAM to identify the appropriate access policy
Client	Any deployed element within the aerOS continuum that wants to access a protected endpoint.	The element obtains a token from Keycloak, then it makes the petition to the backend API with said token.
API Gateway	The proposed API Gateway.	Receives petitions from the client, verifies the token with the Identity Manager and if it is valid, it redirects the petition to the backend.
API	The API endpoint where the petition is being sent.	Performs the functionality specified.
Backend	The backend where the target resource is available in the continuum.	

3.4.4. Candidate technologies and standards

Table 9: Identity and Access Management and Secure API Gateway candidate technologies

Technology/Standard	Description	Justification
Keycloak	Open-source IAM that support a plethora of technologies and protocols for authentication and	Keycloak is one of the well-known open-source solutions for IAM that is user-friendly and can be easily parameterized.

	authorization.	
OpenID Connect	Token-based authentication protocol.	It is a state-of-the-art and secure authentication protocol that is based on the OAuth 2.0 standard.
KrakenD	KrakenD is a stateless, distributed, high-performance API Gateway.	KrakenD fulfils all the requirements and is the most likely option for the API Gateway.

3.5. Node's self and monitoring tools

In the aerOS computing continuum, there is a variety of IE types depending, for example, on their hardware, software, computing capabilities or location in the continuum. This means that one of the main characteristics of aerOS is the heterogeneity of the nodes that compose it. One of the objectives of aerOS is to provide these nodes with autonomy of use, so that their operations do not depend on human interactions. In this way, IEs are becoming capable of executing operations autonomously and constantly monitoring their state of health and/or other parameters individually.

This section describes the features that IEs, in aerOS, will have in order to be self-capable of performing certain actions. As it has been explained, IEs are the granular entities in aerOS that are able to withhold workload execution in the continuum. As such, each IE holds a set of capabilities and features (GPU, total resources, current consumption, domain to which they belong, etc) as described in the scope of T4.1. In addition, IEs can be considered single entities that can perform their own software functionalities to improve/modify/report their state towards the continuum. Maximising the concept of decentralization, augmenting the capacities of self-capabilities in IEs will make the continuum more reliable and resilient to overall network/services downfalls.

In this context, the sub-sections below expose the self-capabilities (or self-features) that have been designed to be included in the IEs of aerOS. This involves monitoring (exposing such parameters to the whole continuum) and inner actions to improve/modify the state of each IE. The content is organised in the same way as in the other subsections of Section 3, directly mapping to the evolution and works performed in Task T3.5 of aerOS.

3.5.1. Research lines

As previously mentioned, one of the objectives of aerOS is to allow the IEs that connect to the continuum to be autonomous. For this, it is necessary that they meet certain characteristics that make them independent. These features are offered through the set of self-* capabilities that aerOS offers to every IE connected to the continuum, regardless of the heterogeneity that characterises it.

In order to better structure those capabilities, it was decided to analyse narrow functionalities that could be separated. The capabilities that IEs must meet can be listed as follows:

- **Self-awareness:** This self-capability, considered as the basis of any autonomous and independent system, aims to observe, analyse and evaluate the environment that surrounds a node and itself, permanently monitoring its own state of health. In addition, it must be able to see beyond its immediate environment, in order to compose a broader spatial vision of the entire environment that surrounds it. The purpose of these analyses is to modify the behaviour of the IE based on the results obtained from the observations made. This self-capability is responsible for providing information to the self-diagnose, self-orchestration, and self-optimization and adaptation components.
- **Self-orchestration:** self-capability of the intelligent nodes (IEs) to collaborate in the management and coordination of their own workloads, in order to achieve common objectives to the rest of the IEs. This self-capability allows each IE to align with the aerOS global orchestration directives and is able of improving the scalability of applications, reducing the number of failures that occur during their execution.
- **Self-diagnose:** capability of an IE to assess its own state of health through constant monitoring and evaluation of data such as CPU and RAM usage, storage unit usage and status, network connection

metrics or data received through the self-awareness or self-security components. These data are translated into a normalised composite indicator adjusted to a scale of pre-established values.

- **Self-security:** it enables malfunctions and vulnerabilities to be detected at the node level to ensure correct operation or secure incorporation into the computing continuum. It does this by monitoring the network traffic of the IE's network interfaces, checking against updated threat databases or applying rules to detect attacks. The attack information is sent to the self-diagnose component.
- **Self-API:** global API installed in each of the IEs that exposes the operations allowed on the rest of the self-* capabilities installed and controls the flow of data that enters and leaves the node within the continuum.
- **Self-healing:** it crystallises the capability of autonomously recovering affected parts of the system both at the hardware and software level caused by failures or abnormal states. It can also restart the system to pre-established routines scheduling, if necessary.
- **Self-scaling:** self-capability of the IEs that can run K8s to horizontally increase or decrease the hardware resources dedicated to the workloads depending on the needs of each one. This modification runs in real-time and is based on time series inference and custom logic.
- **Self-configuration:** capacity of an intelligent node of the continuum to automatically (re)configure itself at any time, under any situation and condition, maintaining the configurations established by the users or administrators in the event of any type of failure or error at the hardware or software level. The system administrator can apply high-level policies during the (re)configuration process.
- **Self-optimisation and adaptation:** it allows to constantly improve the efficiency and performance of an IE, modifying its behaviour in real-time to adapt to changes in the environment that surrounds it. The main objective of this self-capability is to reduce the power consumption of the nodes and the volume of data generated through advanced AI techniques.
- **Self-realtime:** it allows to have knowledge of the time behaviour's of the containers in execution in real-time. In this way, temporary isolation between real-time and non-real-time components can be ensured. In case of performance decrease, it can send warnings to the self-orchestration component.

More details on how they relate among themselves, and the way that they are conceived to be materialised, are depicted in the following sub-sections

3.5.2. Related requirements

The various self-features described in this section will play a role across the Meta-OS installation and validation. In particular, they will be created to satisfy specific technical requirements that have been expressed as needed for the overall functioning of aerOS. In addition, the software suite of features will be tested in various pilots of the projects. The particular list of related technical requirements and pilot scenarios where the self-capabilities and functionalities will be tested is as follows:

- TR-15 Self-* mechanisms
- TR-20 Resource availability
- TR-22 Context awareness
- TR-26 aerOS Infrastructure Monitoring
- TR-27 Infrastructure management automation
- TR-30 Optimized configuration on IoT Devices
- TR-32 aerOS network monitoring
- TR-54 Cybersecurity tools

3.5.3. Structure diagram

As described above, the software suite of self-capabilities in aerOS is organised in ten specific features that will, in practice, result in different software to be run by the IEs. Particularly, it has been designed for each self-

feature (self-awareness, self-orchestration, etc.) to have its own responsibility and role and act as a single entity within an IE. However, the overall suite must work together to deliver the expected behaviour, thus creating intertwined relations among them. There, certain features might need the existence of others to fully provide their functionality. Similarly, others would need to rely on each other to avoid overlaps and/or duplicities.

Therefore, a diagram has been created in order to represent how the self-features would interact to deliver the functionalities of monitoring and self behaviours in the IEs of aerOS. Figure 33 represents the different components of the global IE’s self and monitoring tools. Self-configuration and self-healing connect directly to the peripheral IoT devices (when those are connected to an IE), keeping track of the connection status and the health of such devices (battery, data transmission, desired configuration – e.g., bitrate, bauds...). These two modules will contribute to “health status” reporting of the IE, whenever related to IoT devices. Health status, reported as “health score,” is calculated and managed by another self-feature (self-diagnose), which will be a centrepiece of the whole suite. The score (which is a composite index) will be constructed by the input of other features, such as self-realtimeness, that will analyse whether those IEs holding real-time services are meeting the expected performance behaviour. It will also gather information from the self-awareness feature that will host the main monitoring engine inside an IE. By specifying metrics to be collected, it will be the basis for feeding both other features and the global exposure of an IE to the outside. Such exposure is materialised via self-API (gateway of APIs), that will include the required security according to the guidelines and tool decided globally for aerOS (Task T3.4). In addition, self-security is extended in the inner side of the IE by establishing traffic inspection mechanisms to detect potential vulnerabilities at IE level. Although IEs are conceived (in this part of the Meta-OS) to be sort of independent elements, they live within a continuum and their real “action” (in terms of workload execution) will depend on the orchestration decisions taken along the continuum. Therefore, they should have a way of interacting back with such orchestration. In this sense, the self-orchestration feature comes into play, being fed by various elements, as previously exposed, in order to understand whether the IE should trigger alarms or requests for re-orchestration due to state of resources. Current state of resources will come from self-awareness while future/predicted use of resources will come from self-optimisation and self-adaptation feature. Lastly, the IEs should have the capacity to horizontally scale the replicas of their resources (if they fall under the K8s category), and that is covered by the self-scaling feature.

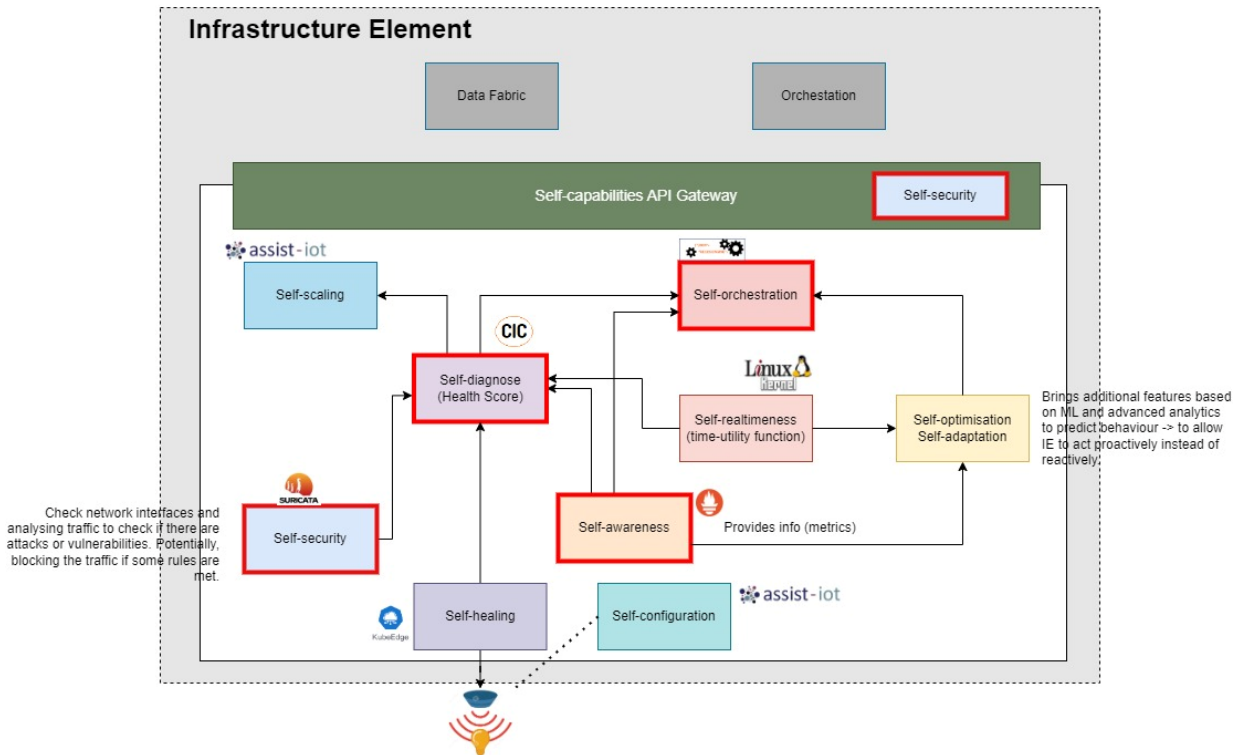


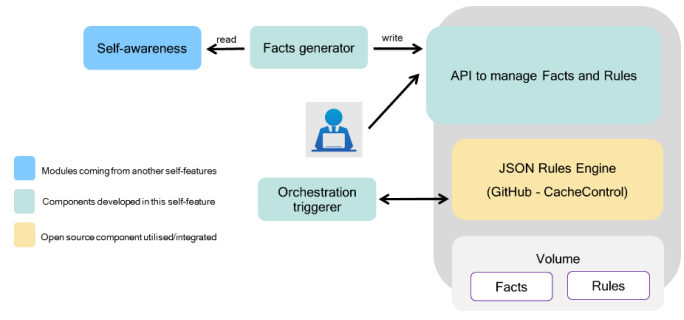
Figure 33. Interaction diagram of self-features in IEs of aerOS

As observed in the figure, five out of the ten self-features stand out red-squared. The goal is to highlight the fact that those are the only mandatory features **in all IEs of a continuum**. Self-realtimeness will only be present in

those cases managing real-time containers, self-healing and configuration will only live in those IEs with peripheral IoT devices, self-scaling will only be used (on voluntary basis) if the IE is a K8s worker node and self-optimisation and adaptation will be an optional plugin to the IEs. In Table 10, the specific functionalities, details and interactions are further described.

Table 10. Self-capabilities components, description and interactions

Component	Description	Interactions
Self-awareness	<p>This is the feature that allows the monitoring of values of the state of the IE. Per definition, it materialises the ability of computer systems to observe and analyse the environment that surrounds them, with the aim of making changes in their behaviour according to the observations made. Specific relevant aspects of the self-feature as it is designed in M12:</p> <ul style="list-style-type: none"> • Gather parameters such as CPU, RAM, network usage, etc. • Custom parameters can be defined. • Need to adapt to various IE flavours (e.g., using Prometheus in K8s node). • Can provide information about the services running in the IE. • Must be able to inform about energy consumption in “real time”. 	<p>It gathers info of the IE and directly feeds the self-diagnose, self-orchestration and self-optimisation and adaptation. It also provides context to the associated Context Broker to that IE:</p>
Self-diagnose	<p>The element that computes the health score, a value that represents the health value of an IE at every specific instance. Per definition, the feature assessing the device’s state of health. It collects and analyses data from multiple sources of information such as memory usage, CPU usage, or network connection metrics, providing a health score. It will construct a composite indicator (normalised, weighted and aggregated according to decisions in the task.</p>	<p>It will be a crucial element, feeding the self-scaling and self-orchestration features. It will be fed by self-awareness, self-security, self-healing and self-realtimeness, that will contribute to construct the health of the IE.</p>
Self-orchestration	<p>It is the feature that allows the interaction with the global orchestration directives of aerOS. Per definition, the implementation of potential ways to manage workloads within IEs. As illustrated in Figure 34, some aspects of the self-feature, as designed in M12, include:</p> <ul style="list-style-type: none"> • <i>Rule engine</i> that contains the parameters and their thresholds upon which an orchestration request / issue must be triggered “upwards”. • <i>Facts generator</i> to translate from current state to real facts that will match to the parameters in the engine. • <i>API</i> to allow dynamic creation/modification of rules. • <i>Orchestration triggerer</i> to properly format the required messages to the global aerOS orchestration. 	<p>It is directly fed by three components: (1) self-awareness (metrics to construct the facts), (2) self-optimisation / adaptation, to understand whether future predictions of the IE state should trigger orchestration actions in advance and (3)</p>

	 <p><i>Figure 34. Inner structure of self-orchestration feature in an IE</i></p>	<p>self-diagnose, to embed health score in the decision process.</p>
<p>Self-security</p>	<p>An additional security layer that deepens into the inner-IE aspects. Per definition, module in charge of giving context of the node in terms of security (not global), ensuring secure addition to the continuum, analysis of traffic and check against updated threat databases.</p> <p>Specific relevant aspects of the self-feature as it is designed in M12:</p> <ul style="list-style-type: none"> • It inspects the IE network interface traffic monitoring. • It will be able to apply (dynamically defined rules) to detect attacks to the IE. • It will handle the initial authentication of the IEs once they enter in the continuum. This will be managed in conjunction with the decisions of the architecture (in D2.6), specifically with the incorporation of domains. 	<p>The information of the attacks will be reported to the Self-Diagnose component</p>
<p>Self-API</p>	<p>A feature to expose the potential management of the different self-capabilities. It will be able to retrieve certain management aspects and will allow parameterisation of e.g., dynamic rules in the self-orchestration software. It consists of the global API of a node (IE).</p> <p>Specific relevant aspects of the self-feature as it is designed in M12:</p> <ul style="list-style-type: none"> • It may take the form of an API Gateway, • It will align with OpenAPI. • It will control the volumes of information that can flow in/out of an IE. 	<p>It will interact with all the rest of self-features in order to manage their configuration / parameters / data.</p>
<p>Self-scaling</p>	<p>The capacity of an IE to adapt to service demand and leverage available techniques to horizontally scale (up or down) the resources devoted to a specific workload (inside that IE) in a dynamic fashion, based on time series inference and custom logic. This is reserved to IEs with K8s flavour.</p>	<p>One of the inputs that this feature might use is the health score, deciding whether or not to rely on the IE state to increase replicas.</p>
<p>Self-realtimeness</p>	<p>Awareness of timing behaviour for real-time critical system to assure temporal isolation between real-time and non-real-time components.</p> <p>Specific relevant aspects of the self-feature as it is designed in M12:</p> <ul style="list-style-type: none"> • It will handle aspects of real-time containers such as deadline misses, number of interrupts, interrupt latency, cache hits/misses... 	<p>Interaction with self-orchestration</p>

	<ul style="list-style-type: none"> • Dynamically issue re-orchestration requests in case of real-time performance degradation 	
<p>Self-configuration</p>	<p>Ensuring that an IE keeps the desired configuration of the IoT devices (parameters, connection...) expressed by the user in case resources go down or something unexpected happens.</p> <p>Specific relevant aspects of the self-feature as it is designed in M12:</p> <ul style="list-style-type: none"> • It works in a “steady-state” fashion. • Enables a system to dynamically adapt to changing environments using policies provided by the administrator • Configuration declares resources needed by a certain functionality • Configuration specifies reactions for specific events • The system communicates with the resources • If a resource becomes limited, then the system decides which functionalities to keep. 	<p>This component will be one of the few that will be able to work in a standalone fashion, not requiring interaction with other features.</p>
<p>Self-healing</p>	<p>Related to IoT devices, capabilities of actively attempting to recover themselves from abnormal states, based on a pre-established routines scheduling.</p> <p>Specific relevant aspects of the self-feature as it is designed in M12 include the following (Figure 35):</p> <ul style="list-style-type: none"> • It works in a “runtime” fashion. • Definition and detection of abnormal states is expected (relying, when possible on KubeEdge device mappers to adjust to communication technologies with devices). • If needed, a translator from communication technology to MQTT protocol should be included. <div data-bbox="438 1272 1212 1444" data-label="Diagram"> <pre> graph TD AS[Abnormal states] --> S[Security] AS --> D[Dependability] AS --> LT[Long-term] S --- S1[- jamming] S --- S2[- DoS] D --- D1[- data corruption] D --- D2[- network protocol violation] LT --- LT1[- hardware EOL] LT --- LT2[- hardware unsupported capabilities] </pre> </div> <p><i>Figure 35. Inner schema of abnormal state detection in self-healing</i></p>	<p>In the case of detecting abnormal states, or healing actions to be taken, this will feed the health score of the IE (interacting with self-diagnose feature).</p>
<p>Self-optimisation /adaptation</p>	<p>Per definition, the self-capability to provide continuous improvement of the performance and efficiency of the IE.</p> <p>Specific relevant aspects of the self-feature as it is designed in M12 include the following:</p> <ul style="list-style-type: none"> • It bases on the concepts of approximative and adaptative sampling techniques, to dynamically adjust the sampling, in aerOS case, “workload orchestration”. • The goal is to reduce the data volume and energy consumption employing those AI techniques. 	<p>It will receive information by self-awareness (monitoring of IE state) and self-realtimeness and will send valuable information to the self-orchestration feature.</p>

3.5.4. Candidate technologies and standards

As per M12, the current list of technologies that are planned to be used/developed/integrated/customized for delivering the self-functionalities indicated above is as follows:

Table 11. Self-features and monitoring candidate technologies

Technology/Standard	Description	Justification
Prometheus (self-awareness)	Open-source monitoring and alerting toolkit.	It is a reliable system. It allows to diagnose problems quickly, does not depend on network storage or other remote services and its configuration is simple and fast.
Monitorix (self-awareness)	Open-source tool that monitors operating system resources and services of a node.	It consumes very low resources, allows to obtain the energy consumption in Intel and AMD processors (AMD64 architectures), can customise the data collection interval and allows to extract the data to a log file.
PowerTOP (self-awareness)	Open-source diagnostic tool that provides energy consumption by host and by process (per PID).	Allows experiment with various GNU/Linux power management configurations, run as a repetitive task, and obtain power consumptions from Intel, AMD, ARM and UltraSPARC processors.
json-rules-engine (self-orchestration)	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.
KubeEdge (self-healing)	Open-source system for extending native containerised application orchestration capabilities to hosts at edge.	It is based on K8s, supports MQTT, allows to create custom logic and communication between IoT devices at the edge.
MQTT (self-healing)	Standard for data transmission between IoT devices.	It requires minimal resources for its operation and is very efficient. Therefore, it is possible to use it on IoT devices at the edge.
Technologies based on RAINBOW EU project (self-adaptation and self-optimisation)	Adaptive sampling techniques and adaptive filtering techniques.	It allows to reduce the volume of data used and the energy consumption of the devices compared to other implementations.
Self-configuration enabler from ASSIST-IoT EU project (self-configuration)	Enabler to keep heterogeneous devices and services in sync with their configurations. The configuration can be updated and backup configurations can be defined in case of error.	The component is capable of reacting to the changing environment and automatically updating the configuration as needed. It is also capable of detecting if alternative configurations should be used.
Resource provisioning enabler from ASSIST-IoT EU project (self-scaling)	This tool can scale horizontally (up or down) the resources of a specific enabler (within an IE) in real-time and depending on time series inference and custom logic.	It is able to store time series with component usage metrics for each active enabler and use deep learning techniques to predict usage metrics and scale out resources dedicated to each enabler component automatically.

Neural prophet (self-scaling)	Framework for interpretable time series forecasting.	Combine neural networks and time series data, and is written in Python.
Predictive Horizontal Pod Autoscaler (self-scaling)	This tool is a Horizontal Pod Autoscaler (HPA) with extra predictive capabilities, allowing to autoscale using statistical models for ahead of time predictions. This is a potential alternative to Resource provisioning enabler from ASSIST-IoT EU project.	It is able to improve scaling results by scaling resources ahead of demand request through early decision making.
Custom aerOS monitor (self-realtimeness)	Tool that monitors the execution behaviour of containers by extending the Linux kernel monitor.	This custom tool is capable of monitoring container metrics, calculating time-utility functions, requesting re-orchestration, etc.
Suricata (self-security)	Is a high performance, open-source network analysis and threat detection software.	It is capable of analysing all the network interfaces of an IE, detecting attacks on the network, reporting attacks to the self-diagnose component and preventing intrusions.
Keycloak (self-security)	Open-source program that allows managing access and identities.	It allows easy configuration, user federation and administration, detailed authorisations, etc.
Express (API)	A minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.	It allows to create robust, secure and lightweight APIs quickly and easily.
Swagger (API)	Open-source toolset for designing, building, documenting and using RESTful web services.	It allows to generate documentation, code and test cases in an automated way.
Custom development	It is expected that many of the self-features will incorporate custom developments to achieve their functionality.	Lightweight languages and code will be used. Best practices coming from DevPrivSecOps will be used.

4. Conclusions and future work

During the first twelve months of the project, aerOS has mostly focused on the research and design phase, which involves the study of the available solutions in the literature, analysis of the requirements, definition of the use cases, and selection of technologies. The successful completion of this phase has resulted in the definition of the initial version of the aerOS architecture and the specification and initial proposal of the aerOS WP3/4 software components. Consequently, this influence is reflected in the work carried out in WP4 during this period, as well as in the results presented in this document. Specifically, D3.1 is centered around the initial distributed compute infrastructure specification and implementation. WP3 aims to provide a set of components to enable a secure, scalable IoT-Edge-Cloud continuum, supporting resources and services orchestration to boost operation of autonomous systems based on the aerOS architecture.

Deliverable D3.1 is part of an iteration of the set of outcomes of WP3. This document is the first version, which is due to M12 of the project and presents work done so far. Each of these deliverables aims at releasing the

outcomes related to the software for delivering intelligence at the edge, which occurs during intermittent time periods. The second iteration of the deliverable is planned to deliver the intermediate release in M18, and the third iteration is planned to deliver the final release in M30. Since this deliverable will have two more subsequent iterations, the specifications included here may be extended or updated as the project evolves. Solutions presented were classified according to WP3 tasks division and described using a template to maintain a common structure of information given. It covers description of the task, main research lines and the advances and decisions already performed in each of them, including diagrams when pertinent and a clear list of candidate technologies for their implementation. In the next iteration of the deliverable, the templates of each solution will be updated to include additional information, such as usage stories. Next version of the deliverable will include solution description template extended with additional information such as usage stories. In addition, the next iteration (D3.2) will already include actual software as first outcomes of WP3 tasks.

Regarding the future work, the finalization of this deliverable will boost the already initiated development of related software. Next tasks will be to:

- Fill in missing information in the design of specific components (e.g., communication interfaces),
- Conduct any necessary adjustments (e.g., slight modification in provided functionalities, change in structure, refinement of selected technologies),
- Establish needs for interactions between WP4 and other WP3 components,
- Preparation of the backlog of tasks, distribution of work and kick-off of implementation activities,
- First software results (MVP) ready and available by M18 (however, the degree of development is foreseen not be homogeneous for all the components).

Finally, during WP5 execution in the upcoming months, the main goal will be to deploy the use cases and scenarios to validate the aerOS system as a whole, including the solutions proposed in WP3 and WP4.

References

- [1] T. Z. Benmerar, T. Theodoropoulos, D. Fevreiro, L. Rosa, J. Rodrigues, T. Taleb, P. Barone, K. Tserpes and L. Cordeiro, “Intelligent Multi-Domain Edge Orchestration for Highly Distributed Immersive Services: An Immersive Virtual Touring Use Case,” in *iEDGE 2023 - IEEE Symposium on Intelligent Edge Computing and Communications*, 2023.
- [2] J. Dobies and J. Wood, *Kubernetes operators: Automating the container orchestration platform*, O'Reilly Media, 2020.
- [3] operatorframework.io, “Welcome to operator pattern framework,” 2023. [Online]. Available: <https://operatorframework.io/operator-capabilities/>. [Accessed 11 07 2023].
- [4] R. Duan, F. Zhang and S. U. Khan, “A Case Study on Five Maturity Levels of A Kubernetes Operator,” in *2021 IEEE Cloud Summit (Cloud Summit)*, 2021.
- [5] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp and J. Soldani, “The essential deployment metamodel: a systematic review of deployment automation technologies,” *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, pp. 63--75, 2020.
- [6] J. Wettinger, U. Breitenbücher and F. Leymann, “Standards-based DevOps automation and integration using TOSCA,” in *2014 IEEE/AOCM 7th International Conference on Utility and Cloud Computing*, 2014.
- [7] A. Zerouali, R. Opdebeeck and C. De Roover, “Helm Charts for Kubernetes Applications: Evolution, Outdatedness and Security Risks,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023.
- [8] L. Mamushiane, A. A. Lysko, T. Mukute, J. Mwangama and Z. Du Toit, “Overview of 9 open-source resource orchestrating ETSI MANO compliant implementations: A brief survey,” in *2019 IEEE 2nd Wireless Africa Conference (WAC)*, 2019.
- [9] R. Botez, A.-G. Pasca and V. Dobrota, “Kubernetes-Based Network Functions Orchestration for 5G Core Networks with Open Source MANO,” in *2022 International Symposium on Electronics and Telecommunications (ISETC)*, 2022.
- [10] C. Contu, A. Ciobanu, E. Borcoci, M.-C. Vochin, I. A. Balapuwaduge, S. Topoloi and R.-F. Trifan, “Deploying Use Case Specific Network Slices Using An OSM Automation Platform,” in *2022 25th International Symposium on Wireless Personal Multimedia Communications (WPMC)*, 2022.
- [11] ETSI, “OSM Usage,” 2023. [Online]. Available: <https://osm.etsi.org/docs/user-guide/latest/05-osm-usage.html>. [Accessed 20 07 2023].
- [12] ETSI, “KNF Onboarding Walkthrough (Work in Progress),” 2023. [Online]. Available: <https://osm.etsi.org/docs/vnf-onboarding-guidelines/07-knfwalkthrough.html>. [Accessed 20 07 2023].
- [13] ETSI, “Day 1: VNF Services Initialization,” 2023. [Online]. Available: https://osm.etsi.org/gitlab/vnf-onboarding/osm-packages/-/tree/master/simple_ee_vnf/. [Accessed 20 07 2023].
- [14] ETSI, “simple_ee_vnf example,” 2023. [Online]. Available: https://osm.etsi.org/gitlab/vnf-onboarding/osm-packages/-/tree/master/simple_ee_vnf/. [Accessed 20 07 2023].