NEMO

Next Generation Meta Operating System

D3.1 Introducing NEMO Kernel

Document Identification			
Status	Final	Due Date	31/10/2023
Version	1.0	Submission Date	03/11/2023

Related WP	WP3	Document Reference	D3.1
Related Deliverable(s)	D1.1, D1.2, D2.1	Dissemination Level (*)	PU
Lead Participant	RWTH	Lead Author	Jonathan Klimt (RWTH)
Contributors	RWTH, ATOS, INTRA, STS, SYN, ENG, ICCS, WIND, TID	Reviewers	Nikos Drosos (SPACE)
			Aitor Alcázar-Fernández (ATOS)

Keywords:

NEMO, Kernel Space, Secure Execution Environment, Privacy, Policy Enforcement Framework, Cybersecurity, Authentication, Access Control, meta-Orchestrator, IoT, Edge, Cloud Continuum.

Disclaimer for Deliverables with dissemination level PUBLIC

This document is issued within the frame and for the purpose of the NEMO project. This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101070118. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

The dissemination of this document reflects only the author's view, and the European Commission is not responsible for any use that may be made of the information it contains. This deliverable is subject to final acceptance by the European Commission.

This document and its content are the property of the NEMO Consortium. The content of all or parts of this document can be used and distributed provided that the NEMO project and the document are properly referenced.

Each NEMO Partner may use this document in conformity with the NEMO Consortium Grant Agreement provisions.

(*) Dissemination level: **(PU)** Public, fully open, e.g., web (Deliverables flagged as public will be automatically published in CORDIS project's page). **(SEN)** Sensitive, limited under the conditions of the Grant Agreement. **(Classified EU-R)** EU RESTRICTED under the Commission Decision No2015/444. **(Classified EU-C)** EU CONFIDENTIAL under the Commission Decision No2015/444. **(Classified EU-S)** EU SECRET under the Commission Decision No2015/444.

Document Information

List of Contributors	
Jonathan Klimt Stefan Lankes	RWTH
Aitor Alcázar-Fernández Ignacio Prusiel Mariscal Enric Pages Montanera	ATOS
Dimitris Siakavaras	ICCS
Dimitrios Skias	INTRA
Nicolas Peiffer	TSG
Gianluca Rizzi	WIND
Terpsi Velivassaki Harry Skianis	SYN
Norredine Abouriche Vassilis Bouras Sofia Giannakidou Yannis Papaefstathiou	STS
Hugo Ramon Pascual Luis M. Contreras Murillo Alejandro Muñiz Da Costa	TID

Document History			
Version	Date	Change editors	Changes
0.1	31/05/2023	Jonathan Klimt (RWTH)	Initial version, TOC creation
0.2	21/06/2023	Aitor Alcázar-Fernández (ATOS)	Document minor refinements, first contributions
0.3	11/07/2023	Jonathan Klimt (RWTH)	Removed Development chapter and reworked ToC
	6/9/2023	Dimitris Siakavaras (ICCS)	Added first draft of ICCS contribution in 2.1.3.3
0.4	12/10/2023	Jonathan Klimt (RWTH)	Updated formatting and contents to new version
0.5	16/10/2023	Aitor Alcázar-Fernández (ATOS)	Last contributions for T3.4
	16/10/2023	Norredine Abouriche, Vassilis Bouras, Sofia Giannakidou (STS)	Updated and final version of Security Modules
	17/10/2023	Terpsi Velivassaki, Harry Skianis (SYN)	Contribution regarding the NEMO Access Control component
0.6	18/10/2023	Jonathan Klimt (RWTH)	Added missing parts of introduction, conclusion, and T3.1 related content. Split chapter 2 in 4 separate chapters. Integration of different 0.5 contributions. Formatting rework.

0.7	25/10/2023	Aitor Alcázar-Fernández (ATOS)	Peer review
0.8	25/10/2023	Nikos Drosos (SPACE)	Peer review
0.9	03/11/2023	Jonathan Klimt (RWTH), Dimitrios Skias (INTRA), Yannis Papaefstathiou (STS), Terpsi Velivassaki (SYN), Nicolas Pfeiffer (TSG)	FINAL VERSION
1.0	03/11/2023	Rosana Valle (ATOS)	Quality check and submission to EC.

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Jonathan Klimt (RWTH)	03/11/2023
Quality manager	Rosana Valle Soriano (ATOS)	03/11/2023
Project Coordinator	Enric Pages Montanera (ATOS)	03/11/2023
Technical Manager	Harry Skianis (SYN)	03/11/2023

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
AAA	Authentication, Authorization, and Accounting
API	Application Programming Interface
CFDRL	Cybersecure Federated Deep Reinforcement Learning
CMDT	Cybersecure Micro services Digital Twin
CNAPP	Cloud-Native Application Protection Platforms
CNCF	Cloud-Native Computing Foundation
CNI	Container Network Interface
CoCo	Confidential Containers
CRD	Custom Resource Definition (Kubernetes)
CSV	Comma Separated Values
DAST	Dynamic Application Security Testing
Dx.y	Deliverable number y belonging to WP x
EC	European Commission
eBPF	(extended) Berkeley Packet Filter.
HTTP	Hypertext Transfer Protocol
IAST	Interactive Application Security Testing
IBMC	Intent-Based Migration Controller
IMS	Identity Management System
IoT	Internet of Things
JSON	JavaScript Object Notation
K8s	Kubernetes
meta-OS	Meta-Operating System
ML	Machine Learning
mNCC	meta–Network Cluster Controller
MO	meta-Orchestrator
MOCA	Monetization and Consensus-based Accountability
NAC	NEMO Access Control
OCI	Open Container Initiative
OSS	Open-Source Software
PaaS	Platform as a Service
PA-LCM	Plugin & Apps Lifecycle Manager
PPEF	PRESS and Policies Framework
PRESS	Privacy, data pRotection, Ethics, Security & Societal
QoS	Quality of Service
RASP	Run-time Application Security Protection
REST	Representational State Transfer
SAST	Static Application Security Testing
SIEM	Security Information and Event Management
SEE	Secure Execution Environment
SLA	Service Level Agreement

TEE	Trusted Execution Environment
URI	Uniform Resource Identifier
VM	Virtual Machine
WP	Work Package
XML	eXtensible Markup Language

Executive Summary

This deliverable presents a comprehensive overview of the significant progress made in Work Package 3 - NEMO Kernel Space, providing a holistic perspective on the advancements within the NEMO ecosystem. It reflects the project's unwavering commitment to creating a secure, efficient, and seamlessly integrated operating framework to meet the evolving demands of contemporary computing environments.

As the foundational deliverable within this topic, this report focuses on elucidating the fundamental components, architecture, and the intricate design of the solutions being developed. By emphasizing the critical elements of the NEMO Kernel, it sets the stage for the subsequent deliverables, illustrating the meticulous groundwork and in-depth research that underpin the project's trajectory.

- *Task 3.1 Micro-services Secure Execution Environment (SEE)*: In this Task, a set of Kubernetes enhancements are developed to form the “Secure Execution Environment”. These are a runtime for the highly isolated Unikernel technology, a runtime for trusted execution environments, and the extension of the migration capabilities of pods. So far, the background research and solution architecture have been completed and a prototype for the Unikernel runtime exists. The next step is the development of all components and the validation and integration in the NEMO Kernel.
- *Task 3.2 PRESS, Safety & Policy enforcement framework*: The meticulous deployment of the Privacy and Policy Enforcement Framework (PPEF) ensures the ethical and secure handling of NEMO-hosted services. Its alignment with GDPR principles and adherence to the insights from prior deliverables underscores the project's commitment to maintaining the highest standards of data protection and user privacy.
- *Task 3.3 Cybersecurity & Digital Identity attestation*: This Task focuses on cybersecurity aspects including as authentication and access control with all NEMO services, as well as operating system monitoring and network monitoring and network management and encryption thanks to cutting edge kernel monitoring probes.
- *Task 3.4 NEMO meta-Orchestrator*: The NEMO meta-Orchestrator stands as a pivotal solution for the challenges within the IoT to Edge to Cloud Continuum. Guided by intelligence, it optimizes workflow management, emphasizing interoperability for seamless integration with diverse systems. Currently, it boasts a functional Orchestration Engine and Integration Component, with ongoing efforts toward the implementation of the first integration Proof of Concept. The next phase involves exploring solutions for additional component implementation, aiming to integrate a more mature version into the NEMO Kernel Space beta version.

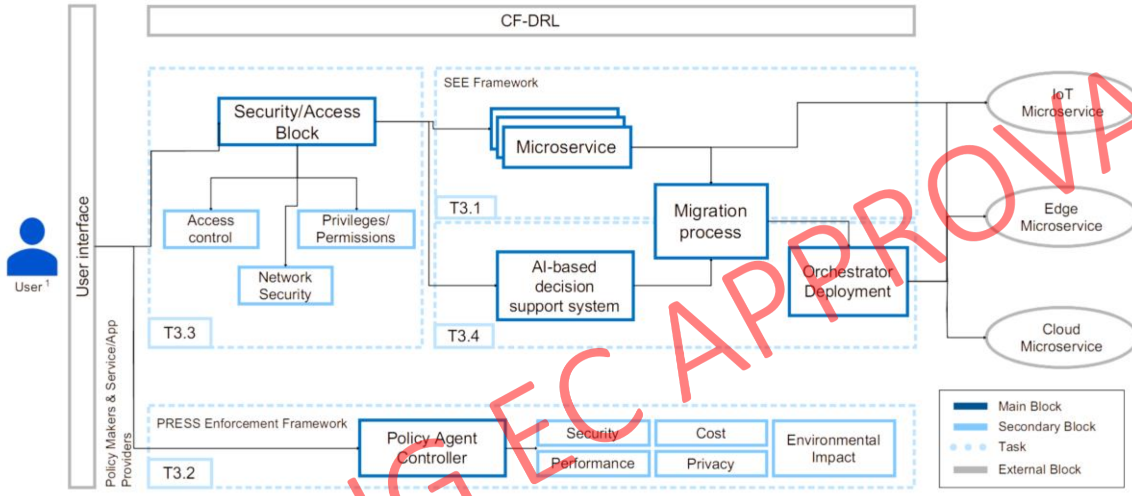
The successful integration and meticulous validation of all the intricate components within the NEMO meta-operating system reflect the project's comprehensive and cohesive approach to developing a sophisticated and resilient infrastructure. This integrated approach, meticulously crafted with an acute attention to detail, serves as a testament to the project's unwavering commitment to achieving excellence in every facet of the NEMO ecosystem. By ensuring that every element operates seamlessly within the larger framework, the project not only establishes a robust and efficient operating model but also sets a precedent for future developments in the field.

The groundwork laid in this deliverable sets a solid foundation for the future advancements that will be detailed in the forthcoming deliverable, D3.2. With the project's momentum steadily gaining traction and the foundations firmly established, the forthcoming deliverable is poised to provide deeper insights into the evolving intricacies of the NEMO ecosystem, further solidifying its position as a trailblazer in the realm of advanced solutions.

1 Introduction

In the Next Generation Meta Operating System (NEMO), the NEMO Kernel technologies are essential in providing the necessary infrastructure for higher level logic functionalities to fully realize the vision of a next generation IoT-Edge-Cloud continuum. This document is the initial report of a series of three reporting on the development progress of these technologies.

The work on the NEMO Kernel is structured in four main groups, which are resembled in the structure of the Workpackage. These are the work on a *Secure Execution Environment for Microservices* (SEE) (T3.1), the work on a *PRESS, Safety & Policy enforcement framework (T3.2)*, multiple components focussing on *Cybersecurity & Digital Identity attestation* (T3.3) and the development on the *NEMO meta-Orchestrator* (T3.4). Figure 1 illustrates these components and their interaction – a detailed description on their functionality and interaction is given in the following chapters. It can be seen that the NEMO Kernel covers numerous technologies from the user to the running services, providing the base layer for the next generation cloud services.



1. User / Smart IoT / Near Edge / Far Edge / National Cloud /Federated Cloud Continuum

Figure 1: Nemo Kernel Space

1.1 Purpose of the document

This document reports the progress of the four tasks that comprise the work package three (WP3). Its aim is to give a reader the necessary background to understand the individual tasks, as well as describing the ideas and plans for the individual solutions. Moreover, the links between the components in the NEMO Kernel space are described, as well as the relationship to components and solutions from other work packages. As it contains plenty of background information, this document targets all readers that have a basic understanding of cloud computing and service-oriented architectures.

1.2 Relation to other project work

The current document, D3.1, marks a crucial step in the development and implementation of the NEMO project. Its significance is best understood within the context of the broader project landscape, particularly in relation to the works outlined in D2.1 and other project deliverables.

The comprehensive analysis provided in D1.1 [1] serves as the foundational groundwork for the establishment of the ethical, privacy, and GDPR compliance frameworks governing the Living Labs. D3.1 builds upon this groundwork by further delving into the technical aspects of the NEMO Kernel components, thus complementing the broader understanding of the project's ethical and technical landscape.

The initial version outlined in D1.2 [2] sets the technical specifications and the architecture's fundamental design principles. D3.1 aligns with D1.2 by delving deeper into the specification and initial results of the SEE, PPEF, Cybersecure modules, and meta-Orchestrator, providing an extended perspective on the meta-architecture and component specifics.

Concurrently submitted with D3.1, D2.1 shares a similar focus on the technical aspects of the project, detailing the advances of CMDT, CF-DRL, and mNCC. D3.1 further contributes by reporting on the NEMO Kernel components specification and initial results, forming a cohesive narrative of the technical progress made in the project's development.

The seamless interplay between D3.1 and these interconnected deliverables underscores the holistic approach taken in the development of the NEMO project, emphasizing the integration of ethical considerations and practical implementation, all essential in achieving the project's overarching objectives.

This integrated approach enables the project to remain in alignment with the project roadmap and the broader objectives set forth by the European Commission, ensuring a comprehensive and coherent implementation of the NEMO initiative.

1.3 Structure of the document

This document mostly follows the structure of the work package. It is divided in four chapters, each representing one task: Chapter 2 is about the Micro-services Secure Execution Environment; Chapter 3 presents the work on the PRESS, Safety and Policy enforcement framework; The work on cybersecurity and digital identity attestation is presented in chapter 4; and chapter 5 is all about the NEMO meta-Orchestrator. Additionally, chapter 6 is dedicated to the proof of concept of the technologies. A conclusion section summarizes the major points of the deliverable.

2 Micro-services Secure Execution Environment

Enhancing security and isolation in modern cloud environments is paramount as it directly addresses the escalating cybersecurity challenges prevalent in today's digital environment. Strong security not only safeguards sensitive data but also bolsters the integrity of applications, assuring users of their privacy and data protection. Developing a seamless solution for the migration of pods between different instances of a cluster is imperative in addressing latency challenges in heterogeneous and distributed setups. Existing setups encounter delays in data transfer, adversely affecting user experience.

In this section we present a solution for the creation of a secure execution environment for modern security critical and dynamic services, the *Secure Execution Environment* (SEE)

2.1 Overview

The Secure Execution Environment (SEE) is the first component in the NEMO Kernel Space. It is a set of enhancements and modifications for the common orchestration engine Kubernetes. We add two components for enhancing the isolation and integrity of the microservices and one that enhances the migration capabilities of Kubernetes.

The need for enhanced security was already introduced in the previous section - enabling the seamless migration of pods between different instances of a cluster is essential in heterogeneous and distributed setups, to reduce latency. By facilitating dynamic pod migration, applications can dynamically move closer to the end-users, minimizing data transfer delays and significantly enhancing user experience.

In NEMO, the SEE provides one of the basic layers for the execution of the components by executing the necessary services and providing control capabilities for the meta-operating system's higher-level logic.

2.2 Background

To understand the concept behind the Secure Execution Environment, it is important to have a base knowledge of the underlying technologies. Therefore, the following sections give an introduction into Unikernels, Kubernetes and Trusted Execution Environments.

2.2.1 Unikernels

In comparison to container technologies, using virtual machines provides strong isolation but comes with increased overhead. Figure 2 illustrates this problem: The VM also contains a full operating system and often several general-purpose services of which some are necessary for the application, and some are not.

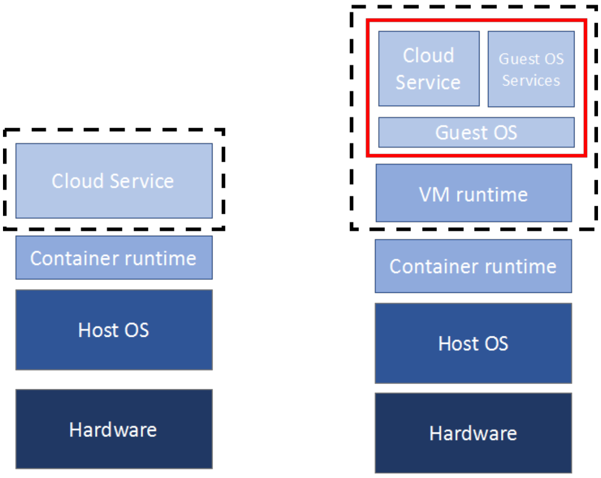


Figure 2: Containerization vs Virtualization

To employ such virtualisation technologies effectively in cloud environments, where this overhead scales linearly with the number of deployments, it's crucial to minimize this overhead. One step is the adoption of microVMs. Instead of creating full-fledged virtual machines that emulate entire computers to run traditional general-purpose operating systems, these microVMs are streamlined to include only the necessary components for running specific applications efficiently. Solutions like Solo5[1], Firecraker[2], uhyve[3], and Qemu's microVM machine type[4] are already established and can significantly reduce memory usage and boot time for such services.

The logical second step is the reduction of the software overhead introduced by the operating system itself. In today's cloud deployments, we can see a shift towards specialized single application service, often focused on handling network requests. In this context, using a traditional multiprocessing, multitasking, multi-user operating system like Linux as a guest OS introduces unnecessary overhead. Library Operating Systems, also known as Unikernels, present an attractive alternative to mitigate this overhead. The core concept is linking the kernel directly as a library to the application, effectively transforming it into a bootable application bundle. This results in a single-address-space machine image containing only the essential code for the application, thereby reducing memory usage and boot time. Furthermore, the entire software stack, from the kernel to the IP stack to the application itself, can be thoroughly analysed and optimized using established compiler techniques. This not only improves performance but also reduces the attack surface of the application, enhancing security. Figure 3 illustrates this architecture.

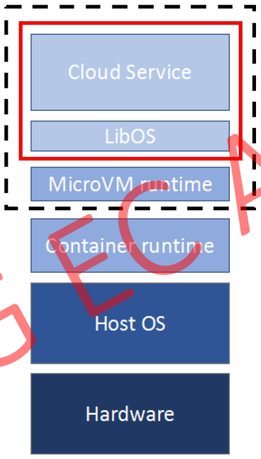


Figure 3: Unikernel Technology Stack

2.2.2 Kubernetes

Kubernetes[5] (K8s) is a powerful open-source container orchestration platform that plays a pivotal role in modern cloud computing. Its primary importance lies in its ability to automate the deployment, scaling, management, and orchestration of containerized applications. Kubernetes provides a robust and standardized framework for container management, allowing developers and operations teams to abstract away the complexities of infrastructure management and focus on application development. By offering features like automatic load balancing, self-healing, and seamless rolling updates, Kubernetes ensures the efficient utilization of cloud resources, enhances application reliability, and enables the seamless scaling of applications in response to varying workloads. This not only simplifies cloud

[1] https://github.com/Solo5/solo5
[2] https://firecracker-microvm.github.io/
[3] https://github.com/hermit-os/uhyve
[4] https://qemu.readthedocs.io/en/latest/system/i386/microvm.html
[5] https://kubernetes.io/

application deployment but also facilitates the efficient use of cloud resources, ultimately driving cost savings and accelerating the development and delivery of modern cloud-native applications.

In the latest Kubernetes and cloud native operations report [3], 43% of the participants run applications in their organization at least partly on Kubernetes and an additional 30% are evaluating the use of Kubernetes or are planning a migration. In the same survey, the most popular answer to the question "What are your requirements when it comes to implementing an edge strategy?" with almost 50% of the respondents was "Security and compliance".

Since 2015, the Open Container Initiative (OCI) has developed a standard for integrating different runtimes into container technologies each fulfilling different needs. Examples are the original Docker container runtime "containerd", the low-level runtime "crun"[6]. Figure 4 illustrates the interaction of different runtimes with container management tools. The separation between different runtime is partly weak. For instance, "containerd" is a high-level runtime, which manages also container images, while low-level runtimes like "runc" just isolate existing images from the host system and start services within the container. A high-level runtime is able to use a low-level runtime but could also instantiate the container by its own. The OCI standard simplifies the reusing of existing components.

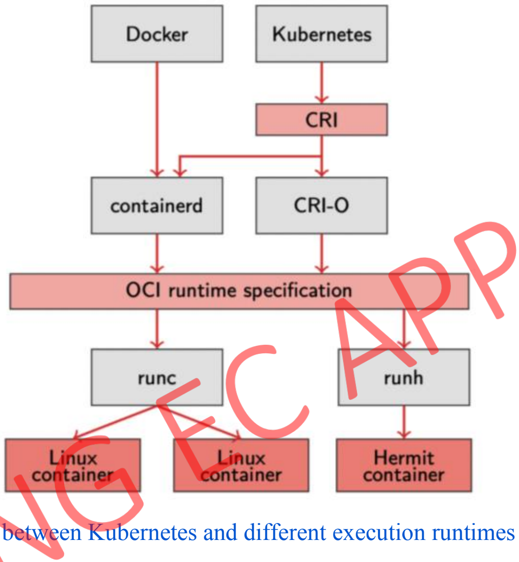


Figure 4: The interaction between Kubernetes and different execution runtimes (here runc and runh)

2.2.3 Trusted Execution Environments/Confidential Containers

Confidential Containers (CoCo) [4] is a cloud-native confidential computing project initiated by the Cloud Native Computing Foundation[7] (CNCF), which leverages various hardware platforms and securities technologies such as Intel SGX[8], Intel TDX[9] and AMD SEV[10] in combination with new software frameworks to secure that in use.

The project focuses on safeguarding data and application process within hardware-based Trusted Execution environments (TEE) to ensure data integrity, data confidentiality and code integrity providing increased security for applications and data in use. CoCo offers two approaches for confidential containers VM-Based TEEs (i.e., Intel TDX and AMD SEV) and process-based TEEs (i.e., Intel SGX). Both approaches aim to remove cloud and infrastructure providers from the trusted computing based and integrate seamlessly with Kubernetes maintaining an unmodified user experience. Also, the project provides a set of key components required for creating a holistic confidential containers platform that

[6] https://github.com/containers/crun
[7] https://www.cncf.io/
[8] https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html
[9] https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html
[10] https://www.amd.com/en/developer/sev.html

can be use by the users to build their platform. This includes their attestation service based in the RATS architecture, their key management service and their Image registry and build services. An example stack is presented in Figure 5.

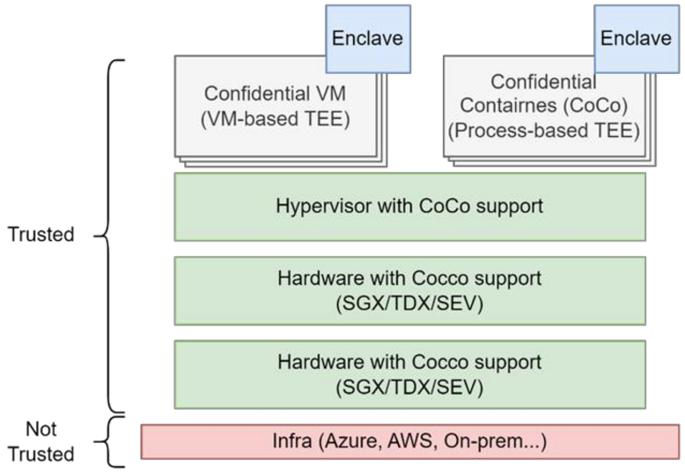


Figure 5: CoCo software stack

2.3 Architecture & Approach

In NEMO we want to build the Secure Execution Environment on top of the powerful and well-established Kubernetes orchestrator. This is done by adding a runtime for Unikernels and one for Trusted-Execution-Environments, as depicted in Figure 6. Additionally, an enhancement of the migration capabilities of Kubernetes is investigated, to allow more fine-grained control over the localization of the services, allowing for more specialized use-cases in edge-cloud scenarios. All of these components are explained in the next sections.

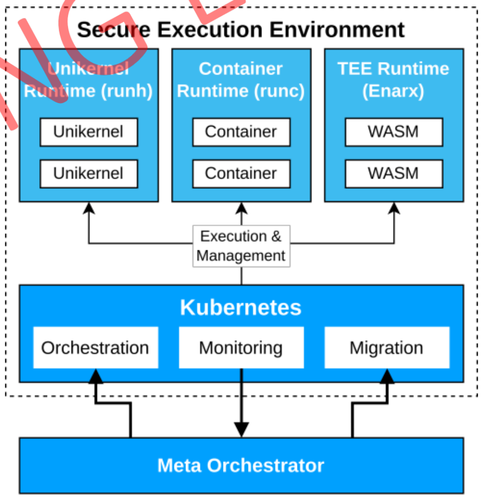


Figure 6: Secure Execution Environment - Architecture

Before we go into the details, we want to illustrate the interactions and usage of this component a little more.

One of the core concepts we investigate for the SEE is the creation and management of heterogeneous clusters. As can be seen in Figure 7, heterogeneity is understood in a 2-dimensional way. For once, we want to execute the services on different runtimes (runc, runh, TEE-runtime), but also on different kinds of nodes. The latter allows for modern edge-cloud scenarios, where computation can be offloaded in small systems that are very close to the user, thus providing minimal latencies. Including the IoT-devices in the cluster to benefit from the orchestration capabilities across the client-server barrier is an idea that we are investigating.

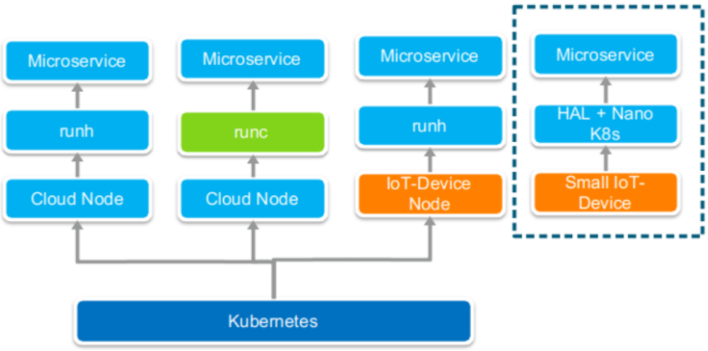


Figure 7: Secure Execution Environment - Development View

As we add multiple runtimes, the execution of the respective calls might be handled differently internally. Figure 8 illustrates exemplary: We have two Kubernetes nodes in our cluster, one providing an edge node and the other one is an IoT-Device in the field. In a stateless migration scenario of an application running on the former, we would first stop the service and restart it on the second device. As the service is running as a container on the edge node, runc handles the stopping whereas runh starts the service on the IoT-Device as a Unikernel for enhanced isolation.

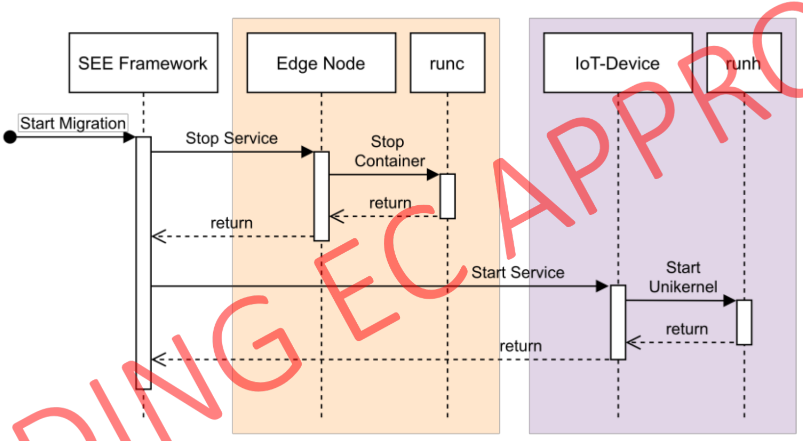


Figure 8: Secure Execution Environment - Process View (migration example)

2.3.1 Unikernel Runtime for Kubernetes

The first necessary enhancement of Kubernetes necessary to form the SEE is a Unikernel runtime. By default, Kubernetes can only orchestrate containers, but as shown in Figure 4, it is possible to exchange the container runtime, as long as it is compatible with the OCI. The respective OCI specification can be found in the Open Containers Github project [5].

A prototype for a Unikernel runtime was already developed: runh [6]. runh can be integrated in Kubernetes' container runtime CRI-O by modifying the configuration at /etc/crio/crio.conf. In that file, the following lines have to be added:

```
[crio.runtime.runtimes.runh]
runtime_path = "<path-to-runh>"
runtime_type = "oci"          # Type of the runtime
runtime_root = "/run/runh"    # Root directory for storage of containers
```

In principle these lines specify the location of the container runtime handler and the interface to interact with the handler. Consequently, this handler can be used to initiate a container. In case of Linux, runc is the typical runtime handler and initiate the container based on Linux' cgroups and namespaces, which limits the access to the hardware and the view to the system. In case of runh, the application runs in minimal virtual machine, which provides a stronger isolation to the hots system.

To offer a new runtime to Kubernetes, the runtime must be registered. Since Kubernetes 1.20 the resource “RuntimeClass” is able to announce a new container handler. The following lines shows an example for the definition of a “runh” runtime class with the name “runh”:

```
# RuntimeClass is defined in the node.k8s.io API group
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  # The name the RuntimeClass will be referenced by.
  name: runh
  # The name of the corresponding CRI configuration
  handler: runh
```

After the registration of runh, a deployment can be defined, which used as container runtime. The following lines show, how a simple web server can be deployed in Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hermit-httpd
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hermit-httpd
  template:
    metadata:
      labels:
        app: hermit-httpd
    spec:
      runtimeClassName: runh
      containers:
      - name: hermit-httpd
        image: ghcr.io/hermit-os/httpd:latest
        ports:
          - containerPort: 9975
        env:
          - name: RUNH_USER_PORT
            value: "9975"
```

The difference to the common deployments, the keyword *runtimeClassName* is used to specify the usage of runh. The current version of runh requires that the container port is also specifies in the environment of the container. Consequently, the environment variable RUNH_USER_PORT is specified with the same value of *containerPort*. In the future, this redefinition will not be required anymore. But in the current stage of the development, it simplifies the development process.

In other words: runh is a drop-in replacement for the common container runtimes like runc, inheriting the standard interfaces created by the OCI. In this example, the container image is available at GitHub’s container registry[11]. However, the image is not a classical image, which based on a Linux distribution. Only the Unikernel and the loader must be part of the image. Common tools like docker to build container images offer the option to build an image from scratch. The following lines shows the content of a “Dockerfile”, which creates a container image from scratch and just copy the Unikernel and the loader to the directory hermit:

[11] https://ghcr.io/hermit-os/httpd:latest

```
FROM scratch
COPY rusty-loader-x86_64 hermit/rusty-loader
COPY httpd hermit/httpd
CMD ["/hermit/httpd"]
```

A Unikernel can be optimized by using static code analysis techniques. A nice side effect is that the compiler is able to remove unused code of the complete image. By the usage of these optimization techniques and the creation of container images from scratch, the resulting image is just 5.8 MB large for a simple web server (including the complete operating system). A similar Linux container with a small web server is 158MB large and clearly larger in comparison to a Unikernel.

The goal is, that a Unikernel Pod running on runh can run without any restrictions, when compared to a normal container. However, there are still a number of important features missing, like the support of micro VMs[12] and encrypted memory. In NEMO we will improve this runtime to fit the NEMO Kernel requirements. Also, runh is currently implementing the OCI's container specification. There exists a VM specification as well, and it will be investigated, if this is a better fit for runh. Runh is licensed under the Apache License, so the use and modification of the runtime is free. The nature of the software and the chosen license still does not hinder commercial services to be executed on this runtime.

2.3.2 TEE Runtime for Kubernetes

To establish a secure environment independently on the infrastructure where the different applications (pods) are going to be run, in NEMO we have explored the different technologies that can provide a TEE a run time independently of the underlying technologies of the host. To run CoCo on top of a Kubernetes Cluster, first is needed to present the compatible TEE technologies of running the pods:

- VM-based TEEs: this model encrypts memory along a traditional VM boundary on top of a VMM. As traditional VMs which offer some isolation, the VMs in this TEE model are shielded by hardware-based encryption keys. This model can be implemented directly using a confidential containers runtime or relying in the kata hypervisor [7].
- Process-based TEEs: processes that need to run in a trusted environment are divided two components, on. The first one, resides in encrypted memory and handles the confidential computing. The untrusted part offers interfaces to the operating to interact with the encrypted memory. Managed data by those services can only enter and exit the encrypted region through predefined channels with strict checks and must be encrypted on transit so it can be only understood by the software running in the TEE.

Also, the CoCo architecture provides a set of elements that are used to ensure that the environments where a TEE runs can be trusted:

- Attestation Service and Key Broker Service (KBS): these components are used to verify and attestate the TEE following the architecture and models defined in the RATS architecture and provide the necessary keys so they can get access to the images stored in the Containers Images Registry after a successful attestation.
- Image build service and Image registry: both services are part of the building and storing process of confidential containers and VM images that will be later stored in the Containers Images Registry. Once an application is created, it will be stored encrypted and/or signed for the attested CoCo workloads.

As an example, Figure 9 presents workflow of a deployment using CoCo in a VM-Based TEE:

- Steps 1 and 2, are part of the remote attestation procedure, where the agent request this process following the Background-Check Model of RATS [8].
- If the evidence sent by the enclave agent are valid, the relying party (also known as Key Broker Service) in steps 3 and 4 will ask for the necessary keys to decrypt the containers images, so they can be forwarded to the enclave agent.

[12] micro VMs

- Using these keys, the enclave image management will request to the container's registry the encrypted container images in step 5, so in step 6 they can decrypted and start the container workload.

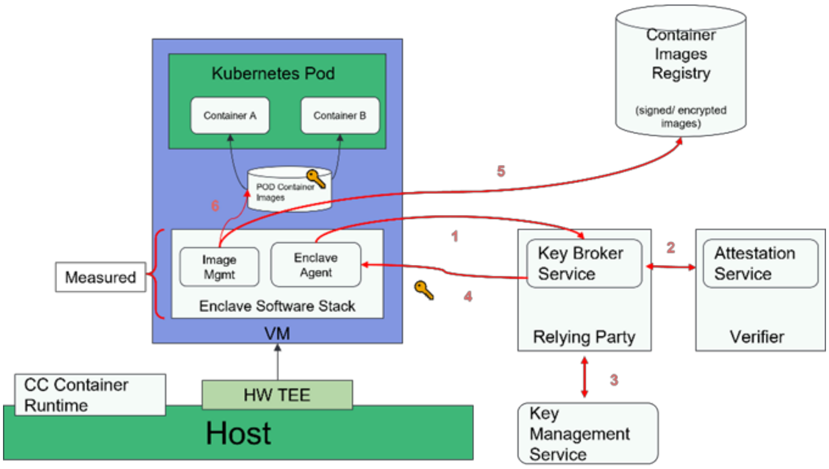


Figure 9: Workflow when deploying a Kubernetes pod with VM-based TEEs

Another example can be found in Figure 10, where they rely in process-based TEEs using in this example Intel SGX, where the only differences are in the last steps. Where first, it stores the container images in a local registry after decrypting and verifying the downloaded image from the container's images registry. Then, after the runtime initiates the application enclave, which is required to access the container bundle from the encrypted file system, it must commence by establishing a key exchange between the agent and the application enclave. This exchange is necessary to subsequently extract the workload and initiate its execution.

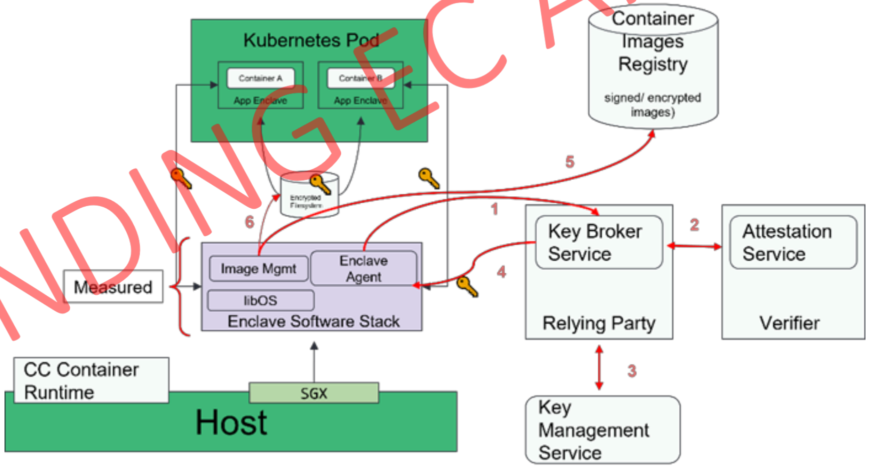


Figure 10: Workflow when deploying a Kubernetes pod with a process-based TEEs

2.3.3 Migration extension for Kubernetes

The NEMO migration extension for K8s is necessary to enable the migration of microservices across nodes in the IoT-Edge-Cloud continuum. K8s does not currently support migration of Pods out-of-the-box. The state-of-practice approach for migrating Kubernetes services is by stopping the pods that run on the source node and recreating and executing them from scratch in the destination node. This is the approach followed in the NEMO meta-OS.

The migration extension is not responsible for the migration decision (i.e., under what circumstances should a migration occur and between which nodes) but only for the low level execution of the Pod migration. It interacts with other NEMO components such as the meta-Orchestrator's decision engine by taking as input the migration request which includes the Pod name and the destination node.

In order to support migration of stateless services (i.e., services without persistent state that needs to be preserved across the migration process) we exploit the K8s Application Programming Interface (API) in the following way depending on whether the migration concerns a single Pod or a deployment. In case we want to migrate a deployment things are simpler since we can directly edit the deployment K8s object and change the nodeName field to match the desired destination node. After we do that the K8s engine will by itself stop all the deployment's Pods in the source node and start them in the destination node. In case we want to migrate a single Pod things are slightly more complicated since K8s does not allow to modify the nodeName field of a running Pod. To overcome this limitation, we first export the Pod's state yaml file, delete the Pod, appropriately modify its state yaml file and reapply the modified config in the K8s cluster. This way the new Pod has the exact same config with the deleted one with the only difference that it now runs on the destination node.

The above procedure is a logical and performant approach for stateless services, but it is not particularly useful for the case of stateful services. In such scenarios we cannot just stop the service and start it on another node since its state will be lost. For these cases several approaches use the CRIU Linux tool[13]. CRIU enables to freeze a running Linux process (i.e., a container process), checkpoint its whole state to disk, upload it to another node and restore the Linux process with its full state on the destination node. There are already efforts that use CRIU for migration in Kubernetes [9] [10] [11] [12] [13] and our plan is to explore whether and how this approach cab be integrated with K8s to enable stateful migration of microservices, when possible, in the NEMO meta-Operating System.

An important aspect in the stateful migration procedure is the compatibility of the migration mechanism (i.e., CRIU mechanism) across different node architectures. More specifically migrating a stateful microservice from an Edge node to a Cloud node, and vice versa, with CRIU might be challenging, or even impossible, due to different architecture characteristics (e.g., x86 to aarch64).

2.3.4 Interaction with other NEMO components

In Nemo, the SEE provides a fundamental building block that can execute the relevant microservices. As such, it provides three major interfaces for other components, as can be seen in Figure 1.

The first is a control interface for configuring and starting the microservices. This interface is inherited by Kubernetes, as such it mostly consists of the well-known yaml files that configure a service. The components using this interface are the security and access control technologies developed in task 3.3, as well as the meta-Orchestrator developed in task 3.4.

The second interface is for controlling the migration of the tasks. This is planned to integrate seamless in the Kubernetes API, but in contrast to the first interface it is a new interface. The exact definition of this interface is not defined yet.

As a microservice orchestrator, the SEE does of course also interface with the microservices it executes. The API for this depends on the kind of service (VM vs Container vs Unikernel), but in all cases contains functionality for controlling and monitoring the service.

2.4 Conclusion, Roadmap & Outlook

The presented architecture is the outcome of this project with regards to task 3.1. The SEE will be a collection of Kubernetes enhancements to enable modern security technologies for this crucial building block in today's cloud environment. Furthermore, the migration capabilities of Kubernetes are under investigation, so that it fits better into the edge-cloud paradigm, promising lower latencies for critical services.

So far, task is well in schedule and no major roadblocks were encountered. The next period of the task will focus on the implementation and refinement of the architecture in an iterative manner to present a

[13] Checkpoint/Restore in Userspace: https://criu.org

final picture in the upcoming deliverables of work package 3. Specifically, it is planned to provide a prototype of the components for deliverable D3.2 and refine and finalize the implementation for deliverable D3.3.

3 PRESS, Safety & Policy enforcement framework

The PRESS, Safety & Policy Enforcement framework delivers two distinct functionalities. The first one described by PRESS and safety concerns Privacy, data pRotection, Ethics, Security & Societal aspects associated with next generation AIoT, especially related with personalized sensing and potential privacy and ethical intervention to the human life. The latter tackles the Policy Enforcement compliance of the NEMO-hosted micro-services with regards to the policies defined by the service and the application providers. Hereinafter, we refer to the PRESS, Safety & Policy Enforcement framework as the Privacy and Policy Enforcement framework (PPEF).

3.1 Overview

The PPEF's main objective is to safeguard security, privacy, ethics, cost, performance, environmental requirements and associated concerns that are defined for each of the NEMO-hosted services by the service provider. Subsequently the PPEF should be able to monitor and enforce these dynamically defined services' requirements expressed by the NEMO user in the form of a Service Level Agreement (SLA). At the same time the PPEF safeguards the provisioning of user and data privacy preservation characteristics that should always be offered by any system such as NEMO meta-OS.

Therefore, Privacy and Policy enforcement is not only crucial for guaranteeing the optimal performance of the NEMO-hosted services, but it is also mandatory for safeguarding the privacy, integrity and ethical aspects of NEMO-user data. As NEMO meta-OS concerns the deployment and optimal management of the hosted third-party applications in the continuum of IoT, edge and cloud underlying infrastructures, it is evident that privacy and policy enforcement endeavour is a rather challenging and demanding task.

NEMO puts a significant effort to be fully compliant with General Data Protection Regulation (GDPR) directives and relevant legislations. In D1.1 [1], section 6 provides an overview of the compliance management activities regarding ethical issues that the NEMO project addresses. In addition, D5.1 – "Data Management Plan", focuses on the management of the research information as well as the processing of personal data in the context of the NEMO Living Labs.

From the point of view of PPEF, we consider the aforementioned work that has been presented in D1.1 and D5.1 [14] a mandatory prerequisite that has already been achieved, thus enabling the orchestration of the policy enforcement activities that will be managed in practical terms from the PPEF dynamically in NEMO meta-OS.

The GDPR provides seven principles of personal data processing: 1) lawfulness, fairness and transparency; 2) purpose limitation; 3) data minimization; 4) accuracy; 5) storage limitation; 6) integrity and confidentiality and 7) accountability. The PPEF by design aims to address the aforementioned principles in the framework of its monitoring and policy enforcement activities pertaining to the NEMO-hosted services lifecycle management. Having said that, SLA management as a contractual agreement between a service provider and a customer that outlines the level of service that will be provided is the main focal point of PPEF.

3.2 Background

Service Level Agreements (SLAs) has been initially incorporated by telecom operators in the late 1980[14] as means of an agreement between the service provider and the customer. SLAs specify in short, particular aspects of the quality and performance of a service. Nowadays, SLAs definition is a common tool and is usually established in order to help reassure the Quality of Service (QoS) amongst different parties.

SLAs typically include details such as uptime guarantees, response times, support availability, and penalties for not meeting the agreed-upon service levels. A particular, quantifiable objective which is

[14] https://en.wikipedia.org/wiki/Service-level_agreement

derived from an SLA is known as a Service Level Objective (SLO). It establishes the performance or quality standards that a service provider must meet in order to provide a given service. Typically, SLOs are described in terms of one or more metrics, such as uptime, response time, error rate, or throughput. They offer a measurable indicator of how well the service is performing at the specified level. A specific metric or group of metrics known as a Service Level Indicator (SLI) are used to assess the functionality or behaviour of a service. SLIs are frequently created from monitoring data and offer a numerical illustration of the service's effectiveness.

SLAs define the specific metrics, targets, and responsibilities related to the service being provided. An SLA can most commonly concern the (1) *Service description,* (2) *Service metrics,* (3) *Performance targets,* (4) *Responsibilities,* (5) *Reporting and monitoring,* (6) *Remedies and penalties,* (7) *Termination and dispute resolution.* In the light of the above, SLAs are necessary for defining expectations and establishing accountability between service providers (NEMO meta-OS) and clients (3rd parties – NEMO users). They provide as a benchmark for evaluating and controlling service performance, enabling mutual evaluation and enhancement of the level of service being rendered. Steps in SLA Lifecycle are presented in Figure 11 and described in detail below. SLA lifecycle, as stated above, concerns to roles. The first one refers to the organization that has adopted NEMO meta-OS and the service provider that refers to the 3rd party that aims to deploy a service (app provider) or offer infrastructure resources (infrastructure provider).

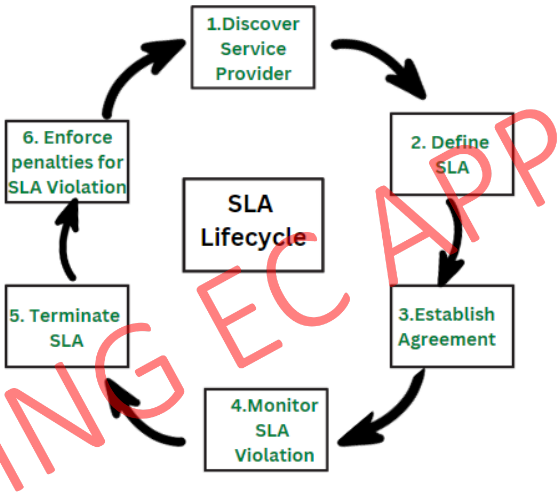


Figure 11: SLA lifecycle steps[15]

In more detail, the SLA lifecycle steps can be described as follows.

1. Discover service provider: This step involves identifying a service provider that can meet the needs of the organization and has the capability to provide the required service.
2. Define SLA: In this step, the service level requirements are defined and agreed upon between the service provider and the organization. This includes defining the SLOs, metrics, and targets that will be used to measure the performance of the service provider.
3. Establish Agreement: After the service level requirements have been defined, an agreement is established between the organization and the service provider outlining the terms and conditions of the service. This agreement should include the SLA, any penalties for non-compliance, and the process for monitoring and reporting on the service level objectives.
4. Monitor SLA violation: This step involves regularly monitoring the service level objectives to ensure that the service provider is meeting their commitments. If any violations are identified, they should be reported and addressed in a timely manner.

[15] https://www.geeksforgeeks.org/service-level-agreements-in-cloud-computing/

5. Terminate SLA: If the service provider is unable to meet the service level objectives, or if the organization is not satisfied with the service provided, the SLA can be terminated. This can be done through mutual agreement or through the enforcement of penalties for non-compliance.
6. Enforce penalties for SLA Violation: If the service provider is found to be in violation of the SLA, penalties can be imposed as outlined in the agreement. These penalties can include financial penalties, reduced service level objectives, or termination of the agreement.

With respect to the NEMO project technical orientation needs and due to the complex nature of a thorough SLA template description, the work presented in this report focuses on the SLA definition and Monitor SLA violation steps (2 and 4 respectively). A main limitation of current Platform as a Service (PaaS) offerings is that the provided guarantees are based exclusively on *resource availability* (e.g., number of virtual machines, memory size), rather than on application *QoS properties* (e.g., response time, throughput), which are more meaningful and useful to the user. As a result, it used to be responsibility of PaaS user to ensure QoS properties for their applications, which limits the value of PaaS systems. The NEMO project attempts a deep dive into QoS workload performance requirements' definition, management and monitoring enhancing the Quality of Experience (QoE) of the users of a NEMO meta-OS adopted platform.

3.2.1 State-of-the-art

The PPEF design and development capitalizes on a thorough research on Cloud Native Cloud Foundation (CNCF) policy definition and enforcement tools. A thorough investigation of state-of-the-art Cloud Native technologies and practices tailored for SLA definition and resource monitoring have been conducted as presented in the following paragraphs.

Kubernetes admission control[16]

Kubernetes Admission Control is a feature in Kubernetes that allows administrators to define and enforce policies on the cluster. It acts as a gatekeeper, determining whether requests to create or modify resources in the cluster are allowed based on defined policies. Administrators may enforce uniform policies across the cluster, enhance security, and avert misconfigurations by employing admission control. It aids in keeping the cluster in the desired condition and guarantees that resources are created and updated in accordance with the established rules.

Kubernetes API[17]

The Kubernetes API enables users to carry out a variety of tasks on a Kubernetes cluster, including the deployment and management of applications, resource scaling, networking configuration, and cluster health monitoring. It offers a uniform interface for communicating with the cluster and abstracts away the underlying difficulties of maintaining a distributed system. The Kubernetes API adheres to the Representational State Transfer (REST) principles and is hence RESTful. It makes use of JSON or YAML for data serialization and the HTTP methods GET, POST, PUT, and DELETE to conduct operations on resources.

The Kubernetes API is structured around resources, which stand in for various components of a cluster, including pods, services, deployments, and namespaces. Users can interact with these endpoints to add, read, update, or delete resources because each resource has a distinct endpoint in the API.

Open Policy Agent (OPA)[18]

An open-source policy engine called Open Policy Agent (OPA) offers a declarative language for establishing and enforcing policies throughout different software systems. It allows for flexible policy enforcement, policy-based decision-making, and fine-grained access control in cloud-native systems.

[16] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/
[17] https://kubernetes.io/docs/concepts/overview/kubernetes-api/
[18] https://www.openpolicyagent.org/

OPA is intended to be a multifunctional policy engine that can be linked with a variety of platforms, such as Kubernetes, microservices, API gateways, and more. It offers a standardized method for defining and managing policies across different systems, encouraging uniformity and minimizing complexity. Some key features and concepts of OPA include:

1. Declarative policy language: OPA uses a high-level declarative language called Rego to express policies. Rego allows users to define rules and constraints in a human-readable and expressive manner.
2. Policy evaluation: OPA evaluates policies against incoming requests or data to make policy-based decisions. It performs efficient and fast evaluations by using a rule-based evaluation engine.
3. Policy data model: OPA provides a flexible data model that allows policies to be written against any data structure. It supports JSON, YAML, and other data formats, allowing policies to be written based on the specific data being evaluated.
4. Policy enforcement points: OPA can be integrated into various systems as an external authorization component. It can intercept requests and provide fine-grained access control by evaluating policies before allowing or denying access.
5. Dynamic policy updates: OPA supports dynamic policy updates, allowing policies to be updated and reloaded without restarting the system. This feature enables real-time policy changes and reduces downtime.
6. Policy composition: OPA allows policies to be composed and organized into modules. This facilitates policy reuse, modularity, and separation of concerns.

Kubernetes Scheduler Extender[19]

Kubernetes Scheduler Extender is a feature in Kubernetes that allows users to customize the scheduling process by extending the existing scheduler functionality. It enables users to influence how pods are assigned to nodes based on their specific requirements and constraints. The Kubernetes scheduler first examines the default scheduling algorithm when making a scheduling decision. The extender is invoked to offer extra input or overturn the default choice if the default scheduler is unable to make one or if there is a registered scheduler extender. Users have the freedom to modify the scheduling process in accordance with their particular needs thanks to the Scheduler Extender function. In addition to taking into account certain restrictions and preferences, it enables more intelligent and effective pod placement.

The Kubernetes scheduler first examines the default scheduling algorithm when making a scheduling decision. The extender is invoked to offer extra input or overturn the default choice if the default scheduler is unable to make one or if there is a registered scheduler extender. Users have the freedom to modify the scheduling process in accordance with their particular needs thanks to the Scheduler Extender function. In addition to taking into account certain restrictions and preferences, it enables more intelligent and effective pod placement.

Prometheus[20]

Prometheus is a Cloud Native Computing Foundation (CNCF) accepted, open-source monitoring and alerting system. It is a commonly used tool for gathering and analyzing metrics from multiple systems and is one of the most well-liked monitoring solutions in the cloud-native ecosystem. Prometheus is appropriate for monitoring containerized applications running on Kubernetes because it is made to monitor highly dynamic and dispersed settings. It operates according to a pull-based approach, periodically scraping metrics from targets that have been set up, such as applications, services, or infrastructure parts.

Key features of Prometheus include:

- Data model: Prometheus stores time-series data in a compressed and efficient format. It uses a multi-dimensional data model, allowing metrics to be identified by a combination of labels. This enables flexible querying and filtering of metrics.

[19] https://kubernetes.io/docs/concepts/extend-kubernetes/

[20] https://prometheus.io/

- Metrics collection: Prometheus can collect metrics from various sources, including instrumentation libraries, exporters, and service discovery mechanisms. It supports multiple protocols for metric ingestion, such as HTTP, SNMP, and remote write.
- Query language: Prometheus provides a powerful query language called PromQL (Prometheus Query Language) for analyzing and aggregating metrics. PromQL allows users to perform complex queries, calculations, and transformations on the collected time-series data.
- Alerting and notification: Prometheus has built-in support for defining alerting rules based on metric thresholds or patterns. It can trigger alerts and send notifications via various channels like email, PagerDuty, or other custom integrations.
- Visualization: Prometheus has a basic built-in web-based graphical interface called the Prometheus Expression Browser. However, it is often used in conjunction with other tools like Grafana for more advanced and customizable visualization of metrics.
- Integration with Kubernetes: Prometheus has native integration with Kubernetes, allowing it to automatically discover and monitor Kubernetes services, pods, and nodes. It can gather metrics from the Kubernetes API server and other components, providing insights into the health and performance of the cluster.

Prometheus is highly extensible and has a rich ecosystem of exporters and integrations with various systems and frameworks. It is widely adopted by organizations for monitoring and observability in cloud-native environments, providing valuable insights into the performance and behaviour of applications and infrastructure components.

Scaphandre[21]

Scaphandre is an open-source energy monitoring tool which is designed to measure and analyse energy consumption in a data center or server infrastructure. Scaphandre collects energy-related data from servers and provides insights to help optimize energy usage and improve efficiency. More specifically Scaphandre provides:

- Real-time energy monitoring: continuously collects energy-related data from servers, including power consumption, CPU utilization, temperature, and other metrics. It provides real-time visibility into energy usage at the server level.
- Alerts and notifications: can be configured to send alerts or notifications when certain energy-related thresholds are exceeded. This enables proactive monitoring and helps detect anomalies or potential issues.
- Integration with monitoring systems: can integrate with existing monitoring systems, such as Prometheus or Grafana, to visualize and analyse energy-related metrics alongside other infrastructure monitoring data. This allows for a holistic view of the infrastructure's performance and energy efficiency.
- Historical data and reporting: can store historical energy consumption data, allowing for trend analysis and long-term energy usage monitoring. It can generate reports and visualizations to help track energy efficiency improvements over time.

Kepler[22]

Kepler is an open-source energy monitoring tool designed to measure and analyse energy consumption in computing systems, with a particular focus on high-performance computing (HPC) environments. Kepler collects energy-related data from various sources, such as power meters, energy sensors, and system-level measurements, to provide insights into energy consumption and efficiency. It is primarily used in HPC clusters and data centers to understand and optimize energy usage. Kepler’s key features include:

- Energy data collection: Kepler gathers energy-related data from different sources, including power meters and sensors, to monitor and measure energy consumption in computing systems. It can collect data at various levels, from individual nodes to entire clusters.

[21] https://github.com/hubblo-org/scaphandre

[22] https://github.com/sustainable-computing-io/kepler

- Real-time monitoring: Kepler provides real-time visibility into energy consumption, allowing users to monitor and analyse energy usage in their computing infrastructure. It enables administrators to identify energy-intensive applications or inefficient nodes.
- Energy efficiency analysis: Kepler analyses the collected energy data to identify energy inefficiencies and opportunities for optimization. It can help identify resource-intensive tasks, bottlenecks, or areas for improvement to enhance energy efficiency.
- Visualization and reporting: Kepler offers visualizations and reports to present energy consumption trends and patterns. It allows users to track energy usage over time, compare different nodes or applications, and generate reports for further analysis and decision-making.
- Integration with HPC systems: Kepler integrates with HPC management tools and frameworks to collect energy data and correlate it with other system metrics. It can work with job schedulers, resource managers, and monitoring systems commonly used in HPC environments.

Grafana[23]

Grafana is an open-source data visualization and monitoring tool. It is commonly used to create interactive dashboards and visualizations for time-series data, making it popular for monitoring and observability purposes. Grafana supports a wide range of data sources, including databases, cloud services, and monitoring systems. Grafana is widely used in various domains, including IT operations, DevOps, IoT, and business intelligence. It provides a flexible and intuitive platform for visualizing and analysing data, enabling users to gain insights and make data-driven decisions.

Keycloak (policy driven authorization)[24]

Keycloak is an open-source identity and access management (IAM) solution that provides features like authentication, authorization, and user management. Policy-driven authorization is a key aspect of Keycloak that allows administrators to define fine-grained access control rules based on policies. In policy-driven authorization, access control decisions are based on policies that are defined and enforced by Keycloak. These policies can be created and managed within Keycloak's administration console.

The abovementioned technologies and tools have been reviewed and either inspired the development of the PPEF particular modules or were adapted to meet the framework’s functional needs or were integrated into the framework as is. Additional details on the utilization of specific technologies by the PPEF are provided in the following sections.

The monitoring of the underlying NEMO infrastructures’ resources will be orchestrated by Prometheus, which is a proven CNCF accepted, systems monitoring toolkit. Additional CNCF approaches and/or tools might be selectively adopted. Moreover, AI/ML solution, provided through the CF-DRL component will be investigated in the context of the PRESS & Policy Enforcement framework to enhance the quality of the policy related decision making on NEMO hosted micro-services.

3.3 Architecture & Approach

The PPEF architectural description has been defined based on the meta-OS meta-Architecture Framework (MAF) which aims to facilitate the design of meta-OS ecosystems, in a way that they will be scalable, extensible, modular and interoperable. The MAF incorporates a set of meta-Architecture viewpoints that PPEF capitalizes in order to structure and refine its particular reference architecture. The MAF viewpoints are presented in D1.2 [2]. This section introduces the high-level architecture of the PPEF, focusing on the high-level description of the PPEF architectural layers and the respective functionality that they deliver. At the same time the placement of the PPEF within the NEMO meta-OS architecture is discussed.

[23] https://grafana.com/

[24] https://www.keycloak.org/docs/latest/authorization_services/index.html

The Logical viewpoint of the PPEF has been described in detail in D1.2 (section 5.5.4). The purpose of this viewpoint is to highlight on one hand the required functionalities that need to be delivered by the system and on the other to identify some high-level relationships and associated workflows including the actors who is envisioned to interact with the system. For the sake of completeness, the high-level functionality that is offered by the PPEF can be summarized below:

- By design provisioning of Privacy, data pRotection, Ethics, Security & Societal aspects (PRESS). These PRESS considerations are incorporated into the SLA template that is negotiated and subsequently agreed by the NEMO meta-OS enabled organization and the 3rd party (meta-OS consumer). The majority of the PRESS considerations should be by design provided by the PPEF. However, for security related PRESS considerations imposed by a 3rd party NEMO-user, this would potentially lead to the utilization of particular secure-oriented resources and tools that are inherited to NEMO meta-OS functional stack (e.g. SEE).
- SLA definition. A service level agreement (SLA) is a written contract that specifies the services needed and the expected quality of service between NEMO meta-OS enabled organization or Platform as a Service (PaaS) provider and a consumer.
- Monitor SLA violations (Step 4 of the SLA lifecycle). Concerns the monitoring of the performance and usage of various resources inside a distributed and NEMO meta-OS enabled heterogeneous infrastructure environment the covers the IoT to Edge to Cloud continuum.

The PPEF architecture presented in Figure 12, is driven by the functional requirements that were introduced during the logical design of the framework. The list of the requirements that the PPEF aims to address were presented in D1.2 (section 5.5.4).

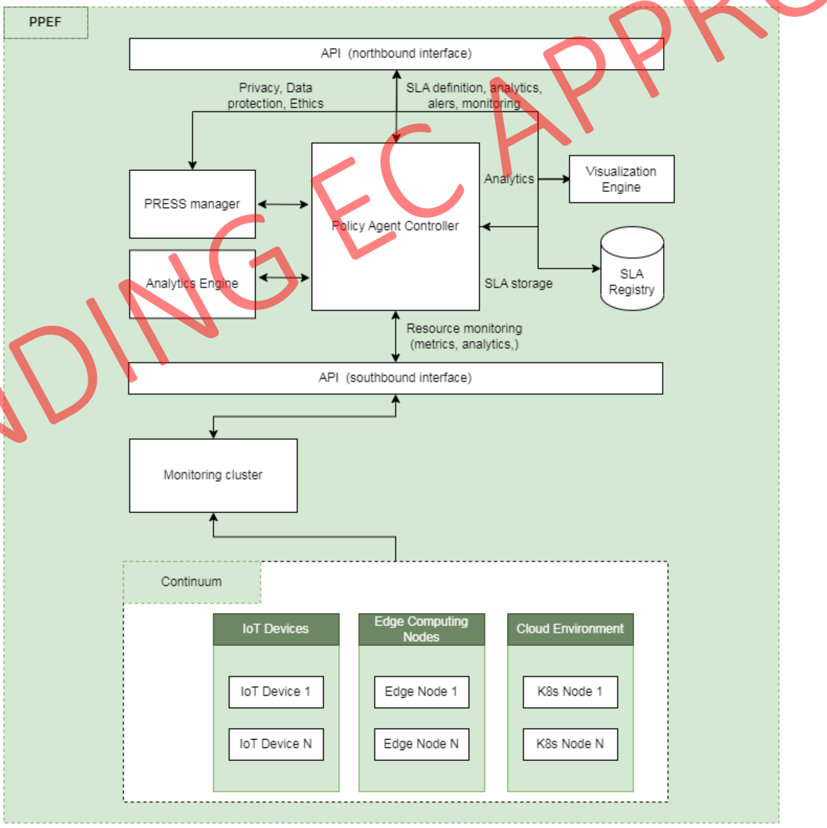


Figure 12: Privacy & Policy Enforcement Framework architecture

The PPEF consists of various of modules that deliver a specific set of functionalities facilitating the SLA definition by the NEMO user, the underlying infrastructure resource monitoring, the NEMO-hosted micro-services performance monitoring, including the provisioning of this information back to the user and also the inter and intra communication through REST APIs and the storage needs of the framework. The PPEF architectural components are described in the section below.

- The *PRESS manager* facilitates the SLA definition process by providing to the user GDPR, privacy, confidentiality and security enablers. The PRESS manager handles the consensus decision making agreement in view of the SLA negotiation a dynamic way guarantying the ethical, confidential and secure utilization of the NEMO meta-OS capabilities both by and the 3rd party service provider and by the NEMO meta-OS enabled platform. The PPEF aims to investigate the possibility to extend the PRESS manager functionality through the Verification and Validation component that aims to test the security aspects of the NEMO-hosted applications, thus offering an additional layer of privacy and security protection.
- The *Policy Agent Controller (PAC)* is the core of the PPEF. The PAC facilitates the SLA definition process in coordination with the PRESS manager. Its main role is to manage the monitoring process of the NEMO-hosted services which is driven by the SLAs' specified QoS properties. The PAC realizes the SLA definition procedures that handle the SLA's performance targets' definition as part of the SLA negotiation process. The module interfaces internally of the PPEF with the visualization engine, providing analytics that are collected by the underlying infrastructure and through the southbound interface with the deployed Prometheus cluster. In addition it interfaces with an relational database (PostgreSQL) that holds the SLA manifests. Lastly, the PAC communicates its NEMO-hosted service monitoring analysis and notifies the meta-orchestrator in case of an SLA breach.
- The PPEF *Analytics Engine* projects to finetune the SLA compliance endeavour by utilizing predictive analytics based on the collected resource monitoring data and communicate the potentially problematic performance targets to the meta-orchestrator allowing for corrective actions before an SLA breaks. Thus, optimizing both the performance of the NEMO-hosted services and the monetary benefit of the platform.
- *The Monitoring Cluster* follows a federated resource monitoring architectural approach enabling NEMO services' policy enforcement in the Cloud-Edge-IoT continuum. The Prometheus cluster closely interacts with additional resource monitoring tools such as Kepler and Scaphandre allowing for rich resource monitoring data collection addressing the SLA-defined performance targets of a NEMO-hosted service.
- The *SLA registry* realized as a relational database interfaces with the PAC and provides SLA storage, filtering and sorting capabilities.
- *The southbound interface* facilitates the communication of the PPEF with the deployed Prometheus cluster. In addition, the southbound interface allows for the communication with the NEMO kernel and the Meta-Orchestrator component.
- The *northbound interface* of the PPEF is the user-facing layer of the PPEF which provides main abstraction mechanisms allowing for the definition of the SLAs between the NEMO meta-OS consumer and the NEMO meta-OS provider[25]. Through the northbound interface the NEMO meta-OS consumer is able to access PPEF resources, visualized in the *visualization engine*. Moreover, the northbound interface is protected through authentication and authorization capabilities, provided by the NEMO Access Control service.
- The *visualization engine* is realized through a Grafana open source analytics and interactive visualization web application that provides to the NEMO users means of NEMO-hosted services' resource monitoring.

3.3.1 PPEF Service Level Agreement

Regarding SLA definition (Step 2 of the SLA lifecycle) the following high-level template is adopted by the NEMO project focusing on its technical and functional aspects. The NEMO project will investigate

[25] Meta-OS Provider and Consumer user roles are defined in D1.2.

the business and operational aspects of an SLA definition and plans to incorporate them into the adopted template that is presented in this section.

1. *Service description*: Describes the service that is provided by the organization. It should also include information about the service criticality in terms of security, performance and availability.
2. *PRESS:* Data protection processes, GDPR compliance, security and encryption practices, including backup and disaster recovery approach, should also be addressed.
3. *Performance targets (Service Level Objectives - SLOs):*
    a. *Availability*: The NEMO meta-OS enabled organization must ensure high availability. Typically high availability for the NEMO meta-OS operated cloud infrastructure should be in the magnitude of 99.5% (monthly basis). Optionally information regarding the mean time between failures (MTBF) or mean time to repair (MTTR) can be included.
    b. *Scalability*: Details the ability to scale up and down.
    c. *Additional objectives:* Including statistically predefined SLOs on security, privacy, cost, environmental impact and workload requirements SLOs.
4. *Service metrics (Service Level Indicators - SLIs):* Monitoring metrics that address the defined and agreed SLOs.
5. *Reporting and monitoring information:* Describes the monitoring process and capabilities of the system.

3.3.2 SLA Performance targets (SLOs)

The PPEF considers a series of performance targets for managing the NEMO-hosted services. This section summarises the collection of the SLOs and the associated metrics that the PPEF envisages to capture through its resource monitoring tools. These qualitative captured data can provide insights from the point of view of the infrastructure, the NEMO-hosted application, the deployed IoT devices or the network.

- *Latency*: It refers to monitoring the latency of a system, including the response times for APIs and database queries. IT is achieved by measuring the delay or response time between a request being sent and the corresponding response being received. This metric is important for real-time applications, such as video conferencing or XR applications, where low latency is crucial. SLOs for latency can specify the maximum acceptable delay in milliseconds.
- *Error Rates*: It concerns monitoring error rates concern the health and reliability of a system. These metrics can be used to calculate error based SLOs and trigger alerts when error rates exceed defined thresholds.
- *Throughput*: It refers to monitoring the throughput or request rates of the deployed services. This metric examines if the expected workload and can handle the desired traffic volume by tracking the number of requests processed over time.
- *Capacity:* It refers to monitoring resource utilization metrics, can help identify bottlenecks or capacity issues in a system. It optimizes resource allocation, avoids bottlenecks, and minimizes wastage. Associated metrics include the CPU usage, memory usage and disk space storage.
- *Scalability*: It focuses on the ability of an infrastructure to scale and handle increased workload or traffic. It ensures that the underlying infrastructure can scale up or down based on demand and that it maintains performance and availability under varying loads.
- *Availability:* It concerns the collection of metrics related to the uptime or availability of the deployed services.
- *Disaster Recovery/Backup*: Measures the effectiveness of a systems disaster recovery and backup processes. It ensures that the infrastructure has appropriate backup mechanisms in place and can recover quickly in case of data loss or system failures.
- *Security*: Monitors the security of an infrastructure and includes measures such as access controls, encryption and also vulnerability management.

- *Incident Response:* monitors the effectiveness and efficiency of your incident response process. ensuring a timely response and resolution of issues. Associated metrics are the mean time to detect (MTTD) and mean time to resolve (MTTR).
- *PRESS compliance*: Focuses on adhering to regulatory and compliance requirements ensuring that both the infrastructure and the NEMO-hosted service meets the necessary compliance standards, such as GDPR.
- *Energy consumption:* It aims to evaluate the percentage of renewable energy utilized by the infrastructure. It considers metrics such as CO2 footprint and NEMO-hosted service/application specific energy consumption.
- *Cost optimization*: Focuses on optimizing the cost of the platform. It ensures that the platform infrastructure is cost-effective, avoids unnecessary expenses (use of resources), and maximizes return on investment (ROI).
- *Packet Loss*: Packet loss SLOs define the acceptable percentage of lost or dropped packets during data transmission. Packet loss can degrade network performance and impact the quality of applications. SLOs for packet loss typically aim for a very low percentage, such as 0.1% or lower.
- *Bandwidth*: SLOs for bandwidth specify the minimum or maximum bandwidth capacity guaranteed for a particular service. This ensures that the network can handle the required data transfer rates without congestion or performance degradation. SLOs for bandwidth can be expressed in terms of minimum or maximum Mbps (Megabits per second).
- *Jitter*: Jitter SLOs measure the variation in latency or delay between packets within a network. It is particularly important for real-time applications where consistent packet delivery is critical, such as voice or video communications. SLOs for jitter can specify the maximum acceptable variation in milliseconds.

3.3.3 Interaction with other NEMO components

The PPEF is established as a vertical over the NEMO architectural horizontal layers. The PPEF interacts with the NEMO service management, NEMO kernel and the underlying infrastructure layers. More specifically, PPEF communicates with the following components:

- *Plugins & Application LifeCycle Manager*: The PPEF receives as an input SLAs definition that concern the plugins that will be deployed in NEMO meta-OS and also the PPEF will initiate the SLA negotiation process upon a service deployment request.
- *Monetization and Consensus-based Accountability (MOCA):* The PPEF receives policy requirements by third-parties resource owners (metaOS partners) and communicates back to the MOCA resource utilization metrics that concern the resources that were made available in the framework of the MOCA.
- *Meta-Orchestrator:* The PPEF is tightly interconnected with the NEMO kernel and meta-orchestrator component. Through the established northbound interface, the PPEF communicates to the meta-orchestrator NEMO-hosted resources monitoring analytics, alerts and notifications.
- *meta–Network Cluster Controller (mNCC):* The PPEF collects network related resource monitoring input that are captured by the NEMO underlying network component. The mNCC communicates this data through the PPEF's southbound interface.
- *Cybersecure Micro-services Digital Twins (CMDT)*: The PPEF monitors services' health and performance based on defined SLAs and identifies potential or incidents of violations. Such SLA violation information is communicated to CMDT, in order to ensure traceability of NEMO workloads' lifecycle, which is further supported by CMDT's integration with distributed ledgers.
- *NEMO Intent-based API*: The PPEF services are exposed through the NEMO Intent-based API to external meta-OS users, such as meta-OS partners, wishing to integrate PPEF functionality in their applications or plugins, such as SLA/SLO definition, service analytics, etc.

3.4 Conclusion, Roadmap & Outlook

The PPEF envisages to realize a rather complex system which will be used to define rules, standards, and guidelines for service provisioning, usage, and management and subsequently enforce policies related to the provisioning, management, and usage of the NEMO-hosted services. The PPEF facilitates the SLA compliance by enforcing policies related to performance, availability, and quality of service. It ensures that services within NEMO adhere to predefined policies maintaining also consistency, security, and regulatory compliance across their service offerings. The PPEF provides a centralized mechanism for governing and controlling the performance of a deployed service promoting consistency and reducing the risk of non-compliance or misconfiguration. In addition, the PPEF plays a crucial role in enhancing security by enforcing policies related to access control, data protection, and authentication. It helps prevent unauthorized access, ensure the confidentiality and integrity of data, and enforce security best practices across services.

At the time of writing these lines, the PPEF has a well-defined architecture, has identified the internal and external interfaces within the PPEF and within the NEMO meta-OS. Moreover, it has a mature resource monitoring toolset setup and configured, which is designed as a federated monitoring solution that covers the Cloud-Edge-IoT continuum, thus supporting the resource monitoring and management needs of the project. In the following period the PPEF will focus on the acquisition of the metrics that will be utilized to capture the NEMO-hosted services and application performance addressing the requirements set in SLA agreements. At the same time, the PPEF development will proceed and preliminary integration activities within NEMO meta-OS will be established. Lastly, additional open source tools and technologies will be investigated and potentially incorporated into the PPEF component.

4 Cybersecurity & Digital Identity Attestation

NEMO incorporates a set of modules to provide high levels of security and privacy which are essential for an effective OS, which will mainly support efficient Identity Management, Access Control Management and Intercommunication Security

4.1 Overview

The access control framework within NEMO will deploy an Identity and Access Management (IAM) system, designed to enforce precise access rights for users/groups of users to designated resources. This objective is realized through the synergistic operation of the two principal facets of the IAM system:

- Identity Management, which handles mainly the establishment and removal of user identities and manages the provisioning and de-provisioning processes.
- Access Management, which is responsible for tasks encompassing authentication, authorization, and policy management, ensuring that only authorized users can access specific resources.

The access control management system within NEMO relies on control policies. These policies establish the guidelines and criteria governing the allocation or refusal of access privileges and permissions to users, resources, or functionalities within the system. The security, confidentiality, integrity, and accessibility of sensitive information and resources of NEMO are based on those policies.

The intercommunication security module of NEMO is based on a message broker, serving as the central kernel component enabling communication and synchronization among distributed systems and applications. This message broker acts as an intermediary, facilitating message exchange while offering essential capabilities like message routing, queuing, and transformation. By adopting asynchronous communication patterns, message brokers promote loose coupling between message senders and receivers, enabling them to function independently and asynchronously.

4.2 State-of-the-art

The following section explains the state-of-the-art regarding the Cybersecurity and Digital Identity Attestation Technologies

4.2.1 State-of-the-art for Identity Management and Access control

Identity Management and Access Control are critical components of modern digital security systems, ensuring that only authorized individuals or entities gain access to protected resources. Keycloak[23]is a state-of-the-art, open-source solution that addresses these essential aspects of security by providing a comprehensive and flexible identity management and access control platform. To understand the significance of Keycloak within the NEMO project, it is crucial to delve into the background and current state of identity management and access control.

Background:

The Evolution of Identity Management: In the early days of digital systems, identity management primarily relied on basic username and password combinations. As technology advanced, more complex systems emerged, including LDAP directories, Single Sign-On (SSO), and multi-factor authentication (MFA) [15]. These solutions were effective but often fragmented, making integration and management challenging.

Challenges in Access Control: The growing complexity of digital ecosystems, including web applications, APIs, and cloud services, has made access control a complex challenge. Traditional approaches were often unable to cope with dynamic access requirements and lacked centralized administration. Emergence of Identity and Access Management (IAM): The need for unified and comprehensive solutions led to the emergence of Identity and Access Management (IAM) systems. These systems aimed to provide a unified approach to identity management, access control, and security policy enforcement. However, many commercial IAM solutions were costly, complex, and not easily accessible to smaller organizations.

Keycloak[26]

Keycloak, developed by Red Hat, is an open-source IAM solution that has gained significant attention and adoption in recent years. It offers several key features and benefits which fully address the identity and access management requirements of NEMO as listed in D1.2 while it reflects the state of the art in identity management and access control, making it an ideal choice for NEMO:

1. Single Sign-On (SSO): Keycloak provides SSO capabilities, allowing users to access multiple applications with a single set of credentials. This not only enhances user experience but also strengthens security by reducing the reliance on passwords.
2. Identity Federation: It supports identity federation, enabling the integration of external identity providers, like social media accounts, enterprise directories, or third-party IAM systems. This simplifies user authentication and access management.
3. User Authentication: Keycloak supports a wide range of authentication methods, including MFA, biometric authentication, and OAuth2/OpenID Connect. These mechanisms enhance security by providing adaptive and context-aware authentication.
4. Role-Based Access Control (RBAC): Keycloak offers RBAC, enabling organizations to define fine-grained access policies based on roles, attributes, and permissions. This granular control over access ensures security and compliance.
5. Adaptive Access Control: Keycloak incorporates adaptive access control policies, allowing for real-time risk assessment and adaptive responses to security threats, based on user behaviour and contextual information.
6. Open Source and Community-Driven: Keycloak's open-source nature and a thriving community ensure that it is continuously evolving to meet the changing needs of the industry. This makes it a cost-effective and future-proof solution.
7. Scalability and Integration: Keycloak is designed for scalability and can be easily integrated into various platforms, including web applications, mobile apps, and microservices. It also supports integration with modern container and orchestration technologies like Docker and Kubernetes.

In summary, within the context of the NEMO project, Keycloak represents the state of the art in identity management and access control. It addresses the evolving challenges of security in a digital world by providing a comprehensive, open-source, and adaptable solution. With its emphasis on SSO, identity federation, robust authentication, and adaptive access control, Keycloak is a crucial component of modern security infrastructures, enabling organizations to protect their digital assets and ensure the right individuals have the right access.

4.2.2 State-of-the-art for Intercommunication Security

A Message Broker is a software component that operates as a middleware solution, facilitating communication and data exchange among heterogeneous applications, systems, and services. Its primary function involves the mediation of messages, converting them between distinct formal messaging protocols. This enables disparate services to establish direct communication channels, overcoming language and platform disparities.

Message brokers are essential components of Messaging Middleware or Message-Oriented Middleware (MOM) systems. This middleware category provides to developers a standardized framework for managing data flow between the different components of an application while they serve as a distributed communication layer, enabling seamless interaction among applications deployed on different platforms.

Message Brokers include a comprehensive array of functions, encompassing message validation, persistent storage, intelligent routing, and guaranteed delivery to their designated endpoints. Acting as intermediaries, they enable message senders to dispatch messages without any knowledge of recipient

[26] https://www.keycloak.org/

specifics such as location, availability, or quantity. This decoupling of processes and services enhances system scalability and maintainability.

The key Technical Concepts in Message Brokers are the following:

1. Producer: The Producer represents an originating endpoint responsible for emitting diverse data types, all of which are destined to be processed and distributed by the message broker.
2. Consumer: On the opposing end, a Consumer designates a recipient endpoint that actively requests data, typically in the form of messages, from the message broker.
3. Queue: In the context of message brokering, a Queue functions as a data structure optimized for First In, First Out (FIFO) message storage and retrieval.
4. Exchange: An Exchange assumes a higher-level role within the message broker's architecture. It configures and oversees the creation of message routing rules, effectively governing the creation of communication groups to which consumers and producers can publish or subscribe for message transmission.

Message Brokers support two main message distribution patterns (also referred to as messaging styles):

1. Point-to-Point Messaging
2. Publish/Subscribe Messaging

4.2.2.1 Point-to-Point messaging

This distribution pattern corresponds to the utilization of message queues, wherein a sender and receiver establish a strict one-to-one relationship. In this paradigm, every individual message within the queue is exclusively directed to and consumed by a single recipient. Point-to-point messaging is mainly utilized when a given message triggers an action precisely once.

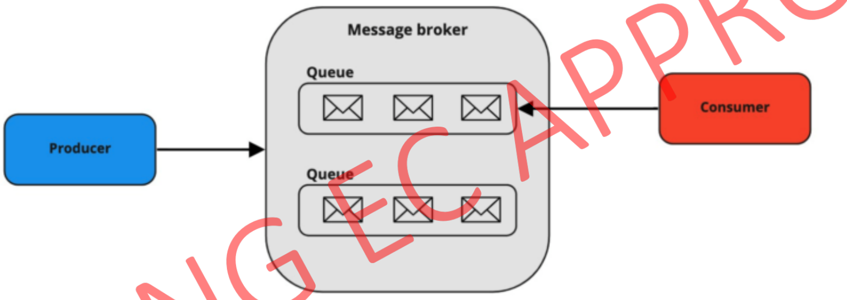


Figure 13: Point-to-Point Messaging Overview

4.2.2.2 Publish/Subscribe Messaging

In the context of message distribution, this pattern, commonly referred to as "publish/subscribe" or "pub/sub," is characterized by the message producer disseminating each message to a specific topic, with multiple message consumers selectively subscribing to topics for message reception. Within this paradigm, all messages posted to a given topic are broadcasted to every application or entity that has subscribed to that particular topic. Consequently, this architecture embodies a broadcast-style distribution mechanism, establishing a one-to-many relationship between the message publisher and several consumers.

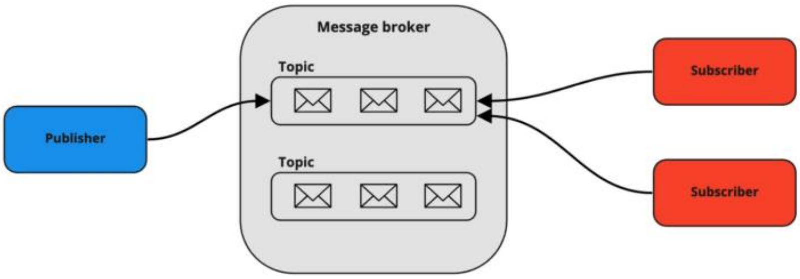


Figure 14: Publish/Subscribe Messaging Overview

4.2.2.3 Widely used message brokers

Message brokers have several advantages such as:

1. Producer Persistence: Independent of the consumer's liveness state, message producers can transmit messages with the sole prerequisite of a functioning message broker, thus the consumer's operational status does not affect the sender.
2. Elevated Asynchrony: Message Brokers introduce asynchronous processing, optimizing system performance by subdividing tasks into discrete processes. This allows for more efficient application execution.
3. Augmented Message Reliability: Message Brokers augment system dependability by supporting rigorous message transmission assurance. They incorporate mechanisms for immediate or scheduled message redelivery following consumer failures. Additionally, they facilitate management of non-deliverable messages via a dead-letter mechanism, significantly increasing message routing resilience.

As a result there are numerous message brokers that have been introduced, the most important of which are the following:

Amazon SNS (Simple Notification Service)[27]

Amazon SNS is a cloud-based service designed to orchestrate the delivery of push notifications from software applications to subscribed endpoints and clients. It facilitates direct notification delivery to customers and supports both individual message delivery and a publish-subscribe pattern. It's an integral component of Amazon Web Services (AWS), notable for its cost-effectiveness and automated workload scaling capabilities.

Amazon SQS (Simple Queue Service)[28]

Amazon SQS is a fully managed message queuing service tailored for decoupling and scaling microservices, distributed systems, and serverless applications. It can dynamically adapt to workload sizes. Amazon SQS employs a pull mechanism, requiring message receivers to autonomously retrieve messages from SQS queues. This service simplifies the complexities of managing message-oriented middleware, allowing developers to focus on innovation. It supports seamless message transmission, reception, and storage between software components at any scale without message loss or dependency on additional services.

Redis[29]

Redis is an in-memory data store claiming that it provides high-performance while serving as both a key-value store and a message broker. It operates as an in-memory data structure store, though it does

[27] https://aws.amazon.com/sns/
[28] https://aws.amazon.com/sqs/
[29] https://redis.io/

not guarantee message durability. Redis supports various abstract data structures, including strings, lists, maps, sets, sorted sets, hyperlogs, bitmaps, and spatial indexes.

Apache Kafka[80]

Apache Kafka represents a robust queue broker and open-source messaging system designed for distributed event streaming. It can efficiently handle high message volumes and stores messages on disk to ensure durability. Kafka also supports seamless message transmission from one point to another. Messages within the Kafka platform are replicated across the entire cluster, safeguarding against data loss. Kafka's primary focus is on real-time event streaming, data pipelining, and efficient data replay for swift and scalable operations.

RabbitMQ[45]:

RabbitMQ stands as the most widely adopted and popular open-source message broker software. Implemented in Erlang and supported by the Pivotal Software Foundation, it provides a standardized platform for processing modules to send and receive messages securely. RabbitMQ can handle intricate routing scenarios and offers four types of exchanges, which are very important in message routing. Unlike other message brokers, RabbitMQ first routes messages to exchanges, which then direct them to appropriate queues. RabbitMQ's main advantages are its exceptional performance, reliability, high availability, clustering, and federation capabilities. It also offers a user-friendly management interface for monitoring and controlling the message broker.

4.2.3 State-of-the-art and background for CNAPPS

This section describes what Gartner[30] calls a CNAPP (Cloud-Native Application Protection Platform). CNAPPs are assemblies of frameworks and tools, which provide security features at all steps of an application life cycle: development, release delivery and runtime. To understand what a CNAPP is, the following subsections give an overview on some CNAPP components such as Linux kernel monitoring probes. First, eBPF is introduced as the enabler of system monitoring features. Then, frameworks that use eBPF are described, because they are used by CNAPPs solutions. The concept of CNAPPs is then explained.

4.2.3.1 eBPF - Extend the Capabilities of the Linux Kernel without Kernel Modules

eBPF[31] is one technology that enable system monitoring in CNAPPs solutions. eBPF is a cutting-edge technology that allows for the execution of sandboxed programs within the Linux kernel without the need to modify the kernel's source code or add additional modules. eBPF does not stand for anything anymore and is a standalone term, but it was previously known as (extended) Berkeley Packet Filter.

eBPF technology provides a safe and efficient way to enhance the kernel's capabilities at runtime. As a result, there has been a surge in eBPF-based projects that address a wide range of use cases, such as advanced networking, system observability, and security features.

For example, eBPF can be used for container runtime monitoring. CNCF Projects like Falco, Tetragon[32] or Cilium[33] uses eBPF to monitor kernel events or network traffic. eBPF monitoring agents can be used in a Kubernetes environment to monitor the Kubernetes node host itself and containers and pods running on this Kubernetes node.

eBPF can also be used to implement other features than system or network monitoring. For example, the project LoxiLB[34] is an eBPF based cloud-native load-balancer for 5G Edge. A list of opens source projects that use eBPF is available on the eBPF website[35].

[30] https://www.gartner.com/reviews/market/cloud-native-application-protection-platforms
[31] https://ebpf.io/what-is-ebpf/
[32] https://github.com/cilium/tetragon
[33] https://cilium.io/
[34] https://github.com/loxilb-io/loxilb
[35] https://ebpf.io/applications/

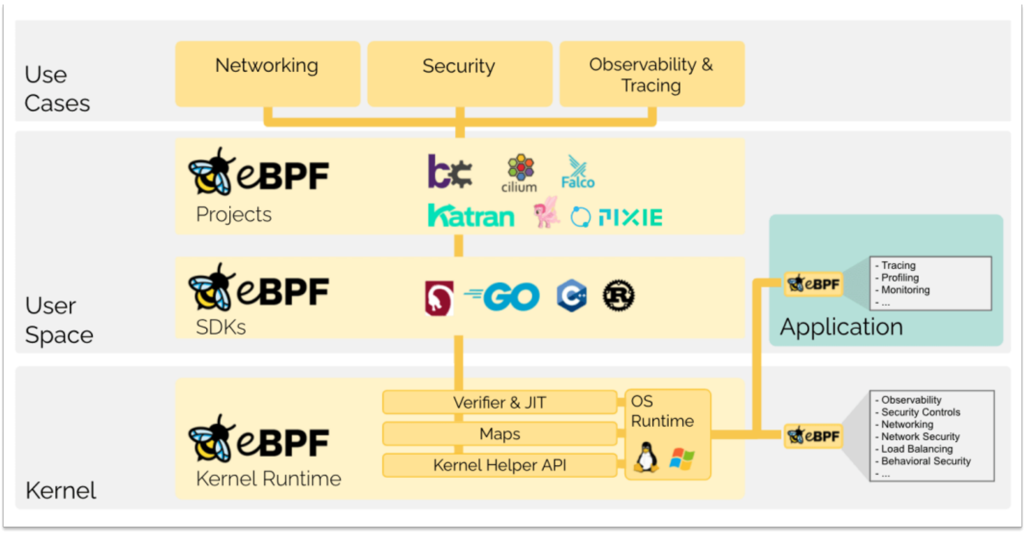


Figure 15: Overview of eBPF and frameworks using eBPF

It is worth mentioning that a developer or a DevSecOps does not need an extensive knowledge of the eBPF framework to use it. Indeed, the eBPF feature can be used by higher-level tools, like Cilium or Falco[36], which will leverage eBPF and an eBPF plugins. From the developer and DevSecOps point of view, the complexity of the eBPF plugin used by tools such as Falco is transparent.

4.2.3.2 State of the Art Linux Kernel Monitoring and Detection Agent

Several projects and frameworks propose to build tools on top of eBPF to leverage eBPF features but masking complexity to its users. This section describe the landscape of these tools.

Before introducing the concept of CNAPP[37] in the next section, it is useful to describe modern Linux kernel monitoring solutions. Indeed these Linux kernel monitoring solutions or probes are often at the center of the CNAPP solutions for the runtime monitoring as described in the next section.

Figure 16 below is a screenshot of the Cloud-Native Computing Foundation or CNCF Landscape, which is a taxonomy of all the projects related to the CNCF. Figure 16 focuses on the Security frameworks.

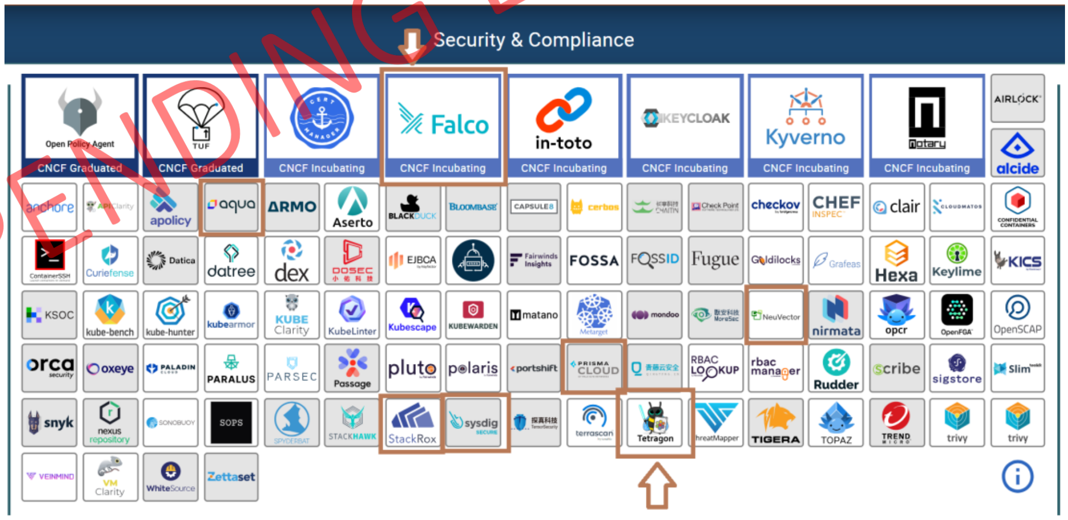


Figure 16: CNCF Landscape[38] - Focus on Cybersecurity frameworks - CNAPP and Linux monitoring probes are highlighted (as of September 2023)

[36] Falco can use both an eBPF probe or a Kernel module as a driver to access syscalls and kernel events. See the dedicated Falco section.

[37] CNAPP Cloud-Native Application Protection Platform

[38] https://landscape.cncf.io/ (screenshot from 2023-09-12)

Figure 16 and Table 1 are non-exhaustive lists of CNAPP vendor solutions. A more exhaustive market analysis is available on Gartner's website [16]. The NEMO project will not perform its own CNAPP market analysis.

Name	Logo	Comment
Sysdig	sysdig	Vendor solution. Its probe agent (eBPF based) is opensource and called Falco.
Isovalent	ISOVALENT	Vendor solution. Other solutions from Isovalent remain open source like Cilium (CNI), Hubble (cloud-native troubleshooting tool) and Tetragon (eBPF based monitoring agent).
SUSE Neuvector	SUSE NeuVector Full Lifecycle Container Security	Vendor solution. Also the vendor of Kubernetes Rancher distributions and Suse OS distributions.
PaloAlto Network Prisma Cloud Defenders	paloalto® NETWORKS	Vendor solution. The solution is mainly a SaaS. It will not be suited for disconnected and airgaped environments.
Aqua Security	aqua	Vendor solution. Most of their tools for benchmarks, SAST, DAST and runtime monitoring remain Opensource.

Table 1: Vendor CNAPP Solutions (not exhaustive)

The purpose of this document is not to have a holistic view of the CNAPP business and market. Therefore, the next paragraphs focus on two different communities, which are responsible for two open source Linux kernel monitoring solutions: Falco and Cilium.

Cilium's parent company is Isovalent[39]. Cilium started as a container network interface with network observability feature. Recently in 2022, Isovalent started to work on Tetragon[40] , which is a Kubernetes-aware security observability, and runtime enforcement tool that applies policy and filtering directly with eBPF, allowing for reduced observation overhead, tracking of any process, and real-time enforcement of policies. Tetragon extends the network observability features already provided by Cilium to a broader Kernel events monitoring. Tetragon can be used as a standalone solution without Cilium[41].

Falco's parent company is Sysdig. Falco[17] is a cloud-native security tool designed for Linux systems. It employs custom rules on kernel events, which are enriched with container and Kubernetes metadata, to provide real-time alerts.

Falco and Cilium have in common that they are open source, and they can rely on eBPF plugins to monitor Linux kernel events. However, Cilium is not a direct competitor to Falco, but Tetragon is. The solutions have also in common that their respective companies provide vendor solutions with more features. Table 5 and Table 6 in section Annexes are example of the kind of differences that can exist between the open source community version of a kernel monitoring probe like Tetragon and its vendor counterpart with support.

For the NEMO project, Falco has the advantage to be more mature than Tetragon. However, the two competing projects are evolving quickly, and Tetragon has arguments when it comes to network related events, especially thanks to the experience with the CNI[42] Cilium.

[39] https://isovalent.com/

[40] https://tetragon.cilium.io/

[41] https://isovalent.com/blog/post/can-i-use-tetragon-without-cilium-yes/

[42] CNI Container Network Interface

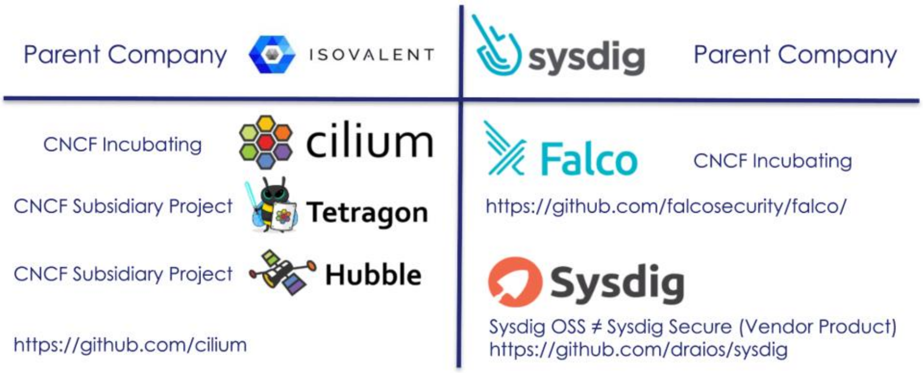


Figure 17: The 2 main Linux open source monitoring solutions and vendors (parent companies) at the CNCF level.

	Falco	Tetragon
License	Apache 2.0	Apache 2.0
CNCF Status	Incubating (production ready)	Subsidiary Project (spinoff project from Cilium) - not production ready yet (need Isovalent support)
Linux Kernel Driver	eBPF or Kernel Module	eBPF
Policy Rules Configuration Format	Custom YAML manifest with custom rule grammar (not a K8S CRD)	A Kubernetes CRD Custom Definition Ressource
Kernel Events Monitoring	YES	YES
SIGKILL	NO (needs Sysdig Secure)	YES
Network Protocol Awareness	++	+++

Table 2: Short comparison between 2 Linux Kernel Monitoring probes: Falco and Tetragon.

More details on how Linux kernel monitoring cyber probes works are available in the section below NEMO CNAPPS: Falco - Linux Kernel Monitoring Cyber Probe and Detection Agent.

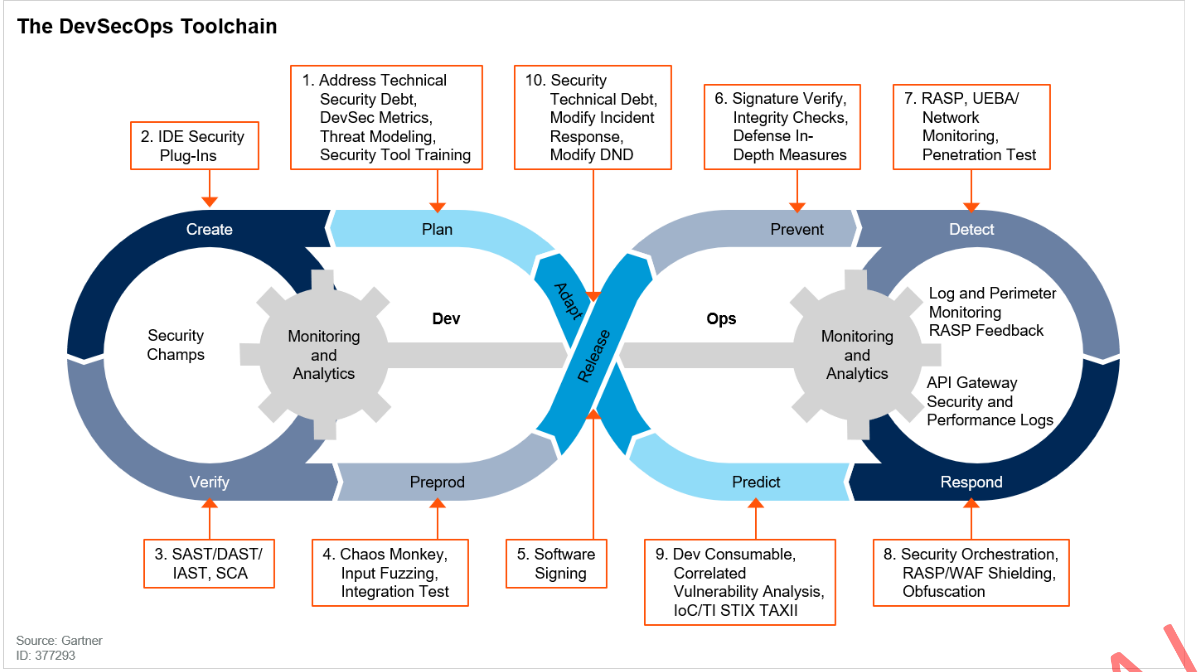
4.2.3.3 CNAPP - Cloud Native Application Protection Platform

CNAPP stands for Cloud-Native Application Protection Platform. CNAPPs are assemblies of frameworks and tools such as eBPF-based tools described in previous sections. The term is used by Gartner to identify solutions and frameworks that cover the full protection of a containerized or cloud-native application during its entire lifecycle. However, all CNAPP solutions are different, and their scope can vary from one solution to another. However, the main idea is that these solutions will be implemented and used during every phase of an application lifecycle.

Figure 18 describes a DevSecOps loop, which shows both the development time, the delivery release time and runtime phase of an application. CNAPP frameworks operates at every step of these three phases.

Within the DevSecOps toolbox, the following terms are used:

- SAST Static Application Security Testing: search for vulnerabilities during build or development time.
- DAST Dynamic Application Security Testing: search for vulnerabilities during runtime.
- RASP Run-time Application Security Protection: search for vulnerabilities during runtime and provides mitigations.
- IAST Interactive Application Security Testing: combine SAST and DAST.


The DevSecOps Toolchain
1. Address Technical Security Debt, DevSec Metrics, Threat Modeling, Security Tool Training
2. IDE Security Plug-Ins
3. SAST/DAST/ IAST, SCA
4. Chaos Monkey, Input Fuzzing, Integration Test
5. Software Signing
6. Signature Verify, Integrity Checks, Defense In-Depth Measures
7. RASP, UEBA/ Network Monitoring, Penetration Test
8. Security Orchestration, RASP/WAF Shielding, Obfuscation
9. Dev Consumable, Correlated Vulnerability Analysis, IoC/TI STIX TAXII
10. Security Technical Debt, Modify Incident Response, Modify DND
Create
Plan
Verify
Preprod
Adapt
Release
Prevent
Detect
Predict
Respond
Dev
Ops
Security Champs
Monitoring and Analytics
Monitoring and Analytics
Log and Perimeter Monitoring
RASP Feedback
API Gateway Security and Performance Logs

Source: Gartner
ID: 377293

Figure 18: Gartner DevSecOps Model[43] . Development on the left, delivery release on the middle, runtime on the right

Those terms SAST, DAST, RASP or IAST and the tools associated are not necessary working on container images only. However, applications are often packaged as container images and deployed thanks to Kubernetes manifests files.

To sum up, CNAPPs solution are an assembly of the following sub-components:

- **Runtime Kernel Monitoring Cyber Probe.** These probes can use eBPF plugins like described in previous sections or kernel modules to perform the monitoring. Example: Falco + Sysdig Secure, Cilium, Tetragon.
- **Container Image Vulnerability Scanning Tools.** SAST, DAST, RASP, IAST.
- **GitOps Interfaces, to be connected with CICD pipelines.**

Where & Why Should we Use CNAPP for IoT? One should understand that CNAPPs are a collection, an assembly of multiple components that intervene at different moment of an application lifecycle. On resources constrained devices like IoT, some components like the vulnerability scanner of a CNAPP solutions will be deployed on the development environment, and not on the IoT device itself. On the contrary other components like the cyber runtime monitoring probe will be deployed on the IoT device itself.

For the NEMO project, and especially for the NEMO Kernel and operating system, the focus will be on the **Runtime Kernel Monitoring Cyber Probe** rather than on SAST and vulnerability scanning.

4.3 Background

The underlying architectural patterns and design options within the meta-OS ecosystem are significant factors to be considered in the selection of appropriate security measures and modules to ensure secure communication among diverse workloads comprising or hosted in the developed meta-OS. In the cloud and edge native world of the NEMO meta-OS ecosystem, infrastructure and business logic are provided by a set of seamlessly integrated, interoperable and cloud native microservices. Core design principles for delivering such structure of functionality include the modular design for workloads' architecture and, largely, synchronous communication. Moreover, plugin-based architectures are becoming increasingly popular as flexible approaches for modular software frameworks. These architectural styles are briefly presented below.

[43] https://github.com/wurstbrot/DevSecOps-MaturityModel/issues/14

Modular Design

Modular Design is a system's architecture in which a complex system is composed of a set of independent, interchangeable modules plugged together. Each module is developed on the premise to be able to execute a single clearly defined aspect of the desired functionality; yet all modules function together as a whole. Modular design and programming in software development emphasises on separating a program's functions into independent pieces or building blocks, each containing all the parts needed to execute a single aspect of the functionality [17]. In a second step, putting the different modules together, the complete functionality is composed and implemented. Modular design refers to decomposing large and complex systems into smaller and more manageable parts. Modules provide the necessary methodology for abstracting arbitrary complexity, behind clean and simple interfaces. Thus, functionality can be enriched incrementally, as the project progresses.

Industry has experienced previously unimaginable levels of innovation and growth by embracing the concept of modularity. Modularity allowed designers to experiment with different approaches, as long as they obeyed the established design rules [18]. Three groups of elements are basically needed for the modular design realization:

- Modules
- Interfaces
- A set of protocols for interconnecting those modules

As long as these elements are defined and respected, the merits of modularity can be exploited in systems' development, integration, update and extensions with minimum overhead on the whole system. The components under development, test or experimentation can be plugged or unplugged with no or limited effects on the rest system's functionality.

Plug-in architecture

Aligned to modular architecture design, plug-in-based architectures address the need for separating application functionality into standalone functional units. As stated in [19] plugins are a cohesive, self-contained unit whose dependencies to other components and services are already predefined. A plug-in is a bundle that adds functionality to an application, called the host application, through some well-defined architecture for extensibility. This allows third-party developers to add functionality to an application without having access to the source code. This also allows users to add new features to an application just by installing a new bundle in the appropriate folder.

The plug-in architecture consists of two components: the core system and the plug-in modules.

In that respect additional features are added as plugins to the core application and such an approach provides extensibility, flexibility, and isolation of application features and customs processing logic.

The specific rules and the processing tasks are separate from the core system; thus, the designer can add, remove, and change existing plugins with little or no effect on the rest of the core system or the other plug-in modules.

Plugin architecture entails significant benefits for developers, as well as for the application/service users. The developers' benefits include:

- Reduced development and deployment time for new application features
- Deployment at runtime with no need to stop the host application
- Enhanced troubleshooting through feature isolation
- Increased application security through functionality isolation from the core functionality
- Opening development opportunities for third parties regarding additional features without requiring awareness of the source code of the host application or any interaction with the original application developer.
- Code language independence, as plugins can be written in different languages, wrapped with well-defined interfaces.

Synchronous communication

The adoption of the REST (REpresentational "State" Transfer)-ful architectural style is common in the modular microservices-oriented cloud/edge native world. REST APIs facilitate development, integration and testing, as outlined below:

- Greater support for interoperability: Through reliance on well-defined, universally understood communication interfaces, like the ones composing RESTful APIs, software components' implementation details are truly hidden from the client, providing real platform and language independence for developed code.
- Scalability: REST is stateless by design, as implied by the full term, as no state about the client session is stored on the server-side. Persistence and management of client session data is up to the client per se. Moreover, the uniform interface specification through universal Uniform Resource Identifiers (URI) does not require individual definitions of operations for every resource or server, allowing for using a multitude of tools compatible with HTTP and for uniform caching. Clients do not need to understand the URI structure, which means that awareness of just the URI link and the data format makes application processing, even after updates, possible.
- Data format: REST supports different data format for the provided response, e.g., JavaScript Object Notation (JSON), Extensible Markup Language (XML), Comma Separated Value (CSV). This provides flexibility to developers in parsing the response in the desired language or format that best suits their application/service.

The design approaches presented will be considered for addressing security in communications among the NEMO components and with external entities, applying the defined access control measures for the NEMO endpoints/interfaces.

4.4 Architecture & Approach

The overall architecture of the main Security Modules of NEMO from a user perspective is demonstrated in the following MAF (refer to D1.2) diagrams starting from the logical view diagram. The user will be assigned access to certain resources through the access management and privileges/permission sub-system while the intercommunication between the NEMO components will be handled and secured through the developed message broker which handles the intercommunication/network management of NEMO.

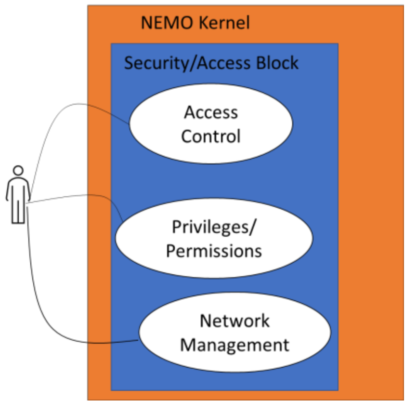


Figure 19 : Logical View on NEMO Security Modules

In order to provide the required functionality to the NEMO end user, the security modules will be interfaced with the NEMO microservices developed in WP2. Moreover, each block in the logical view is implemented in a corresponding NEMO module, while, in order to provide even higher levels of security an additional module which monitors all the services and module execution of NEMO so as to identify potential attacks is also being implemented (i.e. CNAPPS module). In the following figure the development view of the Security/Access block of NEMO is demonstrated.

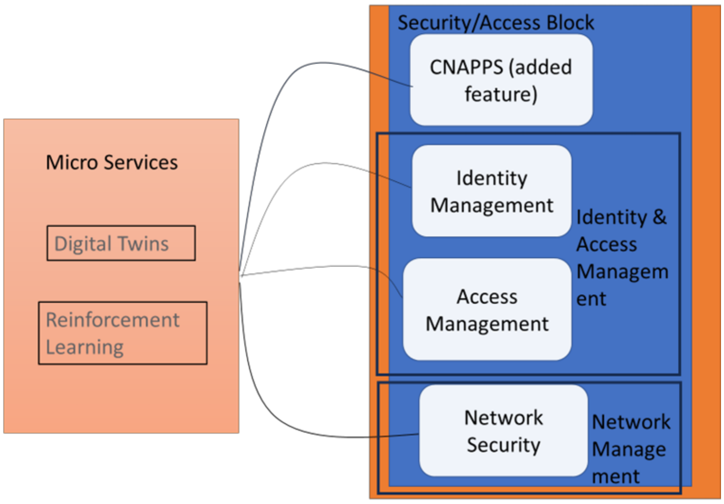


Figure 20: Development view of NEMO Security Modules

In order to be able to efficiently incorporate all those modules we have also worked on developing the process view of the NEMO block which is demonstrated below. The main internal (i.e. within the Security block) interfaces of the modules are the following : The Identity Management and Access control modules are combined into a single module, will provide the necessary meta-data which contain the user identity and the access privileges to the network management system which is responsible for the reliable and mainly secure intercommunication between the modules. Regarding the interface to the NEMO microservices, the security block will get requests for user creation/deletion/handling from them and will provide the meta-data needed for allowing the microservices to implement the required access policies to the different resource while also any access to the NEMO resources that is restricted is also handled by the security modules. The same block is also responsible for the secure and reliable intercommunication between the different microservices; in that respect each microservice module can send the required meta-date regarding the identity of the service as well as the unencrypted data to be sent and their destination and the interconnection management module of NEMO is responsible to send them encrypted to their destination; in the destination another instance of the interconnection management module is responsible to decrypt and send them to the corresponding microservice module.

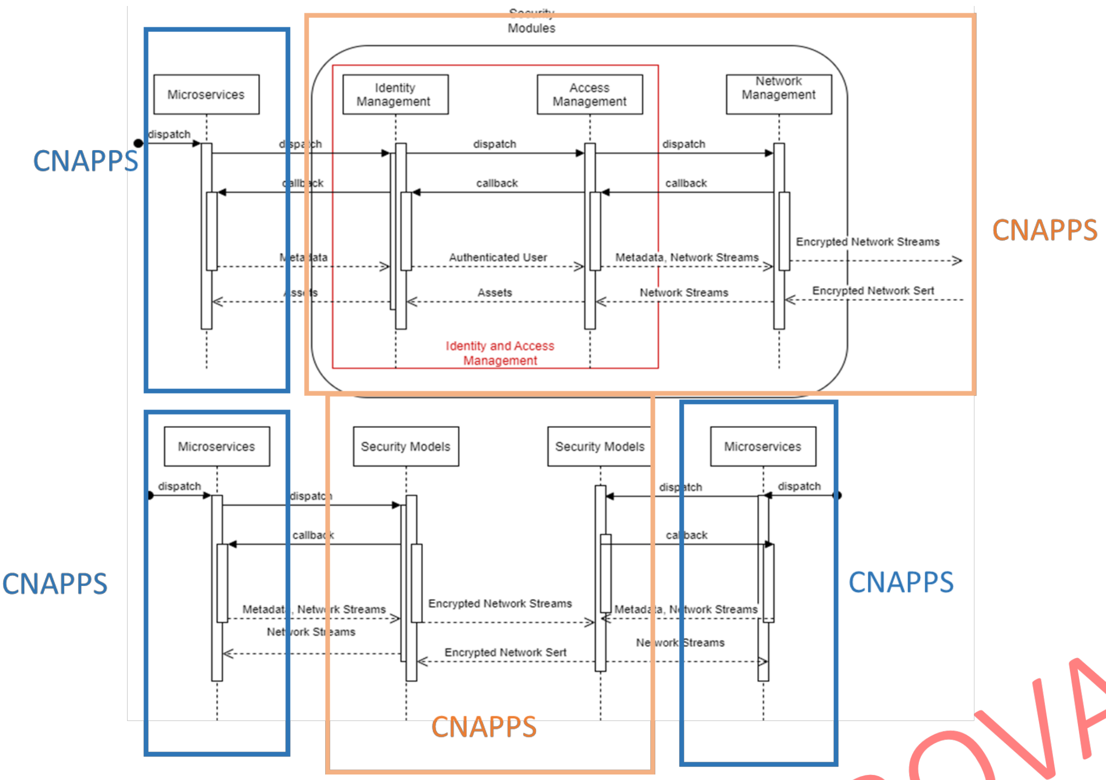


Figure 21: Process View of NEMO Security Modules

In terms of how the Security Blocks will be deployed, there will be deployed virtually in any NEMO device that requires secure communication and access the resources, while it will also be deployed in any possible centralized node (if there is such). This is required due to the fact that the developed module will handle the encrypted intercommunication as well as the local and global resource allocation/access. In the Figure 19 a simple deployment view of the Security Modules is demonstrated.

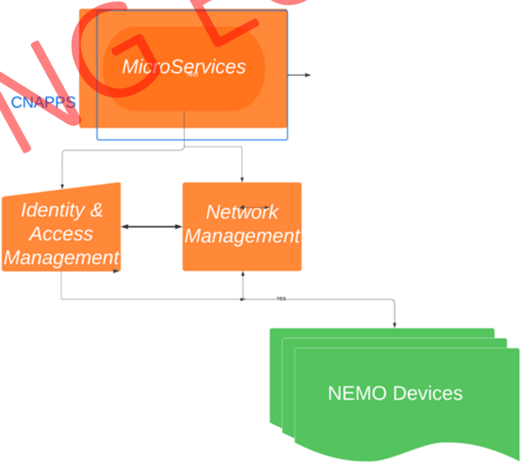


Figure 22: Physical view of NEMO Security Modules

4.4.1 Identity Management and Access control

The growing number of networked devices, connected across IoT, edge and cloud computing clusters, entails increasing and more comprehensive threats for cybersecurity of such systems. The vast number of diverse devices, as well as their integration in the IoT, edge and cloud continuum, has expanded the attack surface, with increasing numbers and types of vulnerabilities. Consequently, additional challenges arise for secure access in the integrated clusters' resources when such access is required both for external users and services and for core NEMO components.

Within the NEMO meta-OS, different types of functionalities are exposed through a set of programmatic endpoints, providing access to monitoring information, control actions, as well as configuration options in the meta-OS. NEMO considers a flexible mechanism for Access Control, aiming to secure these endpoints in a coherent manner across the continuum, facilitating security management while minimizing the attack surface. NEMO Access Control (NAC) allows to implement a comprehensive approach to applying flexible, easily configurable, granular privileged access to NEMO resources by either internal components or, beyond the perimeter, to external entities. NAC provides a security substrate to NEMO resources, enforcing that any attempted connection is brokered through a common API gateway. Then, access control is applied based on a set of modular criteria, which may include identity management, catering for Authentication, Authorization, and Accounting (AAA), but also traffic flow controls and universal controls for specific IPs through whitelist and blacklist rules.

Specifically, the NEMO Access Control (NAC) component aims to apply an access control mechanism which will only permit access to users and services based on their roles. Access will be provided only to the level of information or resources that are eligible for the relevant roles. In fact, the security administrator (in line with the “MetaOS Security Manager subrole of the MetaOS Provider user, as defined in D1.2 [2]) may define the access control criteria per endpoint and per user role.

NAC protects internet-exposed endpoints from unprivileged data breaches and is appropriate for the hybrid and highly diverse environment within the meta-OS. Access control is applied uniformly whether access is attempted by an external user (through any device type) or service or even an inherent NEMO workload.

In the following subsections, we delve into architectural description of the NAC module, following the NEMO Meta-Architecture Framework defined in D1.2; in particular, the development view and the process view with regards to NAC are presented.

Development & Process view

The defined NAC functionality will be offered through the modular design of the architecture depicted in Figure 23.

NAC design follows a modular plugin architecture, which allows isolation of the core functionality from separate features. The core functionality leverages an API Gateway, intercepting requests for accessing NEMO resources, which may be made by external users or third-party services or even other NEMO services.

Every access request is evaluated against a set of defined access control criteria. In NAC, such control criteria are implemented as plugins, attached to the core component (API Gateway). This allows NAC to apply chaining access control criteria, which may differentiate among endpoints or even user roles. In any case, AAA controls are applied, relying on Keycloak as the identity management solution. Once the evaluation check is successful and defined access control criteria are respected for both the endpoint in question and the entity requesting access (user, third-party or NEMO service), then the request is routed to the relevant NEMO endpoint. This leads to the request being processed and the relevant response to be communicated to the requester in a synchronous manner. In the opposite case, in which at least one control criterion fails, access is not granted, and the requester is provided with a relevant response text and code, indicating that access to the requested resource is forbidden. This process is depicted in the sequence diagram of Figure 24.

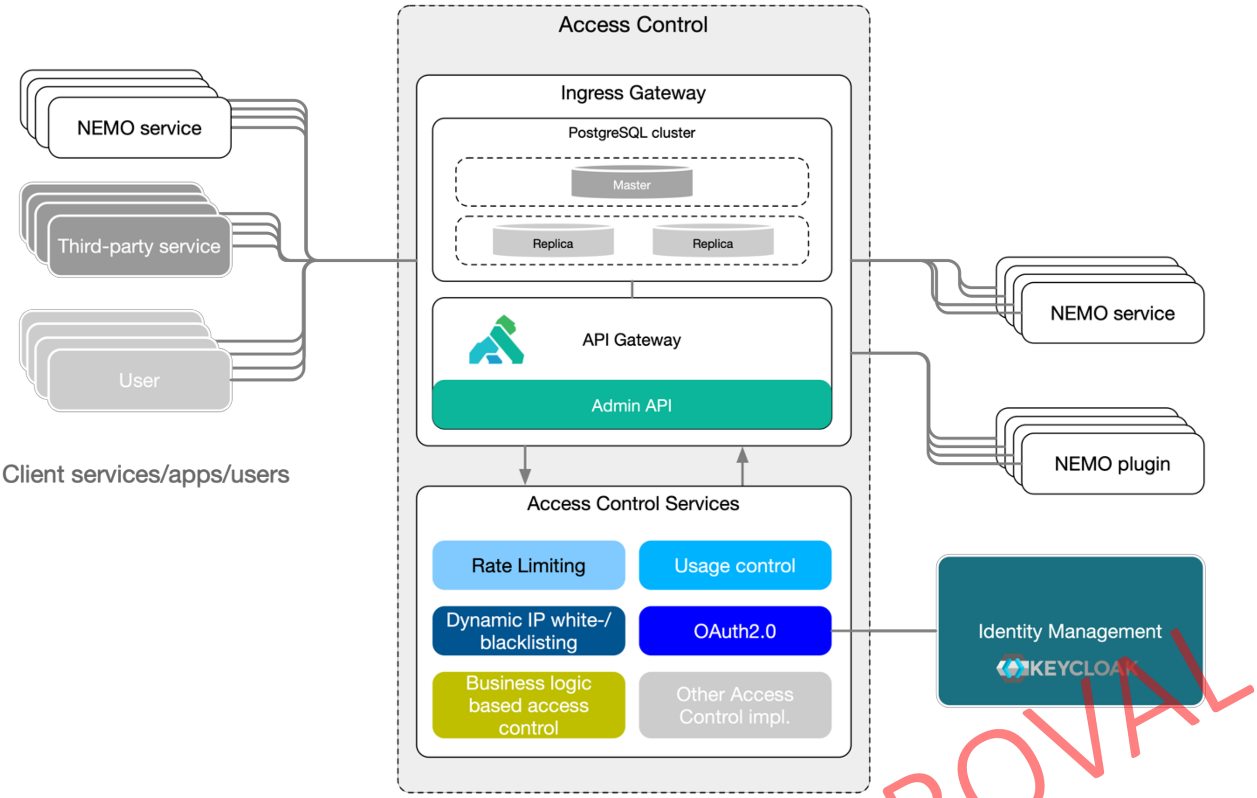


Figure 23: Development architecture for the NEMO Access Control

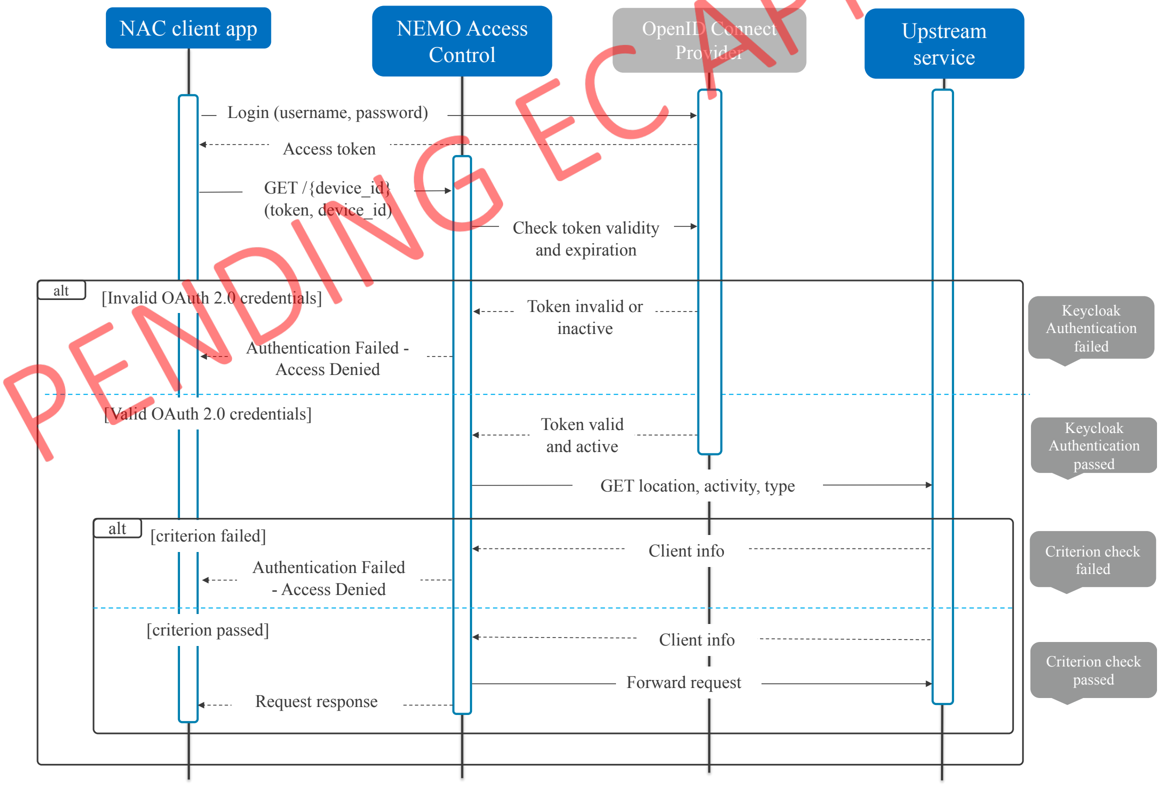

NAC client app
NEMO Access Control
OpenID Connect Provider
Upstream service
Login (username, password)
Access token
GET /{device_id}
(token, device_id)
Check token validity and expiration
alt
[Invalid OAuth 2.0 credentials]
Token invalid or inactive
Authentication Failed - Access Denied
Keycloak Authentication failed
[Valid OAuth 2.0 credentials]
Token valid and active
Keycloak Authentication passed
GET location, activity, type
alt
[criterion failed]
Client info
Authentication Failed - Access Denied
Criterion check failed
[criterion passed]
Client info
Forward request
Request response
Criterion check passed

Figure 24: Process diagram of NEMO Access Control operation

4.4.1.1 API Gateway

Description

An API gateway is an API management tool that sits between a client and a collection of backend services [20]. An API Gateway is a general concept that describes anything that exposes capabilities of a backend service[44]. In addition, it may support traffic routing and manipulation, such as load balancing, request and response transformation, and sometimes more advanced features like authentication and authorization, rate limiting, and circuit breaking.

In NAC, the API Gateway acts as a reverse proxy between the NEMO services and any workload or user requesting access to their resources.

Example of Technology Enablers

Envoy proxy [21] is an open source edge and service proxy, designed for cloud-native applications. Envoy is a self-contained process that is designed to run alongside every application server. All of the Envoys form a transparent communication mesh in which each application sends and receives messages to and from localhost and is unaware of the network topology.

Kong Gateway [22] is an API gateway built for hybrid and multi-cloud, optimized for microservices and distributed architectures. Kong provides a lightweight, fast, and flexible cloud-native API gateway. Kong Gateway runs in front of any RESTful API and can be extended through modules and plugins. It is designed to run on decentralized architectures, including hybrid-cloud and multi-cloud deployments.

Amazon API Gateway [23] is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale.

In addition, NGINX [24] offers several options for deploying and operating an API gateway depending on your use cases and deployment patterns, including both Kubernetes-native tools (NGINX Ingress, NGINX Service Mesh), as well as universal tools (NGINX Plus, F5 NGINX Management Suite API Connectivity Manager).

One advantage of using NGINX as an API gateway is that it can perform that role while simultaneously acting as a reverse proxy, load balancer, and web server for existing HTTP traffic [25].

Technology Chosen

In NEMO, the *Kong* open-source API gateway [26] will be employed as an Ingress/API Gateway, backed up by a simple (PostgreSQL) database cluster mostly used to keep track of the API gateway configured routes, services and upstreams.

Kong Gateway is a lightweight, fast, and flexible cloud-native API gateway. Kong Gateway facilitates developers' work in the following ways[45]:

- Leverage workflow automation and modern GitOps practices
- Decentralize applications/services and transition to microservices
- Create a thriving API developer ecosystem
- Proactively identify API-related anomalies and threats
- Secure and govern APIs/services and improve API visibility across the entire organization.

In addition, NEMO partners are already experienced in using Kong in a production environment. In such setup, the Gateway has been able to handle a vast number of requests per second, in the order of thousands, which indicates that Kong is appropriate in a highly demanding access control environment, within the meta-OS.

Interfaces

[44] https://gateway-api.sigs.k8s.io/

[45] https://crestsolution.com/solutions/kong-api-management-gateway/

The API Gateway interfaces with the Access Control services, acting as plugins implementing access control criteria, as well as with the Data Storage cluster, in which it stores administrative data regarding the access control configuration, e.g., routes and services.

Licensing

NEMO will leverage on the open source version of Kong, which is available through the Apache-2.0 license.

4.4.1.2 Access control services

Description

Rate limiting

Imposes a maximum threshold on the requests rate that can be performed by a client against the API Gateway, effectively preventing API misuse and, possibly, DOS attacks.

Usage control

Imposes policies related to the business logic of the NEMO core applications (e.g., related to the validity of the application requested, the existence of an active client registration etc.) to ensure proper, authorized use of the API, following the principles of use of the NEMO resources.

IP white-/blacklisting

Enforces traffic filtering based on IP whitelisting and blacklisting. When it comes to IP whitelisting, the plugin allows for stricter access control in the cases where a service exhibits stringent security requirements (e.g., in case system data are being exchanged) that prohibit the general public from accessing it. On the other hand, blacklisting is mostly relevant to blocking access to the NEMO resources by IPs that have been identified as possibly malicious (e.g., in case a DOS attack gets detected).

oAuth 2.0

In NEMO, the authentication and authorization services exposed by the oAuth 2.0 plugin are going to be used as primary access control mechanisms. It allows securing applications and services, based OpenID Connect [27].

This plugin relies on Keycloak [28] as the OpenID Connect Provider for supporting AAA services for NEMO resources. For accessing NEMO resources protected by this plugin, the requester entity must send a valid token along with the request. The plugin then communicates directly with the Keycloak instance and checks if the token is valid and has not expired in order to authorize access to the requesting entity.

Interfaces

The Access Control Services are implemented as Kong plugins. NEMO will leverage the Kong Gateway support for Python plugin development, provided by the kong-python-pdk library. The library provides a plugin server and Kong-specific functions to interface with Kong Gateway [22].

Licensing

The Access Control Services will be made available as open source. The exact license will be chosen when the software releases will be made public.

4.4.2 Intercommunication Management / Security

The network management and security module of NEMO will be based on RabbitMQ[46] which is an open-source message broker software that implements the Advanced Message Queuing Protocol (AMQP). It enables applications to communicate with each other by sending and receiving messages. The main components of RabbitMQ are producers, consumers, queues, and exchanges.

RabbitMQ consists of the producer, consumer, and broker. A producer is an application that sends messages to a RabbitMQ broker. The producer creates a message and sends it to an exchange within the

[46] https://www.rabbitmq.com/

broker. The message contains information that the broker uses to route it to one or more queues. The producer doesn't need to know anything about the consumers, only the exchange it's sending messages to. A consumer is an application that connects to the broker and subscribes to a queue to receive messages. The broker pushes messages to consumers when messages are available. Consumers either acknowledge each processed message or set up an 'auto-acknowledge' mode. The broker is a RabbitMQ server which receives messages from producers and pushes them to queues. The broker is responsible for routing messages, based on the exchange type and bindings. It also takes care of tasks like persisting messages, managing acknowledgements from consumers, and more.

A queue is a buffer that stores messages. Queues are bound to exchange and receive messages from them. The messages stay in the queue until they are handled by a consumer. Queues have certain properties that can be defined, such as durability (should the queue survive a broker restart), auto-delete (should the queue be deleted when the last consumer unsubscribes), and exclusivity (used by only one connection and the queue will be deleted when that connection closes).

Subsequently, an Exchange is a message gateway to RabbitMQ. The distance each message has to travel depends on the type of exchange. In NEMO, topic exchange will be used. In a topic exchange, the message is routed to one or many queues based on a match between the routing key and the pattern used to bind the queue to the exchange. The routing key is a list of words, delimited by dots. The binding pattern can contain an asterisk ("*") to match any word in a specific position, or a hash ("#") to match zero or more words.

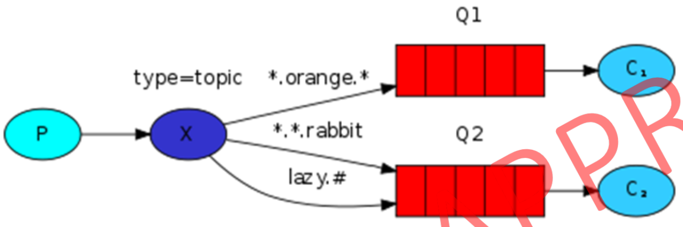


Figure 25: Topic Exchange of RabbitMQ

Looking at the security provided by RabbitMQ, since this is one of the main reasons that it will be incorporated and optimized within NEMO, it offers several security features and mechanisms to safeguard message communication and data integrity within its messaging infrastructure. These security provisions are essential for protecting the confidentiality, integrity, and availability of messages and the overall system. Below is a brief description of the security features that are provided by RabbitMQ and will be incorporated in NEMO:

Authentication:

External Authentication: It also supports external authentication methods like LDAP, OAuth, or other custom authentication mechanisms, allowing integration with existing user directories and single sign-on (SSO) solutions.

Authorization:

a. Fine-Grained Access Control: RabbitMQ enables fine-grained access control by allowing administrators to define access permissions for users and applications down to the level of individual queues, exchanges, and virtual hosts.
b. Role-Based Access Control: Users can be assigned to roles with specific permissions, simplifying access management and ensuring that users only have access to the resources necessary for their tasks.

Encryption:

RabbitMQ supports SSL/TLS encryption for securing data in transit between clients and the message broker. This ensures that messages and sensitive data are protected from eavesdropping during transmission.

Access Control Lists (ACLs):
ACLs in RabbitMQ allow administrators to define specific rules that govern access to resources, queues, exchanges, and management operations, providing granular control over who can perform what actions.
Pluggable Authentication Mechanisms
RabbitMQ's pluggable architecture enables the use of custom authentication and authorization plugins, allowing organizations to implement their security mechanisms if necessary.
Audit Logging
RabbitMQ can log security-related events and activities, helping administrators monitor and audit the system for security compliance and potential threats.

For example, to enable secure communication, NEMO will take full advantage of RabbitMQ's access controls[47] and best practices. More specifically, everything will happen within a vhost, and will block the generic vhost entirely. Continuing, we will create unique users, and the default guest user will be removed. Each user will have their own username and password which should be used only by them. Lastly, each user will be permissioned to only read and write to their own queue based on the queue's unique name.

4.4.3 NEMO CNAPPS: Falco - Linux Kernel Monitoring Cyber Probe and Detection Agent

As described in the State-of-the-art and background for CNAPPS (Section 4.2.3) Falco is a cloud native runtime security tool for Linux. It is designed to detect and alert on abnormal behaviour and potential security threats in real-time and it will be used in NEMO as a CNAPPs[48] monitoring agent and will interact with other NEMO components such as the message brokers.

Falco[17] is a kernel monitoring and detection agent that observes events, such as syscalls, based on custom rules. Falco can enhance these events by integrating metadata from the container runtime and Kubernetes. The collected events can be analysed off-host in SIEM[49] or data lake systems.

Figure 26 is the Falco architecture overview[50]. The Falco probes needs a driver to access the kernel events and syscalls, while it uses a number of different instrumentations to analyse the system workload and pass security events to userspace. The driver provides the syscall event source since the monitored events are strictly related to the syscall context. Falco supports several types of drivers which are the kernel event source[51]:

- a kernel module, which requires to add this kernel module to the Linux Kernel. This requires having full control and full privileges on your Linux kernel and OS.
- a classic eBPF
- probe, which is an alternative to the kernel module, but still needs to be compiled for each kernel version. In comparison, it allows for a least privileged mode usage.
- a modern eBPF probe, which is more transparent toward the Linux kernel (Kernel 5.8 and above) and OS than the classic eBPF probe, as it only needs that the kernel supports eBPF[52].

For NEMO, choosing between an eBPF probe driver and a kernel module will have an impact on how the NEMO Kernel and Linux OS is built. Indeed, either the NEMO Kernel and OS should embed the Falco kernel module, or it should embed the eBPF Probe. Pre-installing either the kernel module or the eBPF probe will guarantee that the Falco agent has everything it needs to perform its kernel monitoring mission.

Figure 27 is an illustration of a typical Falco kernel monitoring probe deployment within a Kubernetes cluster. In this particular example, which is generic and not dedicated to the NEMO architecture, the Kubernetes environment which is used is SUSE Rancher k3s[53]. k3s is a lightweight Kubernetes for edge

[47] https://www.rabbitmq.com/access-control.html
[48] Cloud Native Application Protection Platforms
[49] Security Information and Event Management
[50] https://falco.org/docs/getting-started/
[51] https://falco.org/docs/event-sources/kernel/
[52] https://falco.org/blog/falco-modern-bpf/
[53] Lightweight Certified Kubernetes Distribution | K3s | Rancher

devices. The operating system on the example are Fedora CoreOS, Flatcar or OpenSUSE Micro Leap which are immutable OSes. Please note that the NEMO project might choose another OS or another Kubernetes environment. However, the principle will be the same: Falco can run on a Kubernetes pod.

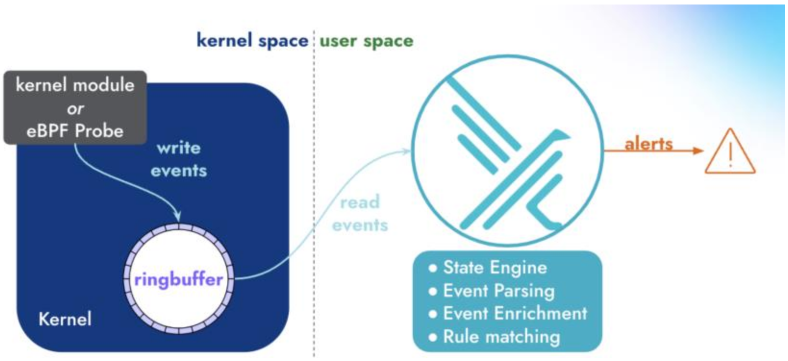


Figure 26: Falco Architecture Overview (taken from Falco documentation)

The examples on Figure 27 and Figure 28 were using virtual machines for k3s nodes. However, for NEMO, those k3s nodes could be running on an IoT or an Edge device. This would not change the way the Falco monitoring probe works.

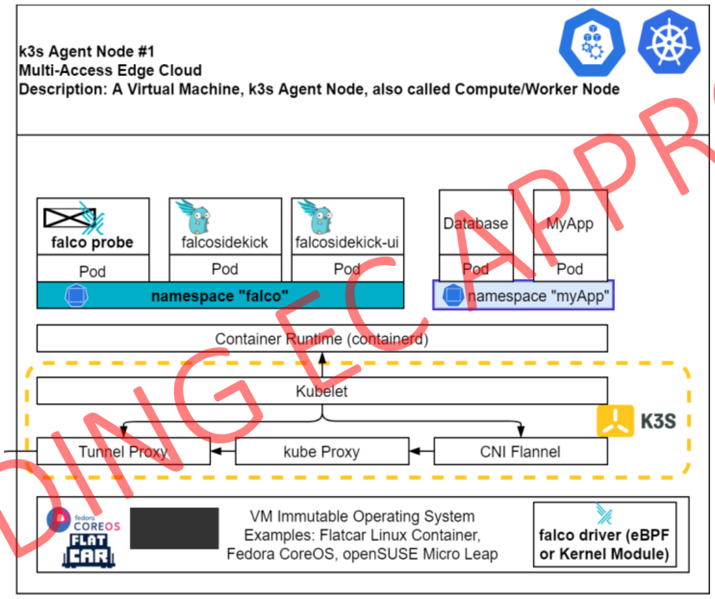


Figure 27: A typical Falco probe installation on a k3s Kubernetes Cluster's control plane node. The Operating System is an immutable OS (either Flatcar, Fedora CoreOS or openSUSE Leap Micro).

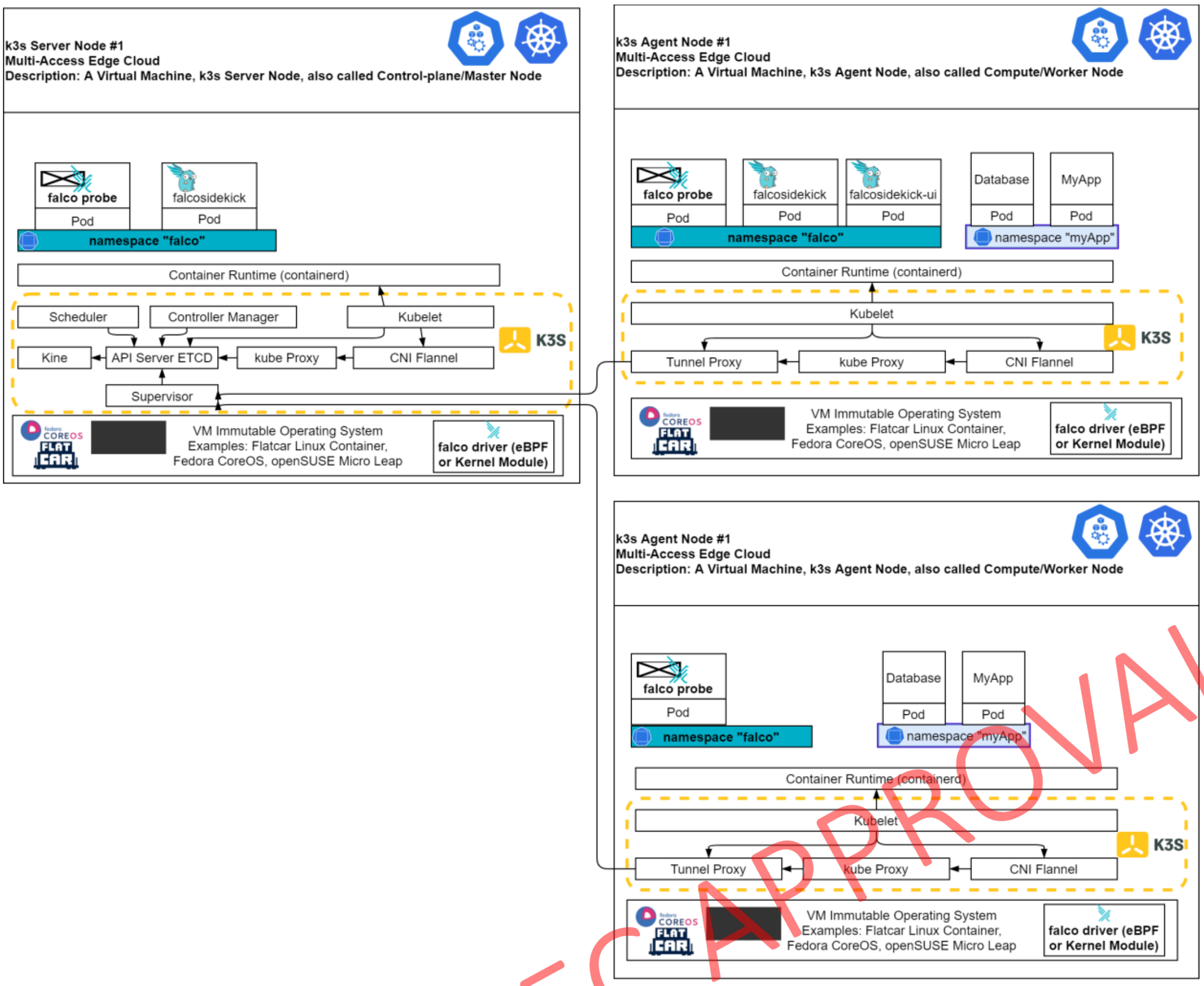

k3s Server Node #1
Multi-Access Edge Cloud
Description: A Virtual Machine, k3s Server Node, also called Control-plane/Master Node
falco probe
Pod
falcosidekick
Pod
namespace "falco"
Container Runtime (containerd)
Scheduler
Controller Manager
Kubelet
K3S
Kine
API Server ETCD
kube Proxy
CNI Flannel
Supervisor
fedora COREOS
FLATCAR
VM Immutable Operating System
Examples: Flatcar Linux Container,
Fedora CoreOS, openSUSE Micro Leap
falco driver (eBPF or Kernel Module)
k3s Agent Node #1
Multi-Access Edge Cloud
Description: A Virtual Machine, k3s Agent Node, also called Compute/Worker Node
falco probe
Pod
falcosidekick
Pod
falcosidekick-ui
Pod
Database
Pod
MyApp
Pod
namespace "falco"
namespace "myApp"
Container Runtime (containerd)
Kubelet
K3S
Tunnel Proxy
kube Proxy
CNI Flannel
VM Immutable Operating System
Examples: Flatcar Linux Container,
Fedora CoreOS, openSUSE Micro Leap
falco driver (eBPF or Kernel Module)
k3s Agent Node #1
Multi-Access Edge Cloud
Description: A Virtual Machine, k3s Agent Node, also called Compute/Worker Node
falco probe
Pod
Database
Pod
MyApp
Pod
namespace "falco"
namespace "myApp"
Container Runtime (containerd)
Kubelet
K3S
Tunnel Proxy
kube Proxy
CNI Flannel
VM Immutable Operating System
Examples: Flatcar Linux Container,
Fedora CoreOS, openSUSE Micro Leap
falco driver (eBPF or Kernel Module)

Figure 28: A 3 Nodes k3s cluster - 1 k3s control plane (k3s server) - 2 k3s compute nodes (k3s agent). Each Kubernetes node needs its own local kernel monitoring probe.

4.5 Interaction with other NEMO components

4.5.1 Identity Management and Access Control

Keycloak efficiently manages tokens, allowing for identities and secure data transmission between components. The interfaces of Keycloak are utilizing tokens, such as JSON Web Tokens (JWT), and serve as a bridge, conveying user authentication and authorization information throughout the system. Keycloak's token management ensures that data remains protected during transit, preserving the integrity of the application.

The NEMO Access Control component aims to protect RESTful APIs of NEMO components and plugins from unauthorized access, following Zero Trust approach. To this aim, NAC interacts with meta-OS hosted applications and services, other NEMO components and plugins, as well as with the Identity Management module, based on Keycloak. In more detail, meta-OS users, external -meta-OS hosted-applications and services or even NEMO components and plugins consume the NAC API in order to eventually access some NEMO service protected by NAC. On the other hand, the NAC API interacts also with the NEMO components and plugins providing the requesting services, in order to forward the requests to them, in case the requesting party is granted access. Moreover, NAC interacts with the Keycloak Identity Management in order to apply AAA services for each request made to the NAC API.

4.5.2 NEMO Intercommunication Security

Regarding the interaction of the message broker with the rest of the NEMO components it mainly consists of

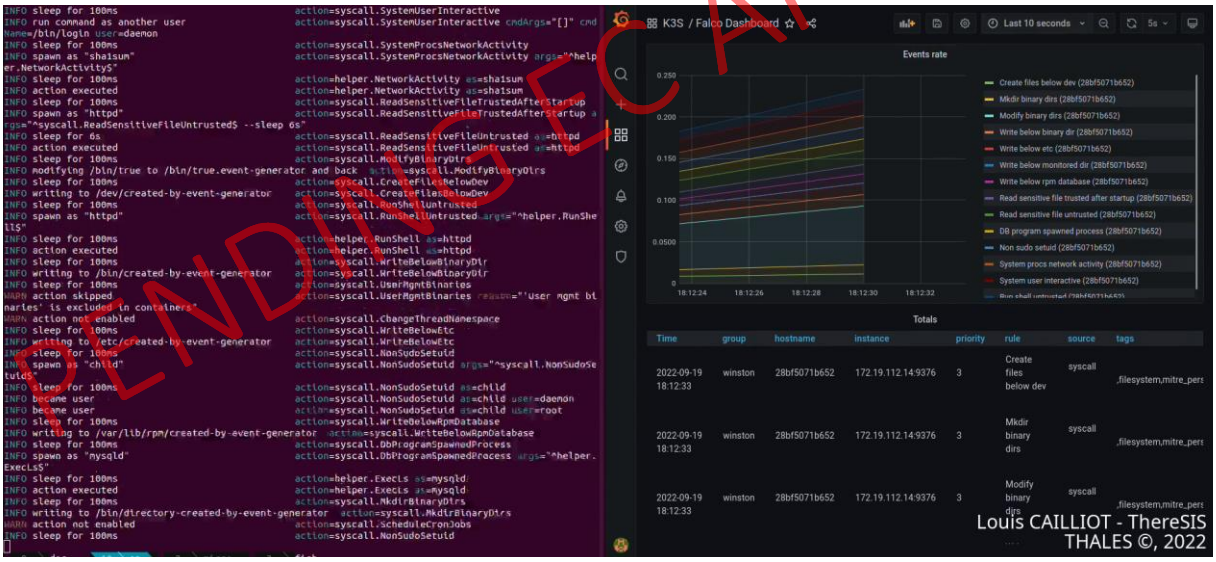
1. The input to the module from the other modules will be the metadata consisting of identity data and/or an encrypted key, the destination of the message and the actual message to be transmitted
2. The output of the module to the other modules, will be the metadata consisting of identity data, the source of the message and the actual message that has been received

Moreover, each instance of the message broker will be interconnected with one or more other instances of the same module (i.e., depending on whether there is a 1-to-1 or 1-to-many topology) to which it will send the encrypted data so as to be properly decrypted.

4.5.3 NEMO CNAPPS

CNAPPs features, and especially the kernel cyber monitoring feature, can interact with other NEMO components. Indeed, kernel cyber monitoring probes such as Falco can send kernel monitoring logs and events to other NEMO components. For example, in case of using Falco, there is a component called Falco Sidekick[54] which can send monitoring logs and event to other components[55], including Prometheus, Kafka and rabbitMQ[56]. Figure 30 illustrates as an example how to gather kernel monitoring logs and events to display them to a human operator on a Grafana Dashboard. Figure 29 shows the Grafana dashboard on the right with various kernel events triggered by a demo script and gathered by the Falco monitoring probe.

Figure 30 is an example of what could be integrated in NEMO: The kernel monitoring events detected by probes such as Falco can be sent to other services, like Prometheus (Figure 29), Graphana or messages brokers like RabbitMQ. This allows to gather monitoring kernel events from multiple nodes at one place. Other NEMO services could then leverage those events to trigger various responses.



```
INFO sleep for 100ms                                   action=syscall.SystemUserInteractive
INFO run command as another user                       action=syscall.SystemUserInteractive cmdArgs="[]" cmd
Name=/bin/login user=daemon
INFO sleep for 100ms                                   action=syscall.SystemProcsNetworkActivity
INFO spawn as "sha1sum"                                action=syscall.SystemProcsNetworkActivity args="^help
er.NetworkActivity$"
INFO sleep for 100ms                                   action=helper.NetworkActivity as=sha1sum
INFO action executed                                   action=helper.NetworkActivity as=sha1sum
INFO sleep for 100ms                                   action=syscall.ReadSensitiveFileTrustedAfterStartup
INFO spawn as "httpd"                                  action=syscall.ReadSensitiveFileTrustedAfterStartup a
rgs="^syscall.ReadSensitiveFileUntrusted$ --sleep 6s"
INFO sleep for 6s                                      action=syscall.ReadSensitiveFileUntrusted as=httpd
INFO action executed                                   action=syscall.ReadSensitiveFileUntrusted as=httpd
INFO sleep for 100ms                                   action=syscall.ModifyBinaryDirs
INFO modifying /bin/true to /bin/true.event-generator and back  action=syscall.ModifyBinaryDirs
INFO sleep for 100ms                                   action=syscall.CreateFilesBelowDev
INFO writing to /dev/created-by-event-generator        action=syscall.CreateFilesBelowDev
INFO sleep for 100ms                                   action=syscall.RunShellUntrusted
INFO spawn as "httpd"                                  action=syscall.RunShellUntrusted args="^helper.RunShe
ll$"
INFO sleep for 100ms                                   action=helper.RunShell as=httpd
INFO action executed                                   action=helper.RunShell as=httpd
INFO sleep for 100ms                                   action=syscall.WriteBelowBinaryDir
INFO writing to /bin/created-by-event-generator        action=syscall.WriteBelowBinaryDir
INFO sleep for 100ms                                   action=syscall.UserMgmtBinaries
WARN action skipped                                    action=syscall.UserMgmtBinaries reason="'User mgmt bi
naries' is excluded in containers"
WARN action not enabled                                action=syscall.ChangeThreadNamespace
INFO sleep for 100ms                                   action=syscall.WriteBelowEtc
INFO writing to /etc/created-by-event-generator        action=syscall.WriteBelowEtc
INFO sleep for 100ms                                   action=syscall.NonSudoSetuid
INFO spawn as "child"                                  action=syscall.NonSudoSetuid args="^syscall.NonSudoSe
tuid$"
INFO sleep for 100ms                                   action=syscall.NonSudoSetuid as=child
INFO became user                                       action=syscall.NonSudoSetuid as=child user=daemon
INFO became user                                       action=syscall.NonSudoSetuid as=child user=root
INFO sleep for 100ms                                   action=syscall.WriteBelowRpmDatabase
INFO writing to /var/lib/rpm/created-by-event-generator  action=syscall.WriteBelowRpmDatabase
INFO sleep for 100ms                                   action=syscall.DbProgramSpawnedProcess
INFO spawn as "mysqld"                                 action=syscall.DbProgramSpawnedProcess args="^helper.
ExecLs$"
INFO sleep for 100ms                                   action=helper.ExecLs as=mysqld
INFO action executed                                   action=helper.ExecLs as=mysqld
INFO sleep for 100ms                                   action=syscall.MkdirBinaryDirs
INFO writing to /bin/directory-created-by-event-generator  action=syscall.MkdirBinaryDirs
WARN action not enabled                                action=syscall.ScheduleCronJobs
INFO sleep for 100ms                                   action=syscall.NonSudoSetuid
```

K3S / Falco Dashboard
Last 10 seconds
5s
Events rate
0.250
0.200
0.150
0.100
0.0500
0
18:12:24
18:12:26
18:12:28
18:12:30
18:12:32
Create files below dev (28bf5071b652)
Mkdir binary dirs (28bf5071b652)
Modify binary dirs (28bf5071b652)
Write below binary dir (28bf5071b652)
Write below etc (28bf5071b652)
Write below monitored dir (28bf5071b652)
Write below rpm database (28bf5071b652)
Read sensitive file trusted after startup (28bf5071b652)
Read sensitive file untrusted (28bf5071b652)
DB program spawned process (28bf5071b652)
Non sudo setuid (28bf5071b652)
System procs network activity (28bf5071b652)
System user interactive (28bf5071b652)
Run shell untrusted (28bf5071b652)

Totals

Time	group	hostname	instance	priority	rule	source	tags
2022-09-19 18:12:33	winston	28bf5071b652	172.19.112.14:9376	3	Create files below dev	syscall	,filesystem,mitre_pers
2022-09-19 18:12:33	winston	28bf5071b652	172.19.112.14:9376	3	Mkdir binary dirs	syscall	,filesystem,mitre_pers
2022-09-19 18:12:33	winston	28bf5071b652	172.19.112.14:9376	3	Modify binary dirs	syscall	,filesystem,mitre_pers

Louis CAILLIOT - ThereSIS
THALES ©, 2022

Figure 29: On the left: a demo script that performs various improper Kubernetes use and attacks to generate events that the kernel monitoring probe will gather. On the right, a Grafana dashboard that shows the kernel events monitored by the Falco probe.

[54] https://github.com/falcosecurity/falcosidekick
[55] https://falco.org/docs/outputs/forwarding/
[56] https://github.com/falcosecurity/falcosidekick/blob/master/docs/outputs/rabbitmq.md

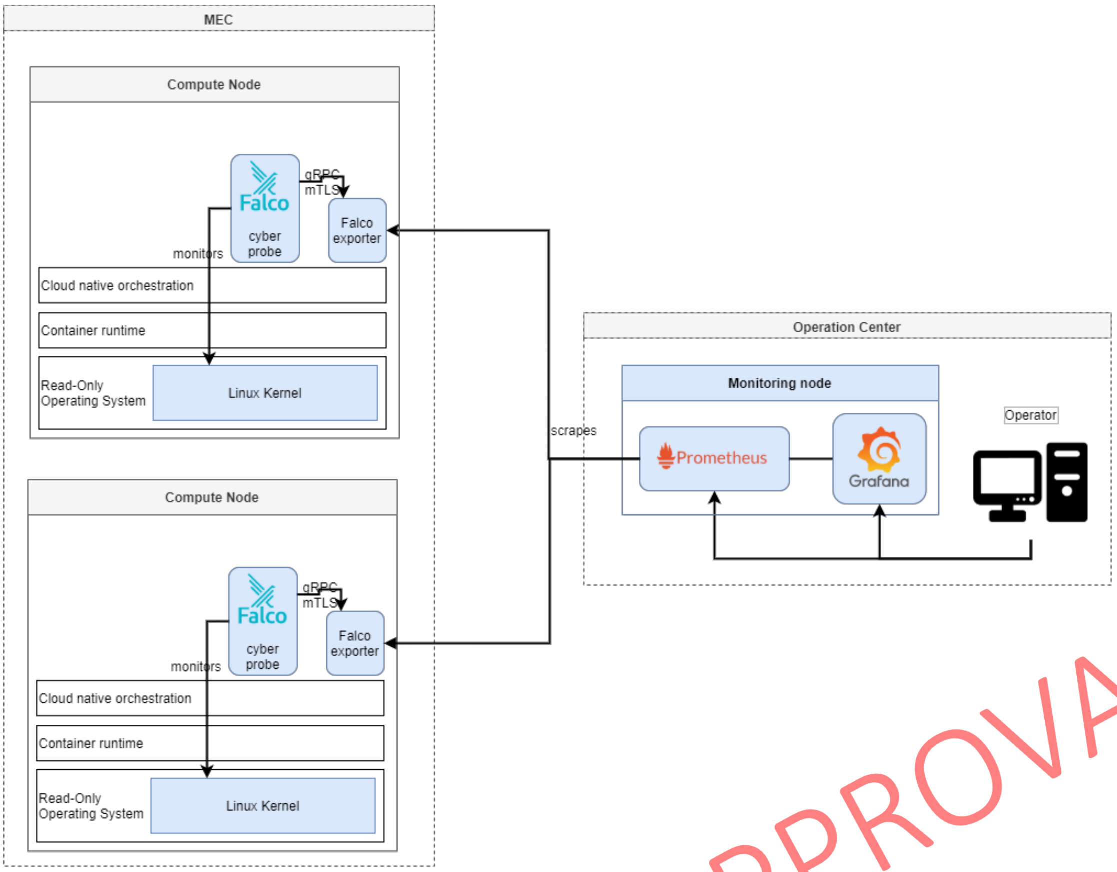


Figure 30: Kernel monitoring probes (ex: Falco) export events detected to Prometheus.

4.6 Conclusion, Roadmap & Outlook

4.6.1 Roadmap for identity Management and Access control

The current version of the Access Control module of NEMO protects HTTP endpoints, applying auth2.0 authentication and authorization services based on Keycloak as the OpenID Connect Provider. In the future versions of this component, additional plugins will be developed and integrated, on the basis of the design presented in section 4.4.1.2. Moreover, additional plugins may be considered, as a result of the security requirements' analysis of both NEMO services/plugins and Living Lab specific components as well as of new applications and services which may be integrated in NEMO through the Open Call projects.

4.6.2 Roadmap for Intercommunication Management

The NEMO intercommunication management module has been set up and tested as a standalone component within the 1st year of the project. For the next phases of the project, it will be tightly incorporated within NEMO. In order to achieve an efficient integration with the rest of the NEMO components several discrete, yet interconnected steps will be followed : a) the detailed security requirements for networking and intercommunication from each use-case/pilot will be analysed so as to as the implemented NEMO message broker will meet all of them b) the interfaces with all the NEMO sub-components will further be analysed so as to be optimized c) the overall message broker processes will be optimized for low-power and high levels of security.

4.6.3 Roadmap for CNAPPS

The NEMO project will benefit from Thales knowledge and involvement in the CNAPP market and from Thales links with the cloud-native Security industry.

The key challenge within NEMO is to use CNAPPs features such as kernel cyber monitoring in an edge IoT cloud continuum context. Indeed on low powered and resource constrained devices, every resource count, and cybersecurity features such as kernel cyber monitoring will add an overhead on the platform. It will be interesting to measure performances and resource consumption by cyber features on the NEMO testbed, especially since NEMO plan to leverage energy monitoring tools like Kepler or Scaphandre.

4.6.4 Conclusions

The 1st proof-of-concept for the cybersecurity and identity management sub-system of NEMO is already operational and it consists of the Identity Management and Access Control sub-system, the network management sub-systems and a CNAPPs implementation. The functionality of the first two sub-systems is described in the DoA and it satisfies a number of the requirements as stated in D1.1 and D1.2. Moreover, in order to provide higher levels of security and satisfy in an even more holistic manner the security requirements, stated in D1.1 the NEMO partners have decided to implement an advanced CNAPPS sub-system. An initial version of all three modules has been developed and functionally verified. In the next period the modules will be integrated, and their functionalities extended based on any limitations identified when the integrated system is evaluated and/or on the feedback received by the Living Labs when utilizing it in their applications.

5 NEMO meta-Orchestrator

The NEMO meta-Orchestrator is driven by the need for efficient decentralization in the IoT to Edge to Cloud Continuum. It simplifies the complexity of distributed computing, making it more manageable. Intelligence guides its decision-making, optimizing workflows for efficiency. Resource efficiency is a priority, especially in resource-constrained edge environments. Adaptability ensures scalability and reliability in a dynamic landscape. Interoperability is key, enabling seamless integration with diverse components and systems. Overall, the meta-Orchestrator addresses fundamental challenges and opportunities in this complex ecosystem, ensuring efficient workflow distribution and management.

5.1 Overview

In the intricate world of IoT to Edge to Cloud Continuum computing, the NEMO platform introduces a central orchestrator known as the meta-Orchestrator. This chapter provides a comprehensive exploration of the meta-Orchestrator's design, capabilities, and its pivotal role within the NEMO ecosystem.

The meta-Orchestrator stands as a testament to the innovative spirit of NEMO, embodying a vision of decentralized and efficient computing workflows across diverse domains. As computing environments become increasingly distributed and heterogeneous, the need for an orchestrator that can seamlessly navigate this complexity becomes paramount. The meta-Orchestrator is the answer to this challenge, offering a holistic approach to orchestration that considers the intricacies of the IoT, Edge, and Cloud domains.

This chapter unfolds the layers of the meta-Orchestrator, beginning with an overview that introduces its core purpose and place within the technology landscape. We then delve into the state-of-the-art principles that underpin its design, emphasizing intelligence, adaptability, and interoperability. The meta-Orchestrator within NEMO platform is explored, highlighting how this orchestrator empowers the meta-OS to thrive amidst the evolving dynamics of modern computing.

With each section, a deeper understanding of the meta-Orchestrator's significance and functionality will emerge, paving the way for an in-depth exploration of its capabilities and contributions to NEMO's mission.

5.1.1 State-of-the-art

In the fast-evolving landscape of distributed computing, remaining at the forefront of technological advancements is paramount to meet the ever-growing demands of modern digital ecosystems. The NEMO meta-Orchestrator proudly embraces state-of-the-art principles and trends, harnessing cutting-edge capabilities to tackle the intricate challenges associated with orchestrating workflows across the vast expanse of the IoT to Edge to Cloud Continuum.

This section aims to provide a clear and summarized overview of the most current advancements in line with the NEMO project's objectives and technology plans. Specifically, it describes a technical approach to implementing the NEMO meta-Orchestrator, considering the behaviours of existing and upcoming projects within the European Union (EU).

Based on previous project meetings, we have identified three main areas: EU projects, initiatives, and communities. Figure 31 displays the key topics identified in the current state of the field, with examples for each. Right now, other European projects (e.g., PHYSICS[57], CODECO[58], ICOS[59]) also involve the idea of virtualization and clustering for cloud computing, but those are work in progress.

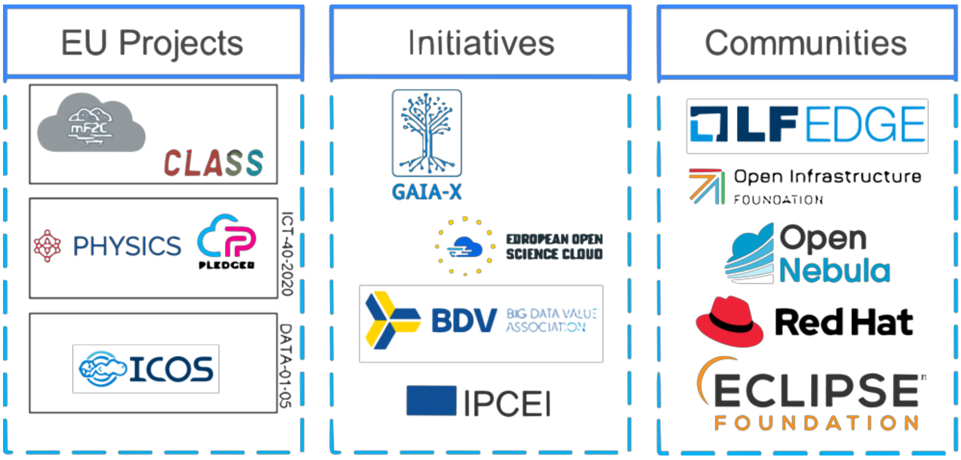


Figure 31: State-of-the-art: EU Projects, Initiatives and Communities

Between the enabling technologies, we have also identified the next ones: Open Cluster Management[60] (OCM), OpenShift[61], Containerd[62], Apache Airflow[63], Liqo[64] and Helm[65] and for each of this enabler technologies we pinpointed different functional aspects that match with some aspects in NEMO ecosystem.

Open Cluster Management (OCM)

Multi-Cluster Management: OCM focuses on managing multiple Kubernetes clusters, which match seamless with NEMO's objective of managing multicloud clusters and ensuring continuous operation across various technological environments.

Community-Driven Approach: The community-driven nature of OCM steers NEMO to put in place a collaborative environment, especially considering its open-source orientation.

Adaptation of Multi-Cluster Environments: Implementing OCM's strategies for managing and configuring target clusters is a key ingredient in keeping efficient workload distribution and management in NEMO's multicloud clusters.

Enhanced User Experience: Leveraging OCM's user-friendly approach to sped up the user involvement as such as administrators and developers, ensuring ease of use and management.

RedHat OpenShift

Lifecycle Management: OpenShift automates installation, upgrades, and lifecycle management, which can be utilized by NEMO to assess innovation throughout and post-project.

Security: With its enterprise-grade security, OpenShift can guide NEMO in ensuring cybersecurity and trust through mechanisms like Mutual TLS and Digital Identity Attestation.

[57] physics-faas.eu
[58] he-codeco.eu
[59] icos-project.eu
[60] Open Cluster Management (open-cluster-management.io)
[61] Red Hat OpenShift enterprise Kubernetes container platform
[62] containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability
[63] Documentation | Apache Airflow
[64] What is Liqo? — Liqo
[65] Helm | Docs

Automated Lifecycle Management: Integrating automated management of applications and services is a best practice for NEMO to reduce manual intervention.

Security Protocols: Adopting and be inspired by OpenShift's security protocols will ensure NEMO's operations are secure and trusted.

Containerd

Container Runtime: Containerd's functionalities in executing containers and managing images can support the NEMO's on-device Cybersecure Federated Deep Reinforcement Learning.

Embeddability and Maintenance: Containerd is embeddable and easy to maintain so it can push NEMO's development of a flexible and adaptable meta-Operating System.

Efficient Container Management: Utilizing Containerd's minimal and efficient container runtime functionalities to improve NEMO's container management.

Maintenance Strategies: Adopting Containerd's maintenance strategies to ensure NEMO's meta-Operating System remains sustainable and easy to manage.

Apache Airflow

Workflow Management: Apache Airflow's capabilities in developing, scheduling, and monitoring workflows can inspire NEMO's multi-technology Secure Execution Environment.

Extensibility: Its high extensibility and parameterization using Python can be beneficial for NEMO's Plugin and Apps Lifecycle Management.

Dynamic Workflow Generation: Implementing Apache Airflow's dynamic workflow is crucial to keep NEMO's Service Level Objectives

Parameterization Strategies: Utilizing Apache Airflow's parameterization strategies assure NEMO's tools and systems are adaptable and can be tailored to various use-cases and environments.

Liqo

Dynamic Multi-Cluster Management: NEMO's objective to create a multi-technology meta-OS that interfaces with various systems can be aligned with Liqo's ability to dynamically manage multicluster topologies.

Liqo workload: Implement Liqo's workload offloading and resource-sharing mechanisms to optimize resource utilization across the continuum, ensuring efficient operation even in resource-constrained environments.

Helm

Simplified Deployment and Management: Be inspired by Helm charts, NEMO can define, install, and upgrade complex Kubernetes applications, aligning with its objective to interface and leverage existing systems and technologies.

Enhanced Scalability and Adaptability: Helm charts approach will allow NEMO to define, install, and upgrade applications, ensuring that the system can adapt to new technologies and concepts introduced during and after the project lifecycle.

5.1.2 Relationship with NEMO

The NEMO meta-Orchestrator plays a pivotal role in realizing the grand vision of the NEMO platform. Positioned as a fundamental component, it serves as the meta-control plane, strategically placed atop existing container orchestration clusters such as Kubernetes (K8s). In this role, the meta-Orchestrator acts as the orchestrator of orchestrators, harmonizing the diverse resources and computing elements spread across the IoT to Edge to Cloud Continuum.

At its core, the meta-Orchestrator embodies the core principles of advanced intelligence, adaptability, and interoperability. These principles align seamlessly with NEMO's overarching objective: to provide efficient, user-centric, and secure computing solutions across the entire continuum. By doing so, the meta-Orchestrator serves as the linchpin that ensures the NEMO platform functions cohesively and effectively in the dynamic and multifaceted digital landscape.

In the context of NEMO, the meta-Orchestrator acts as the intelligent conductor of the symphony of services and resources, orchestrating them in a manner that optimizes performance, minimizes disruptions, and respects user-defined requirements. Its adaptability ensures that NEMO remains responsive to changing conditions and demands, facilitating scalability and reliability across the continuum.

Moreover, the meta-Orchestrator fosters interoperability, allowing seamless integration with a wide range of components, from micro-schedulers to local orchestrators and external tools and frameworks. This interoperability is crucial for NEMO to operate harmoniously, even in complex and heterogeneous computing ecosystems.

5.2 Background

In this section, we will explore the technical foundations that form the backbone of the NEMO meta-Orchestrator. Gaining insights into the crucial technological concepts and enablers is fundamental for grasping the significance and capabilities of this pioneering component.

Containerization

Containerization is the process of packaging the code of an application along with all its files and libraries. It facilitates the deployment of the application on any infrastructure by just uploading the packaged source code in the target machine and executing it.

Containerized applications are then deployed and executed as containers by a container runtime software. There are several container runtimes such as Docker[66], LXD[67], Podman[68], Kubernetes, etc. Some of these runtimes are low-level (e.g., containerd, LXD) while others operate on a higher level (e.g., K8s) providing more advanced operations such as execution of workflows, support for multiple nodes etc. Overall, each container runtime has its own advantages and disadvantages which are out of the scope of this deliverable. NEMO uses K8s for its use cases since it is the most widely used.

Containers have been used as an alternative and more lightweight approach to virtual machines (VMs) for isolating workloads running on a single bare-metal host. In contrast to VMs containers do not virtualize the whole hardware but use isolation mechanisms provided by the operating system, such as cgroups and seccomp, to allow the applications to run safely on the physical hardware. The basic advantage of containers is fast spawn and execution times since they do not need to boot a complete virtualized operating system and their main disadvantage is that they are less secure since they do not typically exploit the virtualization extensions that the hardware provides which adds an additional security layer. Figure 32 showcases the architectural difference between containers and VMs.

Microservices Architecture

Microservices Architecture is an architectural style which splits an application in a collection of small components, named microservices. Each microservice is designed, implemented, and can be deployed independently than others, the only dependence may be that a microservice requires its input to be the

[66] https://docs.docker.com/
[67] https://ubuntu.com/lxd
[68] https://podman.io/docs

output of another microservice. The following figure shows the difference of a monolithic application compared to an application that follows the microservice architecture.

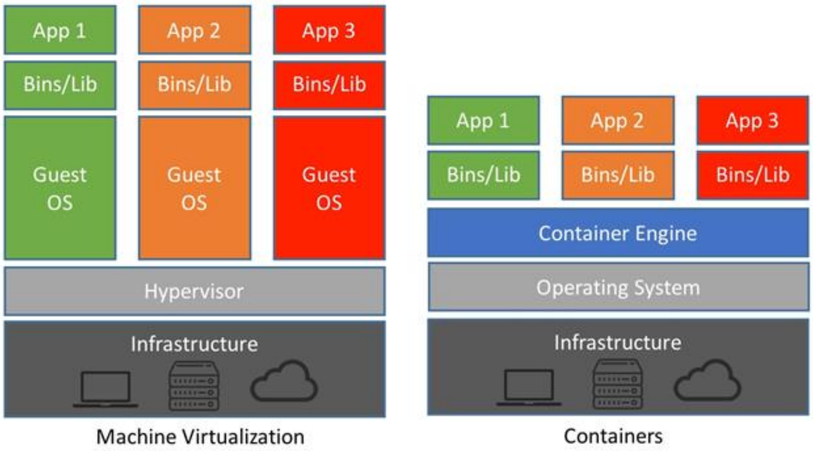


Figure 32: High level view of the architecture of VMs and containers (taken from [29])

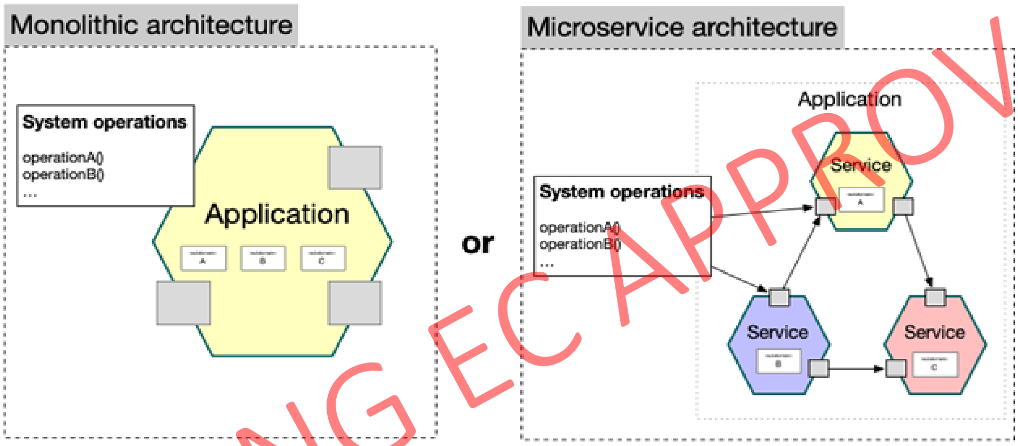


Figure 33: Difference between a monolithic application and one based on microservices (taken from [30])

Microservices architecture is of great importance for applications in the IoT to Edge to Cloud continuum where various components of the application may need to execute on different layers of the continuum. For this it is necessary to split the application in multiple independent microservices each one responsible for a single and simple task. Such applications provide greater flexibility for placement decisions to the administrators of the continuum by allowing them to decide where to execute each microservice independently of the rest ones.

Distributed Computing

In Distributed Computing multiple machines are interconnected and work together to solve a common problem. When it comes to the IoT to Edge to Cloud continuum the whole infrastructure is by-design distributed since there is a vast amount of IoT devices and Edge and Cloud servers that need to be coordinated. Some of the major challenges in this scenario are security, consistency of data across the whole system, network latency between different nodes, resource allocation and particularly migration between Edge and Cloud nodes. Figure 34 depicts the high-level view of the different components of the IoT to Edge to Cloud continuum.

Orchestration

Cloud orchestration refers to the process of automating and managing the deployment, configuration, coordination, and management of various cloud resources and services. It involves the efficient coordination and integration of multiple cloud-based systems, applications, and infrastructure components to deliver a cohesive and optimized cloud computing environment.

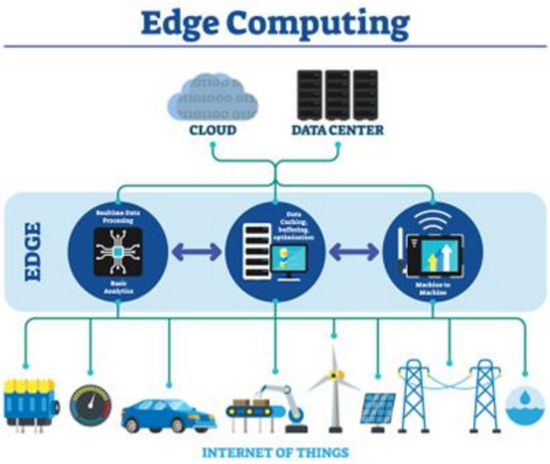


Figure 34: High-level view of IoT-to-Edge-to-Cloud Continuum (taken from [31])

Cloud orchestration enables organizations to streamline and automate complex workflows and tasks involved in provisioning, scaling, monitoring, and managing cloud resources. It provides a centralized control and management layer that abstracts the underlying complexities of the cloud infrastructure, allowing users to define and manage their resources using high-level policies and templates.
Cloud orchestration provides several benefits, including increased agility, scalability, and efficiency in managing cloud environments. It simplifies the management of complex infrastructure, enhances resource utilization, and accelerates the deployment of applications and services. By automating repetitive tasks and workflows, organizations can focus more on innovation and delivering value to their customers.

Cloud-Native Principles

Cloud Native Principles are essential guidelines for developing and managing containerized applications in cloud-native environments. These principles address the unique challenges and opportunities presented by cloud computing, enabling organizations to harness the full potential of containerization and cloud technologies. They offer a framework for creating highly performant, reliable, and scalable applications in this dynamic landscape.

- The **Single Concern Principle** emphasizes that each container should have a singular focus, simplifying management and reducing unexpected issues. Containers often align naturally with this principle by managing a single process, which corresponds to a single concern.
- The **High Observability Principle** stresses that containers should provide APIs for runtime observation, including health checks, logging, tracing, and metrics. Integrating with tools like OpenTracing[69] and Prometheus[70] enhances automation and system resilience.
- **Lifecycle Conformance** means containers receive platform events to manage their lifecycle. Detecting termination signals (SIGTERM) is crucial for clean shutdowns, while other events like PostStart and PreStop may be significant.

[69] https://opentracing.io/ https://opentracing.io/
[70] Prometheus - Monitoring system & time series database

- The **Image Immutability Principle** ensures that containerized applications remain immutable between different environments, enabling practices like automatic rollback and roll forward during updates.
- The **Process Disposability Principle** highlights that containers should be ephemeral, ready for replacement at any time. Smaller containers improve system reliability and startup times.
- The **Self-Containment Principle** dictates that containers should contain everything they need at build time, except for environment-specific configurations.
- Lastly, the **Runtime Confinement Principle** focuses on declaring resource requirements, such as CPU, memory, networking, and disk utilization, to prevent premature termination or migration during resource constraints.

Edge Computing

Multi-Access Edge Computing (MEC) is a technology framework that brings cloud computing capabilities and services closer to the edge of the network, typically at or near the cellular base stations or access points. It is designed to reduce latency and improve the efficiency of data processing for applications and services in mobile and wireless networks. Here is a brief description of MEC:

- **Edge Computing**: MEC leverages the concept of edge computing, where data processing, storage, and application execution occur closer to the data source or endpoint devices, rather than relying solely on centralized cloud data centers. This minimizes the round-trip data travel time, reducing latency and improving the overall user experience.
- **Proximity to End-Users**: MEC deploys computing resources at the "edge" of the network, near where end-users and devices are connected. This proximity allows for faster response times and better support for real-time and interactive applications, such as augmented reality, virtual reality, and autonomous vehicles.
- **Mobile and Wireless Networks**: MEC is particularly relevant in mobile and wireless networks, like 4G and 5G. By integrating computing resources at the base stations or access points, MEC enables network operators to provide low-latency services and optimize network traffic for various applications.
- **Use Cases**: MEC can be used in a wide range of applications, including IoT , smart cities, industrial automation, gaming, video streaming, and healthcare. For example, it can enhance augmented reality applications by reducing latency, making them more responsive and immersive.
- **Network Slicing**: MEC can work in conjunction with network slicing, a 5G technology that allows network operators to create virtualized, customized network segments for specific use cases. Network slicing, when combined with MEC, enables efficient resource allocation and tailored network services for diverse applications.
- **Developer-Friendly**: MEC provides APIs and developer tools that allow application developers to leverage edge resources easily. This encourages the creation of innovative, low-latency applications that can take full advantage of the localized computing infrastructure.

Multi-Access Edge Computing is a technology framework that extends cloud computing capabilities to the edge of mobile and wireless networks, enabling lower latency, improved performance, and enhanced support for a wide range of real-time and interactive applications. It plays a crucial role in the evolution of 5G networks and the proliferation of edge computing solutions.

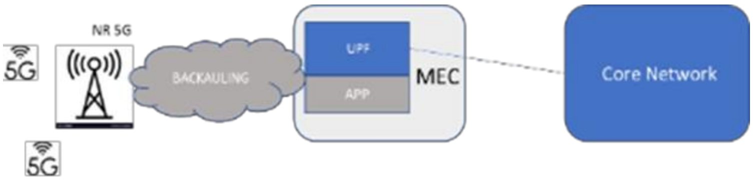


Figure 35: Multi-Access Edge Computing architecture.

5.3 Architecture & Approach

This chapter delves into the architecture and approach adopted by the NEMO meta-Orchestrator, providing insights into the high-level design, the role of each component, and the relationships that define the platform's functionality.

5.3.1 NEMO meta-Architecture Framework: Viewpoints

The NEMO meta-architecture introduces a comprehensive **"meta-Architecture Framework" (MAF)** tailored to the meta-OS environment. The framework encompasses various elements, each catering to distinct concerns and objectives, further described on D1.2. The aim of this section is to extend the understanding of the **Viewpoints**, defined as a collection of principles guiding the creation of an Architecture View, intending to address a set of **Concerns**.

NEMO has outlined several critical **Viewpoints** for the meta-OS meta-architecture, including "Network", "User", "Logical", "Operational", "Functional", "Process", "Development", and "Physical". The following subsections cover the **Development** and **Process Viewpoints**.

5.3.1.1 Development Viewpoint

The Development Viewpoint serves to discern the implementation intricacies of the meta-Orchestrator, an integral component delivering meta-OS capabilities within the NEMO framework. Its primary objectives encompass the identification of detailed implementation procedures for the meta-Orchestrator components and the establishment of measurable targets to ensure efficient capabilities' verification.

Table 3 (from D1.2) is describe the needs to implement the Development Viewpoint.

Development Viewpoint	
Description	This viewpoint aims to • identify the implementation details for components delivering meta-OS capabilities • identify measurable targets for capabilities' verification
Concerns addressed	Functionality Security Interoperability Performance
Usage	Implementation Setting Capability Requirements
Representation	Class diagram (components) Structured text (metrics)

Table 3: The Development Viewpoint

As showcase in Figure 36, this Viewpoint is composed by five main subcomponents that consolidate the whole meta-Orchestrator.

- Orchestration Engine: The central brain of the system, it coordinates and manages complex computing resources, making intelligent decisions to optimize workflows. It ensures seamless integration and simplification of distributed computing, emphasizing efficiency and security.

- Analytics Engine: Monitors and analyses performance metrics, identifying bottlenecks and inefficiencies to maintain peak performance. It provides insightful reports and visualizations, facilitating data-driven decision-making and ensuring transparent and efficient operations.

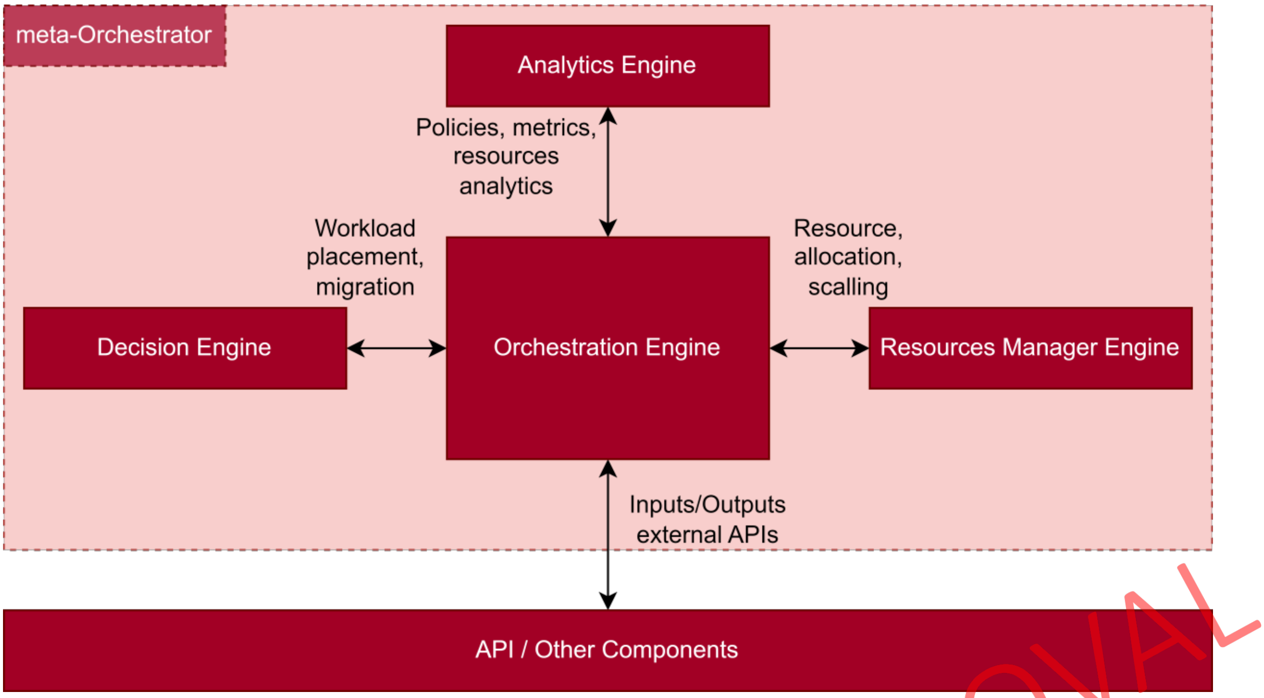

meta-Orchestrator
Analytics Engine
Policies, metrics, resources analytics
Workload placement, migration
Resource, allocation, scalling
Decision Engine
Orchestration Engine
Resources Manager Engine
Inputs/Outputs external APIs
API / Other Components

Figure 36: Development Viewpoint of the meta-Orchestrator

- Resource Manager: Manages the lifecycle of resources by overseeing provisioning, scaling, monitoring, and deprovisioning services. It maintains a comprehensive view of system capabilities, optimizing resource usage and allocation to meet dynamic demands effectively.

- Decision Engine: Acts as the central intelligence unit, enforcing policies, optimizing costs, and allocating workloads based on workload characteristics and performance metrics. It ensures effective orchestration while minimizing operational costs.

- Integration Component: Acts as a mediator for seamless communication between various modules, providing an abstracted view of the system's architecture. Its intent-based approach enables effective communication and interoperability between different components, promoting a secure and functional communication framework within the meta-Orchestrator.

5.3.1.2 Process Viewpoint

The Process Viewpoint revolves around identifying representative use cases that effectively demonstrate the collective capabilities of the meta-OS, with a specific focus on the meta-Orchestrator. It delves into the detailed analysis of the components responsible for delivering these use cases, emphasizing the intricate interactions and data flows between them.

Table 4 (from D1.2) is describe the needs to implement the Development Viewpoint.

Process Viewpoint	
Description	This viewpoint aims to • Identify representative use cases delivering the combined capabilities of the meta-OS • Identify the components delivering those use cases • Identify the interactions among these components for delivering the use cases

	• Identify data flows within these interactions
Concerns addressed	Functionality Security Interoperability
Usage	Highlight potential integration requirements Implement/enable interactions among meta-OS components
Representation	Sequence diagram

Table 4: The Process Viewpoint

The sequence diagram for the meta-Orchestrator depicts the orchestrated workflow and data flow within the NEMO ecosystem. It highlights the intricate interactions among various components, related with the described ones in the Development Viewpoint. The Figure 37 illustrates the approach of such flows within the meta-Orchestrator.

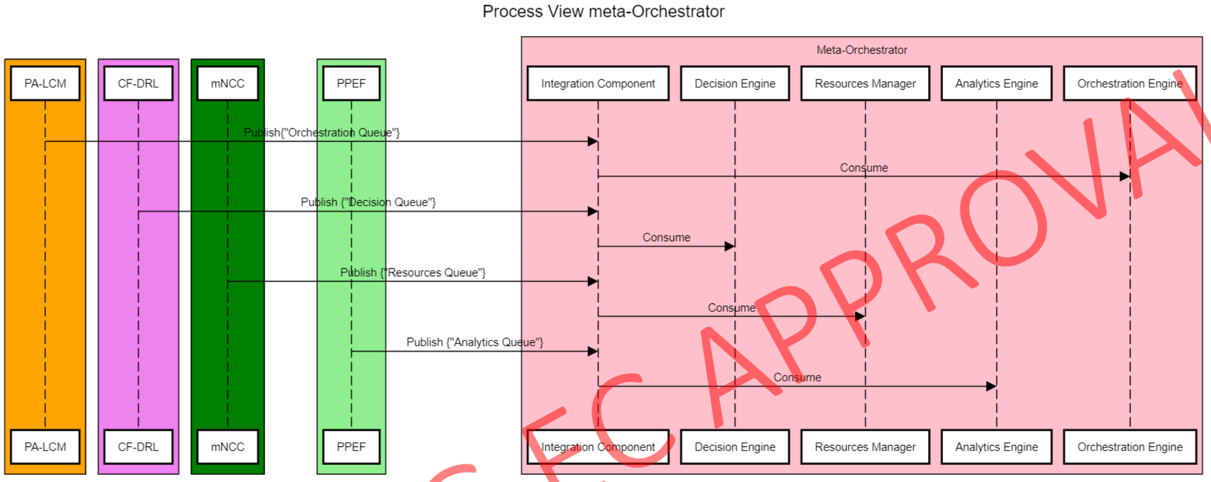


Figure 37: Process Viewpoint of the meta-Orchestrator

The sequence diagram depicts the operational framework of the meta-Orchestrator (MO) in collaboration with various essential components. These include the Cybersecure Federated Deep Reinforcement Learning (CFDRL), the Cybersecure Microservices Digital Twin (CMDT), the Intent-Based Migration Controller (IBMC), the meta-Network Cluster Controller (mNCC), and the Plugin & Apps Lifecycle Manager (PA-LCM) within the meta-OS environment.

The diagram elucidates the orchestrated workflow within the meta-Orchestrator, emphasizing the seamless communication and data flow among the crucial components. It highlights the vital role played by the Integration Component in facilitating efficient communication and coordination among the diverse modules. The Decision Engine is shown to make informed decisions based on cognitive reasoning and data-driven insights provided by the Analytics Engine. Additionally, the Resource Manager optimizes resource allocation and management, ensuring the efficient operation of the entire system.

Overall, the sequence diagram demonstrates the MO's ability to manage complex tasks within the meta-OS, emphasizing the secure and efficient orchestration of workflows and resources.

5.3.2 Approach of This Architecture

The NEMO meta-Orchestrator architecture represents a sophisticated and intelligent open-source software system with a primary objective of facilitating the decentralization and distribution of computing workflows across the IoT to Edge to Cloud Continuum. Serving as a central orchestrator, it assumes responsibility for the coordination and execution of intricate distributed systems, addressing the challenges stemming from their increasing complexity and diversity.

This meta-Orchestrator adopts a comprehensive approach to orchestration by delving into various facets of distributed computing workflows. It conducts an in-depth analysis of the structure and application programming interfaces (APIs) of different micro-schedulers and local orchestrators. This analytical prowess enables the meta-Orchestrator to seamlessly integrate and coordinate with diverse components within the distributed system architecture.

Intelligence is at the heart of the meta-Orchestrator's decision-making process. It considers crucial parameters like migration time, downtime, and overhead time when orchestrating computing workflows. This ensures that workflows are orchestrated in a manner that minimizes disruption and maximizes efficiency.

Beyond these fundamental parameters, the meta-Orchestrator also evaluates a wide array of functional and non-functional requirements. Functional parameters encompass network and resource availability through Intents, crucial for successful workflow execution. Non-functional requirements, on the other hand, encompass policies, energy efficiency, CO2 footprint, and financial operations (FinOps) considerations such as networking and hosting costs. By incorporating these non-functional requirements, the meta-Orchestrator enables workflow optimization based on multiple criteria, including environmental impact and cost-effectiveness.

5.4 Description of Components

The NEMO platform comprises several key components, each contributing to the efficient orchestration of computing workflows within the IoT to Edge to Cloud Continuum. These components include Orchestration Engine, Analytics Engine, Resource Manager, Decision Engine, and Integration Component.

5.4.1 Orchestration Engine

Description

The Orchestration Engine is the heart of the NEMO meta-Orchestrator, serving as its central brain and control center. This component plays a critical role in coordinating and managing the complex flow of computing resources, workloads, and services. It is responsible for smoothly orchestrating complex workflows. It takes on the huge task of simplifying the inherent complexities of distributed computing, enabling NEMO to work efficiently in this intricate ecosystem.

By acting as the central control hub within the NEMO meta-Orchestrator, the Orchestration Engine plays a crucial role in making smart decisions, optimizing resource usage, and bringing together different elements within the NEMO ecosystem. Its importance goes beyond just orchestration; it embodies the essence of NEMO's mission in navigating the complexities of the IoT to Edge to Cloud Continuum.

Example of Technology Enablers

- Open Cluster Management (OCM)[71] is a robust technology enabler that serves as a comprehensive solution for managing clusters of containers and resources across the IoT to Edge to Cloud Continuum. It provides advanced features like cluster lifecycle management, governance, policy enforcement, and multicluster visibility. OCM is chosen for its ability to handle the complexity of orchestration across distributed environments seamlessly. It allows the

[71] https://open-cluster-management.io/

Orchestration Engine to efficiently manage and allocate resources while maintaining a high level of control and visibility across clusters.

- Rafay[72] is a cloud-native platform that specializes in simplifying the deployment, operations, and lifecycle management of containerized applications across various environments. It offers features such as automated deployment pipelines, scaling, and robust security mechanisms. Rafay is selected as a technology candidate for its ability to streamline application deployment and management, reducing operational overhead for the Orchestration Engine. This ensures that NEMO's workflows are orchestrated efficiently and with minimal manual intervention.
- Rancher[73] is a widely adopted open-source platform for managing Kubernetes clusters. It provides a user-friendly interface for cluster management, monitoring, and security. The Orchestration Engine leverages Rancher to interact with Kubernetes clusters, simplifying the management of containerized workloads. Rancher's intuitive dashboard and management capabilities enhance the overall user experience, making it easier to deploy and manage applications within NEMO.
- Karmada[74] is an emerging technology enabler designed to simplify the orchestration of workloads and applications across multiple Kubernetes clusters. It offers capabilities for workload placement, scaling, and resource optimization. Karmada is chosen to enhance the Orchestration Engine's ability to optimize resource allocation and workload distribution across clusters in the IoT to Edge to Cloud Continuum. Its innovative approach aligns with NEMO's goal of efficient and adaptable orchestration.

Technology Chosen

The choice of Open Cluster Management (OCM) as the primary technology enabler for the Orchestration Engine within the NEMO meta-Orchestrator is driven by several compelling factors that align perfectly with NEMO's overarching goals and requirements.

- Robust Cluster Management: OCM is renowned for its robust cluster management capabilities. It provides a unified platform for managing clusters of containers and resources across diverse environments, including the IoT, Edge, and Cloud Continuum. This aligns with NEMO's mission of orchestrating resources seamlessly across this complex landscape. OCM's cluster management features are essential for coordinating the deployment and scaling of containerized workloads, ensuring optimal resource utilization and availability.
- Multi cluster Visibility: OCM offers comprehensive multicluster visibility, allowing the Orchestration Engine to gain insights into the status and performance of clusters distributed throughout the continuum. This visibility is invaluable for making informed decisions about workload placement and resource allocation. It enhances NEMO's ability to optimize the orchestration process, ensuring that computing workflows are executed efficiently while adhering to specified policies.
- Governance and Policy Enforcement: OCM provides robust governance and policy enforcement mechanisms, which are crucial for maintaining control and compliance within the NEMO ecosystem. The Orchestration Engine can define policies related to resource allocation, security,

[72] https://rafay.co/
[73] https://www.rancher.com/products/rancher
[74] https://karmada.io/

and scalability. OCM ensures that these policies are consistently enforced across clusters, contributing to the secure and efficient operation of NEMO.

- Lifecycle Management: OCM streamlines cluster lifecycle management, covering provisioning, scaling, monitoring, and deprovisioning. This aligns perfectly with NEMO's goal of dynamically adjusting resource allocation based on changing conditions and requirements. The Orchestration Engine leverages OCM to ensure that clusters are provisioned and scaled as needed to support the orchestrated workflows.
- Multi cloud and Multi cluster Support: OCM's multicloud and multicluster support is a vital asset for NEMO. It enables the Orchestration Engine to manage resources and workloads seamlessly across different cloud providers and clusters. This flexibility is essential for accommodating the diverse computing environments within the continuum, allowing NEMO to operate effectively in various scenarios.
- Open Source and Extensibility: OCM is open source, offering a high level of flexibility and extensibility. This aligns with NEMO's commitment to openness and adaptability. The Orchestration Engine can leverage OCM's extensibility to customize and extend its capabilities as needed to meet evolving requirements.

Model Interfaces

The Orchestration Engine interfaces with various internal components, including the Resource Manager, Decision Engine, Analytics Engine, and Integration Component, to ensure efficient orchestration.

Licensing

The Orchestration Engine is released under an open-source license, ensuring accessibility and flexibility for users.

5.4.2 Analytics Engine

The Analytics Engine processes the performance metrics of orchestrated workflows and computing resources, supporting the identification of potential bottlenecks and inefficiencies to ensure optimal operation. Through continuous oversight, it ensures that all orchestrated processes maintain peak performance and swiftly identifies any deviations or anomalies communicating them to the component that take decisions Furthermore, it generates insightful reports and visualizations, offering a clear and concise view of the status, performance, and utilization of orchestrated resources and workflows. This not only ensures transparency in operations but also facilitates data-driven decision-making by providing stakeholders and system components with crucial analytical outputs.

- InfluxDB[75]: An open-source time series databases that for storage and retrieval time series
- Graphite[76]: A complex monitoring tool for sophisticated hardware and complex cloud solutions monitoring.
- Zabbix[77]: An open-source monitoring software tool able to extract data from networks and servers from cloud-based services.

[75] https://www.influxdata.com/time-series-platform/

[76] https://graphiteapp.org/

[77] https://www.zabbix.com/

- Kibana[78]:A tool for visualizing and analyzing data, particularly useful for inspecting logs, tracking time-series data, and gaining operational insights.
- Tableau[79]: A data visualization tool that is used for converting raw data in visual and images of easy comprension.
- RabbitMQ[80]: An open-source message broker that is fine tuned for different messaging protocols.
- ActiveMQ[81]: An open-source message broker written in Java that is also a message oriented middleware.
- MQTT[82]: A messaging protocol tailored for smaller devices and sensors. It is planned to cope with environments with high-latency or unstable network connections.
- Prometheus[83]: An open-source monitoring and alerting toolkit that collect data from time series database.
- Kafka[84]: A distributed data streaming platform that allows you to publish, subscribe, store and process streams of data.
- Grafana: An open-source platform for monitoring and observability, offering functionalities to visualize, set alerts and interpret related metrics.

Technology Chosen

- Prometheus as performance monitor and alerting tool kit, can be employed to continuously monitor the performance of the orchestrated workflows and computing resources as well as It can collect metrics from configured targets at given intervals, evaluate rule expressions, and can trigger alerts if certain conditions are observed, ensuring that the Analytics Engine is always informed about the system's health and performance.
- Grafana for reporting and visualization. It is a multi-platform open-source analytics and interactive visualization web application that create comprehensive reports and visualizations based on the data processed and analysed by the Analytics Engine inner components. It can visualize the data collected and processed by Prometheus, providing a clear, concise, and interactive for status of the workflows.
- Kafka Bus as enabler of the communication among the components

Model Interfaces

The Analytics Engine interfaces include relation with the Resource Manager, Decision Engine, to ensure efficient orchestration.

Licensing

The Analytics Engine is released under an open-source license, ensuring accessibility and flexibility for users.

5.4.3 Resource Manager

Description

[78] https://www.elastic.co/kibana
[79] https://www.tableau.com/learn/articles/data-visualization
[80] https://www.rabbitmq.com/protocols.html
[81] https://activemq.apache.org/
[82] https://www.techtarget.com/iotagenda/definition/MQTT-MQ-Telemetry-Transport
[83] https://prometheus.io/docs/introduction/overview
[84] https://kafka.apache.org/

The resource manager is a component that will be in charge of managing the lifecycle of resources. This will include the steps of provisioning, scaling, monitoring, and deprovisioning the services. The main goal is to have a complete and up-to-date view of the different capabilities the system has and how they are being used, selecting the optimal way to use them and, if it is needed, also having a repository to know if they are still needed and if not, being able to detect it and reduce or eliminate the resource provisioning for that request.

This task will have two main blocks: the update of the resource view, that will be feed with data received from the mNCC module; and the decision making, that will detect if the resources are being correctly assigned and request for modification if not. This last block will be realized in collaboration with the Orchestration engine, that will be the module on charge to take these decisions based on the information provided from the Resource Manager module.

Examples of technology enablers

To accomplish its tasks, the Resource Manager leverages various cutting-edge technologies, including but not limited to:

- Infrastructure as Code (IaC): Implementing IaC enables the Resource Manager to automate the provisioning, configuration, and management of infrastructure resources, allowing for rapid and consistent deployment of resources across the network.
- Containerization Technology: Leveraging containerization technology facilitates efficient resource allocation and management, enabling the Resource Manager to handle resources more effectively within containerized environments, thus optimizing resource usage.
- Auto-Scaling Mechanisms: Implementing auto-scaling mechanisms allows the Resource Manager to dynamically adjust resource allocation based on current demand, ensuring that the system can efficiently scale resources up or down in response to workload fluctuations.

Technology Chosen

Once we already have the complete design, we will study the best technologies to realize the deployment, and therefore select the ones chosen.

Model Interfaces

Resource Manager module interfaces with various internal components, including the Orchestration Engine and the Integration Component, to provide them network resources information.

Licensing

Resource Manager module is released under an open-source license, ensuring accessibility and flexibility for users.

5.4.4 Decision Engine

Description

The Decision Engine acts as the central intelligence unit of the system, closely collaborating with the Orchestration Engine to enforce policies, optimize costs, and efficiently allocate workloads. It processes inputs from the Orchestration Engine, considering factors such as workload characteristics, resource availability, and performance metrics, to make informed decisions. By leveraging sophisticated algorithms, the Decision Engine ensures that computing workflows are orchestrated effectively while minimizing operational costs.

Examples of Technology Enablers

To fulfil its functions, the Decision Engine leverages various advanced technologies, including but not limited to:

- Machine Learning Algorithms: These algorithms enable the Decision Engine to analyze historical data, identify patterns, and make informed decisions based on predictive analysis. By learning from past data, the Decision Engine can anticipate future trends and optimize resource allocation accordingly.
- Rule-Based Systems: The implementation of rule-based systems empowers the Decision Engine to enforce specific policies and guidelines. By setting predefined rules and regulations, the system ensures that the orchestration processes adhere to the established protocols, promoting consistency and adherence to best practices.
- Cloud Computing Infrastructure: Leveraging cloud computing infrastructure provides the Decision Engine with the necessary resources and scalability to handle varying workloads. By utilizing cloud services, the Decision Engine can dynamically allocate resources based on demand, ensuring efficient and cost-effective management of computing workflows.
- Application Programming Interfaces (APIs): Integration of APIs allows seamless communication and data exchange between the Decision Engine and other components. By facilitating efficient information transfer and instructions, APIs enable the Decision Engine to coordinate with external systems and applications, streamlining the overall orchestration process.

Technology Chosen

Once we already have the complete design, we will study the best technologies to realize the deployment, and therefore select the ones chosen.

Model Interfaces

Decision Engine module interfaces with the other four main internal components, to provide them intelligent capabilities.

Licensing

Decision Engine module is released under an open-source license, ensuring accessibility and flexibility for users.

5.4.5 Integration Component

Description

The Integration Component is a sub-module that will be in charge of acting as a broker in the communications between the different modules and the meta-Orchestration module. The main goal for this module is to realize an abstraction view to avoid the meta-Orchestrator's inner components to have a complete view of NEMO's architecture and allows to realize these communications without needing the rest of the modules to have a knowledge of the meta-Orchestrator's inner architecture.

The Integration Component will be based as an Intent-Based component, that realizes a translation and inter-modules communication tasks. This will work in both directions, abstracting inbound and outbound communications.

Examples of technology enablers

To design and deploy this module, it will be needed some integration technologies related with the communications between modules, the inter-cluster communications, and the data modelling. Some examples of potential technologies to use are:

- Kubernetes CRDs. The Custom Resources Definitions are Kubernetes' extensions that allow users to define and create their own custom resources with specific data structures and behaviours that are not available in the default Kubernetes resources (e.g., Pods, Deployments, Services, etc.). This is fundamental to the extensibility and customisation of Kubernetes to specific use cases and applications.
- Yet Another Next Generation (YANG). YANG data models are a standard notation used to describe data and operations in network devices and network management systems. They were developed by the NETMOD (Network Configuration Protocol) working group of the IETF (Internet Engineering Task Force) to standardise the representation of data and operations in network management environments. These data models are designed to be readable by both humans and machines and are used to describe the structure and semantics of data handled in network systems. A YANG data model defines how data should be organised and structured, what types of data can be stored and how it can be manipulated.
- Intent-Based Networking (IBN). IBN is an approach to computer networking that seeks to simplify and automate network management by focusing on the business or high-level intentions of users rather than the technical configuration, thus abstracting from implementation details. Instead of configuring each network device individually, administrators specify their goals and policies to a management system that translates those intentions into appropriate network configurations. Intent-Based Networking seeks to simplify network management, improve operational efficiency, reduce human error, and enable networks to be more agile and adaptable to meet high-level defined requests.

Technology Chosen

Once we already have the complete design, we will study the best technologies to realize the deployment, and therefore select the ones chosen.

Model Interfaces

Integration Component module interfaces with the other four main internal components, to provide them an architectural abstraction of the outside. Also, it will communicate with other external modules from NEMO's system as the CMDT, PPEF, MOCA, CFDRL, mNCC and IBMC.

Licensing

Integration Component module is released under an open-source license, ensuring accessibility and flexibility for users.

5.5 Interaction with other NEMO components

In the following, the interaction of the meta-Orchestrator with relevant other components is described.

5.5.1 Interacting with the Intent-Based Migration Controller (IBMC)

The meta-Orchestrator collaborates with the Intent-Based Migration Controller (IBMC) in both input and output capacities. As an input, the IBMC communicates migration intents, specifying constraints and requirements for transferring computing workflows across different domains. The meta-Orchestrator uses this information to ensure smooth and continuous workflow migration during the orchestration process. As an output, the meta-Orchestrator provides feedback and updates to the IBMC, enabling the controller to monitor the status and progress of the workflow migration, and make necessary adjustments as needed.

5.5.2 Interacting with the meta-Network Cluster Controller (mNCC)

The Federated meta–Network Cluster Controller (mNCC) is an AI-driven controller designed to manage and optimize the formation and operation of network clusters in a dynamic and self-healing manner, focusing on fog IoT deployments and utilizing advanced technologies like NOMA, network zones, SDN, and AI for efficient management and guaranteed service levels. mNCC module and meta-Orchestrator will collaborate to create optimal, self-healing hosting clusters, across different administrative boundaries.

We propose an Intent-Based communication between these two modules, in order to provide a abstraction layer to facilitate the integration and the possible adaptation of the internal interface to potential modifications in the expected functionality. The interface should have two connections in the mNCC: One output interface that comes from the Network Exposure sub-module (mNCC) into the Integration component (and therefore to the Resource Manager), and the other one that comes from Orchestration Engine into the Intent-Based System (mNCC) that indicates the infrastructure modifications that must be applied. Even both interfaces are only to share information in just one direction, both should be bidirectional to facilitate communications and state-feedback.

5.5.3 Interacting with the Cybersecure Microservices' Digital Twin (CMDT)

The collaboration with the Cybersecure Microservices' Digital Twin (CMDT) plays a vital role in ensuring the security and integrity of microservices within the distributed computing environment. By receiving input from the CMDT regarding potential security risks and vulnerabilities associated with various microservices, the meta-Orchestrator can proactively incorporate security measures into the orchestration decisions. Furthermore, the meta-Orchestrator's output interactions with the CMDT involve the implementation of security-related configurations and enforcement measures, ensuring the continuous and robust operation of microservices throughout the orchestration process.

5.5.4 Interacting with the Cybersecure Federated Deep Reinforcement Learning (CF-DRL)

The meta-Orchestrator's engagements with the Cybersecure Federated Deep Reinforcement Learning (CF-DRL) component significantly enhance the security and risk management measures within the orchestration environment. Leveraging the CF-DRL component's reinforcement learning models and insights, the meta-Orchestrator efficiently incorporates advanced security strategies, effectively mitigating risks and enforcing robust policy enforcement measures. Additionally, the meta-Orchestrator provides comprehensive feedback to the CF-DRL component, ensuring a continual refinement of security models and strategies, thereby fostering an environment of continuous improvement and heightened security within the orchestration framework.

5.5.5 Interacting with Monetization and Consensus-based Accountability (MOCA)

The collaboration with Monetization and Consensus-based Accountability (MOCA) introduces essential functionalities crucial for the financial and accountability aspects of the NEMO system. As an input, MOCA provides valuable insights into monetization strategies and consensus-based accountability mechanisms, allowing the meta-Orchestrator to incorporate these factors into the orchestration decisions. This integration ensures efficient resource allocation and optimization, aligned with the financial goals and consensus-based principles of the NEMO system. Moreover, as an output, the meta-Orchestrator provides instructions and updates to MOCA, enabling continuous tracking and monitoring of financial transactions and accountability measures, thereby fostering transparency and accountability within the NEMO ecosystem.

5.5.6 Interacting with Privacy and Policy Enforcement Framework (PPEF)

The Privacy and Policy Enforcement Framework (PPEF) acts as a critical mediator between the comprehensive analytics engine used by the meta-Orchestrator and the NEMO infrastructure clusters. Playing both an input and output role, the PPEF collects, processes, and presents essential metrics and analytics derived from the entire infrastructure. These metrics encompass areas such as privacy, data protection, ethics, security, and societal impacts, providing crucial insights for the meta-Orchestrator's decision-making process. Furthermore, the PPEF communicates processed insights and recommendations back to the meta-Orchestrator, enabling the enforcement of robust privacy measures and stringent policy compliance, fostering a secure and ethically sound orchestration environment.

5.6 Conclusion, Roadmap & Outlook

The NEMO meta-Orchestrator serves as a pivotal solution for the intricate challenges posed by the IoT to Edge to Cloud Continuum. Its fundamental role lies in simplifying the complexities of distributed computing, ensuring optimal resource utilization, and enabling seamless integration across diverse components and systems. With its focus on adaptability, scalability, and efficiency, the meta-Orchestrator stands as a testament to NEMO's commitment to fostering decentralized and efficient computing workflows. Through its intelligent decision-making processes and emphasis on interoperability, the meta-Orchestrator not only streamlines workflow distribution and management but also ensures NEMO's ability to thrive within the dynamic technological landscape.

Moving forward, the roadmap for the NEMO meta-Orchestrator entails a continuous evolution aligned with the rapidly changing demands of the technology ecosystem. Firstly, it involves enhancing its intelligence capabilities to further optimize workflow efficiency and resource management. Secondly, the focus will be on expanding its adaptability to accommodate the growing scale and diversity of computing environments. Additionally, efforts will be directed towards reinforcing its interoperability to facilitate seamless integration with emerging technologies and diverse systems. Furthermore, the roadmap includes fostering collaborations to bolster the meta-Orchestrator's functionality and applicability in addressing the evolving challenges across the IoT to Edge to Cloud Continuum. By aligning these strategic objectives, the NEMO meta-Orchestrator is poised to remain at the forefront of orchestrating complex computing workflows, ensuring the continued success of the NEMO platform.

The future of the NEMO platform looks promising, with a strong focus on advancing its meta-Orchestrator components to facilitate seamless and efficient computing workflows within the IoT to Edge to Cloud Continuum. The continual evolution of the Orchestration Engine, Analytics Engine, Resource Manager, Decision Engine, and Integration Component is expected to drive significant advancements in resource management, decision-making, and data integration. By harnessing the power of cutting-edge technologies and fostering collaborations with industry leaders, NEMO aims to establish itself as a pioneer in orchestrating complex computing workflows and enabling efficient resource utilization. The platform's outlook rests on its ability to adapt to evolving technological landscapes, ensuring robustness, scalability, and interoperability within the dynamic IoT to Edge to Cloud Continuum.

6 Proof of Concept: NEMO Kernel Space

This phase primarily focuses on thoroughly validating the NEMO's functionality, performance, security, and compliance. The chapter outlines the specific processes involved in this validation, including workload registration, updates, deployment, monitoring, migration, and secure communication protocols with the aim to reach a first integrated version of the components.

It emphasizes the significance of these processes in establishing a cohesive and efficient operational structure for the system. The chapter also addresses the objectives, expected outcomes, and limitations of the Integration PoC, providing valuable insights into the complexities of system integration and its role in ensuring a robust and reliable operational framework.

6.1 Overview

6.1.1 Workloads integration

The purpose of the 1st Integration Proof of Concept (PoC) is to comprehensively validate and assess the integrated system's functionality, performance, security, and compliance. It aims to evaluate the system's readiness for real-world deployment by engaging in various critical activities. Below is an outline of the PoC's purposes and a schema to illustrate its structure:

- **Workload Registration:** Register workloads through the API/PA-LCM tracking their entry into the system.
- **Workload Updated/Followed:** Verify that CMDT accurately updates and tracks changes in workload configurations. Monitor workload changes in CMDT is related to the registration of new services and metrics workloads related as specified in D2.1.
- **Workload Deployment:** Deploy workloads across the meta-Orchestrator, meta–Network Cluster Controller (mNCC), Intent-Based Migration Controller (IBMC), and Secure Execution Environment (SEE).
- **Workload Monitoring**: Assess the ability of PPEF to monitor workloads and enforce policies.
- **Workload Migration:** Test the feasibility of migrating workloads, including cloud-to-edge, cross-admin domain, and intent-based networking/computing scenarios, using meta-Orchestrator and mNCC.
- **Cognitive Workload**: Integrate CFDRL for enhanced security and reinforcement learning capabilities.
- **Secure Communication:** Implement secure communication protocols, access control, and authentication mechanisms, including Keycloak and RabbitMQ, to safeguard data exchange and access.

6.1.2 Objectives

The objectives of the 1st Integration Proof of Concept (PoC) are to validate the integrated system's functionality, assess its performance, and ensure its security and compliance:

- The first objective is to validate the integration of core system components, including the meta-Orchestrator, CMDT, mNCC, IBMC, SEE, PPEF, and CFDRL, ensuring they work cohesively to support the end-to-end workload lifecycle.
- Performance assessment is another key objective, focusing on evaluating the system's efficiency in handling workload deployment, monitoring, and migration tasks. This includes measuring resource utilization, response times, and overall system responsiveness.
- Security and compliance testing are essential objectives to verify the effectiveness of security mechanisms, access control, and policy enforcement within the integrated system. It aims to ensure data protection and adherence to regulatory and governance requirements.

- Operational efficiency is a critical objective, emphasizing the need for smooth workload management and migration processes. The goal is to minimize downtime and disruptions while maintaining optimal system operation.
- Policy adherence is a central objective, ensuring that the integrated system consistently follows defined policies, SLAs, and governance rules throughout workload deployment and migration activities.
- Scalability assessment is vital to understand the system's ability to adapt to increasing workloads and administrative domains effectively. It explores how the system can gracefully accommodate growth.

6.1.3 Expected Outcomes

The expected outcomes of the 1st Integration Proof of Concept (PoC) encompass functional and performance-related achievements:

- The PoC is expected to demonstrate functional integration, showcasing the seamless collaboration of all key components. This includes registration, deployment, monitoring, and migration capabilities, highlighting their reliability and consistency.
- Security and compliance expectations involve the establishment of robust security measures, access control mechanisms, and policy enforcement. This ensures secure communication and compliance with governance rules.
- Performance optimization is an expected outcome, with minimal downtime during workload migration, efficient resource allocation, and the creation of optimized network paths to enhance system responsiveness.
- Policy adherence and SLA fulfilment are expected to be realized, preventing SLA violations, and ensuring workload compliance with defined policies and performance metrics.
- The PoC aims to showcase scalability and adaptability, proving the system's ability to handle increasing workloads and administrative domains while maintaining operational efficiency.
- Integration of CFDRL is expected to enhance security and reinforce learning capabilities within the system, potentially improving threat mitigation and overall system intelligence.
- Intent-based networking and computing principles, such as micro slices and multipath routing, are anticipated to be effectively implemented to optimize workload management and network resource utilization.
- Data consistency and real-time monitoring in CMDT are expected outcomes, ensuring accurate tracking of workload updates and configurations for reliable data representation.
- Operational efficiency is a key goal, emphasizing controlled workload migration to minimize disruption to ongoing operations and maintain consistent system performance.

6.1.4 Limitations

The 1st Integration PoC exhibits inherent limitations, characterized by its inability to fully replicate the multifaceted intricacies of real-world production environments and its lack of comprehensive exploration of security threats, data privacy considerations, and regulatory compliance facets.

- The PoC may not replicate the full spectrum of complexities encountered in real-world production environments, potentially missing unforeseen variables, and external factors.
- It may not fully mirror the scale and diversity of large-scale production systems, limiting the assessment of how the system handles extensive workloads and administrative domains.
- While addressing security mechanisms, the PoC may not comprehensively explore real-time security threats and vulnerabilities, necessitating a deeper level of security testing for production readiness.
- The PoC does not delve deeply into data privacy and specific regulatory compliance, which may be essential considerations in real-world deployments.

- It assumes an idealized environment and may not account for resource limitations, such as constrained computing resources or network bandwidth, which can impact system performance in practice.

6.2 Workload deployment

The workload deployment from the meta-Orchestrator consists in two main parts:

- The automatic deployment and configuration of a multicluster based in OCM.
- The deployment of services and applications via the “multicloud-operators-subscription”

Both deployments use a broker-based communication defined by the integration component in which a yaml file containing the information of each deployment is send to a queue and consumed by either the OCM or the subscription consumer.

In the first step of the OCM deployment, a hub cluster is created and the clusteradm command-line tool is installed. This tool is used to install the registration-operator on the hub cluster, which is responsible for installing and upgrading a few core components for the OCM environment.

Once the hub cluster is set up, a consumer starts listening for petitions to join the system. Currently, these petitions consist in a yaml file where the number of managed clusters that will join the hub cluster is defined. In addition to the registration of the managed clusters, the multicloud-operators-subscription addon is deployed and ready to use. The current format of such file is as follows:

```yaml
numberofclusters: 4
```

The deployment of services and applications uses its own consumer, which listens for a petition to subscribe to a source repository channel that can be the following types: Git repository, Helm release registry or S3 object storage repository. The petition consists in a yaml file where three parameters are specified: the deployment name, the namespace of the deployment and the path to the repository.

In the given example, a deployment named as service is created, where the namespace is the default one and the service is hosted in a Git repository.

```yaml
name: service
namespace: default
pathname: https://service.github.io/service-1/
```

When the petition arrives, the subscription operator downloads directly from the storage location and deploys to targeted managed clusters without checking the hub cluster first. After the deployment is completed, the channel is monitored for new or updated resources and applies them at specified intervals.

Following the architecture provided by OCM’s documentation[85], a new shape for the Orchestration Engine is constructed. Figure 38 shows the architecture of the Orchestration Engine component using the Integration Component (based on RabbitMQ) to relate to the different microservices hosted on several sources (Git, S3, Minio, Helm, etc).

It is remarkable that the hub cluster might be based on either Vanilla K8s or kind (Kubernetes in Docker), whereas the managed clusters might be based on several lightweight Kubernetes-based flavours, such as K3S, microK8s or kind.

[85] https://open-cluster-management.io/concepts/architecture/

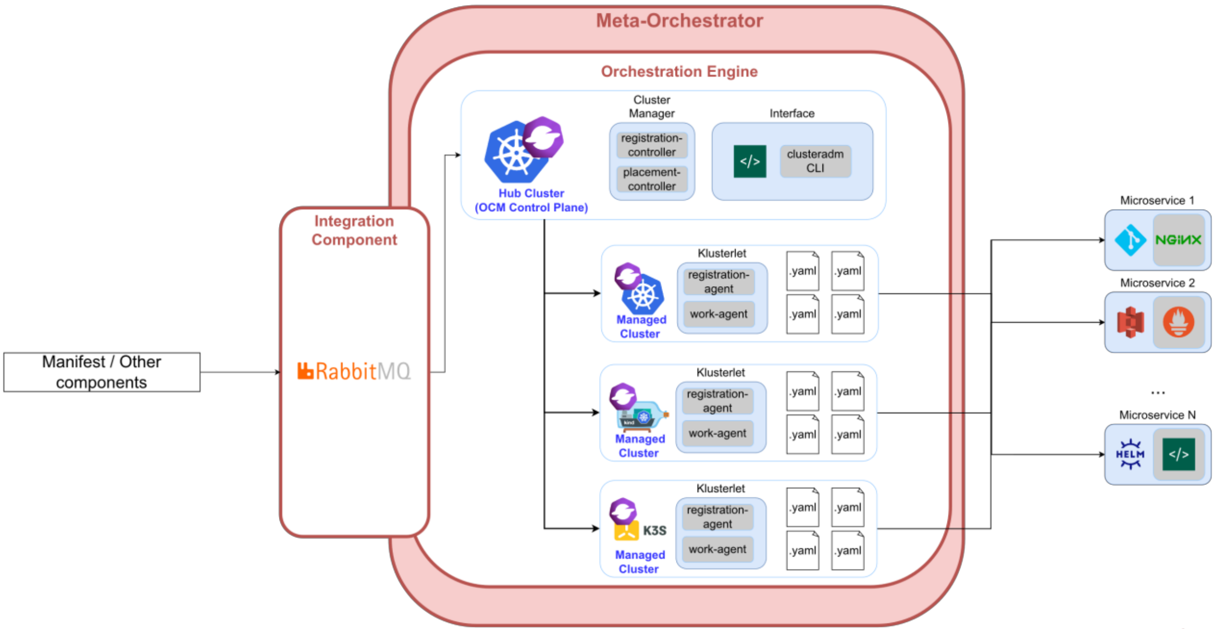


Figure 38: Meta-Orchestrator Workload Deployment: Orchestration Engine and Integration Component microservices subscription and deployment

Further integration with other components is granted if the manifest file used for deployment maintain the expected format. Following the CMDT descriptor file (defined in D2.1), the yaml might be shaped as follows:

```
name: Name
  description: Description of the service
  services: # List of services included
    - Name: service1
      Description: This is the first service that can be mapped to a LL
      #package: charts/service1-0.0.0.tgz
      Properties:  #List of [Property, value]
       - Property 1: Value 1
       - Property 2: Value 2
      Microservices: #List of µServices
      - Name: µService 1
        Description: Microservice 1 description on how to access it
        Endpoints:
         - http://service1/microservice1
      - Name: µService 2
        Description: Microservice 2 description on how to access it
        Endpoints:
         - http://service1/microservice2
```

6.3 Workload migration

The migration process from meta-Orchestrator to mNCC revolves around seamlessly transitioning workloads from the cloud to edge infrastructure, leveraging Kubernetes (K8s, microK8s, K3s, etc.) across various administrative domains. Emphasizing intent-based networking/computing, including microslicing and multipath integration, ensures efficient data processing across the continuum of IoT to Edge to Cloud.

A comprehensive analysis of the existing cloud infrastructure precedes the development of a meticulous migration blueprint, integrating meta-Orchestrator with mNCC to address the complexities of intent-based networking. This entails establishing seamless communication channels between the edge nodes and the meta-Orchestrator, facilitating the secure transfer of workloads from the cloud to the edge infrastructure.

The integration process focuses on configuring the edge nodes to support the seamless deployment of Kubernetes-based services, synchronizing operations with the meta-Orchestrator through robust communication protocols and secure data transmission channels. This ensures the smooth transition of workloads and data processing from the cloud to the edge infrastructure.

With a unified multi-domain orchestration strategy in place, the integration of meta-Orchestrator and mNCC within each administrative domain creates a cohesive control plane that centralizes the management and oversight of network and computing resources, fostering consistency and coherence in workload deployment across the distributed infrastructure.

Intent-based networking and computing are optimized through the implementation of precise resource allocation policies and advanced microslicing techniques. Strategies such as multipath routing are employed to optimize data transfer paths, enhancing the overall efficiency and responsiveness of the system in adapting to varying workload demands and network conditions.

Continuous monitoring mechanisms are then deployed to track the performance of migrated workloads and the network infrastructure, ensuring dynamic optimization and proactive resource management throughout the IoT to Edge to Cloud continuum.

6.4 Workload monitoring

In NEMO the core idea behind PPEF system is to enable the monitoring of NEMO-hosted services' performance metrics that are not only adhere to a business logic agnostic SLA definition but to proceed on low-level SLA definition that tackle the intent-based approach followed by the project and thus deliver SLO based, performance-driven and orchestration aware workload lifecycle management.

In the framework of PPEF development activities an initial deployment and configuration of CNCF accepted monitoring tools focusing as well on energy consumption ones namely Kepler and Scaphandre has been conducted. A Federated architecture of Prometheus resource monitoring tools have been defined and a prototype has been developed. Moreover, initial measurements and results of monitored NEMO-hosted resources have been collected and integrated with visualization tool.

The purpose of this prototype is to illustrate on one hand the Federated monitoring architectural approach that has been adopted by PPEF and on the other to provide a hands-on demonstration of the capabilities of the Cloud infrastructure monitoring tools both for regular and energy specific metrics that will drive the NEMO services policy enforcement procedures. The abovementioned implementation will act as the foundation upon which the PPEF system will be structured. Special focus is laid on the energy consumption monitoring tools that the PPEF incorporates. Finally, the integrated Grafana dashboard is utilized for the visualization of a variety of collected resource measurements.

In this prototype each local Cloud/Edge infrastructure is monitored by a local Prometheus instance. A federated Prometheus is utilized to pull observed metrics from the various local Prometheus instances. A simple illustration of the implemented prototype architecture is presented in Figure 41.

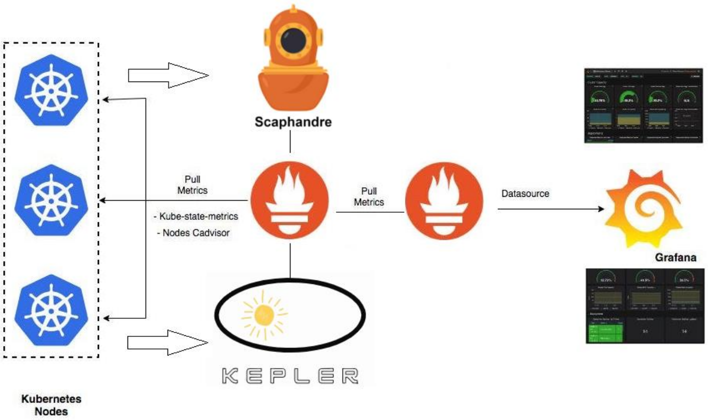
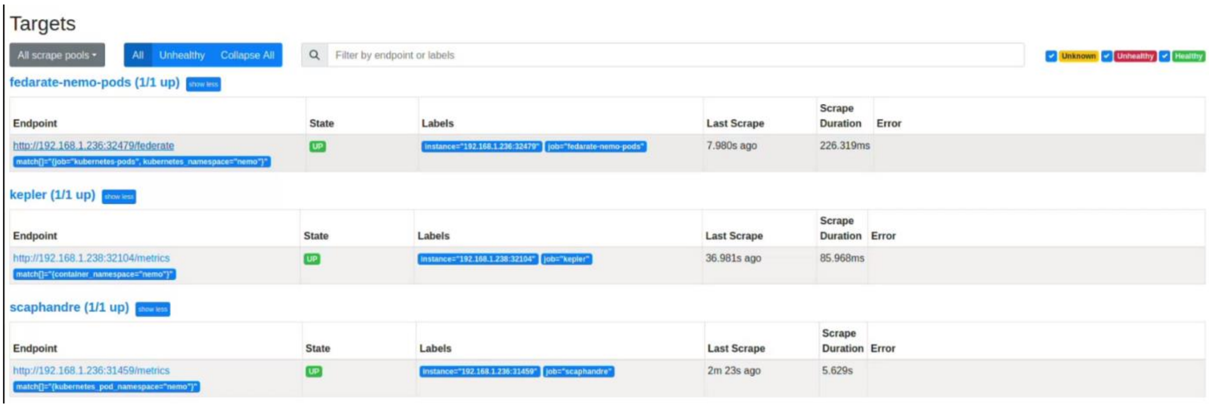


Figure 39: Federated Prometheus setup in K8s orchestrated cluster

In the abovementioned PPEF prototype, each local Prometheus instance is integrated with the Kepler and Scaphandre data exporters. Figure 40 presents the endpoints that are realized by the local Prometheus and that can be consumed by the Federated one.



Targets

fedarate-nemo-pods (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.1.236:32479/federate match[]="{job="kubernetes-pods", kubernetes_namespace="nemo"}"	UP	instance="192.168.1.236:32479" job="fedarate-nemo-pods"	7.980s ago	226.319ms	

kepler (1/1 up) show less

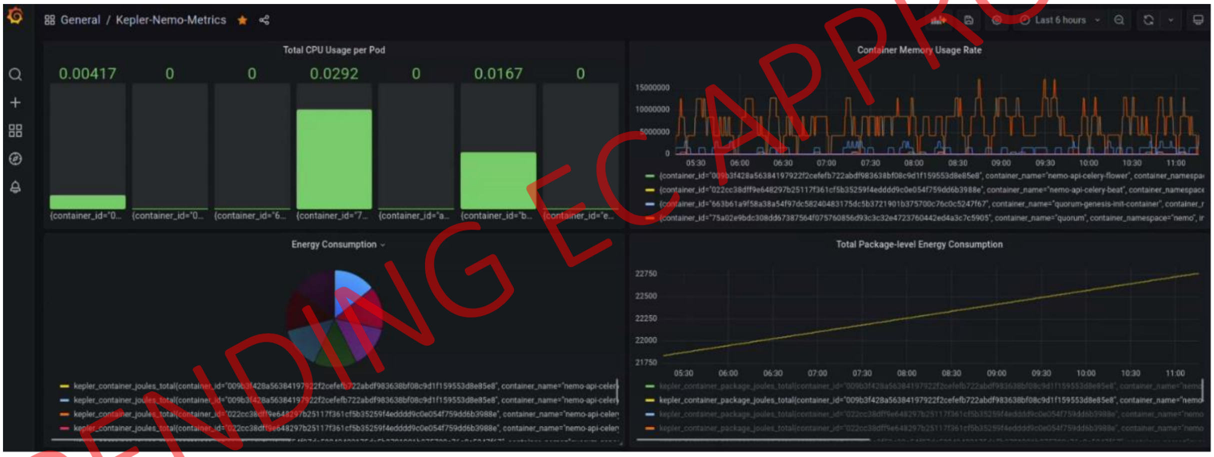
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.1.238:32104/metrics match[]="{container_namespace="nemo"}"	UP	instance="192.168.1.238:32104" job="kepler"	36.981s ago	85.968ms	

scaphandre (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.1.236:31459/metrics match[]="{kubernetes_pod_namespace="nemo"}"	UP	instance="192.168.1.236:31459" job="scaphandre"	2m 23s ago	5.629s	

Figure 40: Local Prometheus endpoints

Lastly, Figure 41 illustrates a sample of the visualization capabilities of Grafana dashboard that is integrated with the Prometheus deployed instances. In the picture we observe metrics that concern the CPU usage per pod, the container memory usage rate, the energy consumption per container and total energy consumption at package-level.



General / Kepler-Nemo-Metrics
Last 6 hours
Total CPU Usage per Pod
0.00417
0
0
0.0292
0
0.0167
0
{container_id="0...
{container_id="0...
{container_id="6...
{container_id="7...
{container_id="a...
{container_id="b...
{container_id="e...
Container Memory Usage Rate
15000000
10000000
5000000
0
05:30
06:00
06:30
07:00
07:30
08:00
08:30
09:00
09:30
10:00
10:30
11:00
{container_id="009b3f428a56384197922f2cefefb722abdf983638bf08c9d1f159553d8e85e8", container_name="nemo-api-celery-flower", container_namespa
{container_id="022cc38dff9e648297b25117f361cf5b35259f4edddd9c0e054f759dd6b3988e", container_name="nemo-api-celery-beat", container_namespace
{container_id="663b61a9f58a38a54f97dc58240483175dc5b3721901b375700c76c0c5247f67", container_name="quorum-genesis-init-container", container_r
{container_id="75a02e9bdc308dd67387564f075760856d93c3c32e4723760442ed4a3c7c5905", container_name="quorum", container_namespace="nemo", ir
Energy Consumption
kepler_container_joules_total{container_id="009b3f428a56384197922f2cefefb722abdf983638bf08c9d1f159553d8e85e8", container_name="nemo-api-celer
kepler_container_joules_total{container_id="009b3f428a56384197922f2cefefb722abdf983638bf08c9d1f159553d8e85e8", container_name="nemo-api-celer
kepler_container_joules_total{container_id="022cc38dff9e648297b25117f361cf5b35259f4edddd9c0e054f759dd6b3988e", container_name="nemo-api-celer
kepler_container_joules_total{container_id="022cc38dff9e648297b25117f361cf5b35259f4edddd9c0e054f759dd6b3988e", container_name="nemo-api-celer
Total Package-level Energy Consumption
22750
22500
22250
22000
21750
05:30
06:00
06:30
07:00
07:30
08:00
08:30
09:00
09:30
10:00
10:30
11:00
kepler_container_package_joules_total{container_id="009b3f428a56384197922f2cefefb722abdf983638bf08c9d1f159553d8e85e8", container_name="nemo
kepler_container_package_joules_total{container_id="009b3f428a56384197922f2cefefb722abdf983638bf08c9d1f159553d8e85e8", container_name="nemo
kepler_container_package_joules_total{container_id="022cc38dff9e648297b25117f361cf5b35259f4edddd9c0e054f759dd6b3988e", container_name="nemo
kepler_container_package_joules_total{container_id="022cc38dff9e648297b25117f361cf5b35259f4edddd9c0e054f759dd6b3988e", container_name="nemo

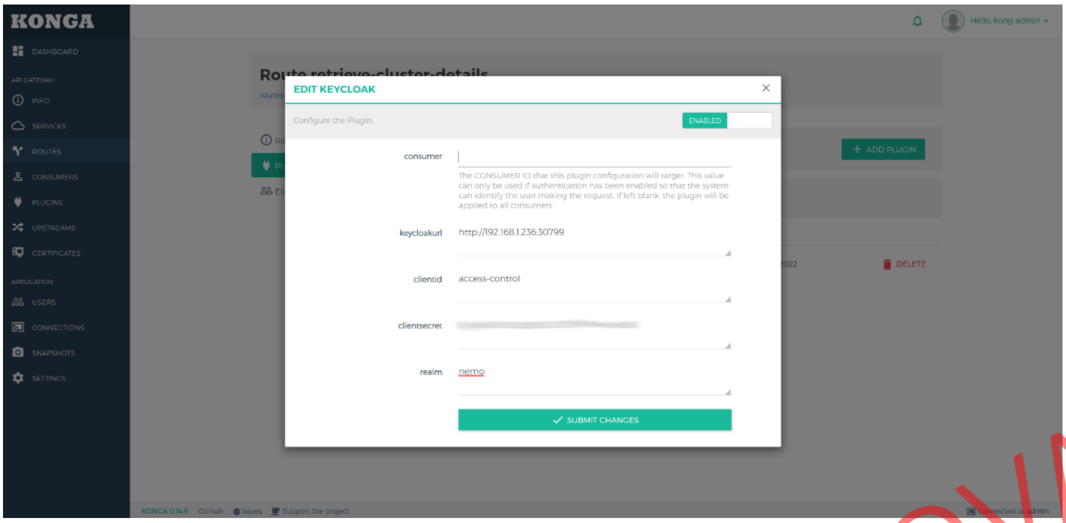
Figure 41: Grafana visualization on collected metrics

6.5 Secure communications

6.5.1 NEMO Access control

As part of the PoC implementation, the NAC operation supporting AAA-based control is presented. For demonstration purposes, the first version of the NEMO Access Control, supporting access control based on the oAuth 2.0 has been deployed and tested in SYN premises. NAC was deployed in a namespace dedicated to NEMO, within a K8s cluster. In the same cluster and namespace, a Keycloak installation has been used to integrate with the oAuth 2.0 plugin. In addition, Konga [32] has been deployed and used as an open-source management tool and GUI for the NAC API Gateway.

As soon as the aforementioned software modules are deployed and operational, the API configuration may take place in the API Gateway. This refers to setting up services and routes. In Kong Gateway, a service is an abstraction of an existing upstream application. Services can store collections of objects like plugin configurations, and policies, and they can be associated with routes. A route is a path to a resource within an upstream application. Routes are added to services to allow access to the underlying application. In Kong Gateway, routes typically map to endpoints that are exposed through the Kong Gateway application. Routes can also define rules that match requests to associated services. Because of this, one route can reference multiple endpoints [33]. Figure 42 depicts the oAuth 2.0 plugin configuration in the NAC API Gateway.



EDIT KEYCLOAK

Configure the Plugin. ENABLED

consumer:

The CONSUMER ID that this plugin configuration will target. This value can only be used if authentication has been enabled so that the system can identify the user making the request. If left blank, the plugin will be applied to all consumers.

keycloakurl: http://192.168.1.236:30799

clientid: access-control

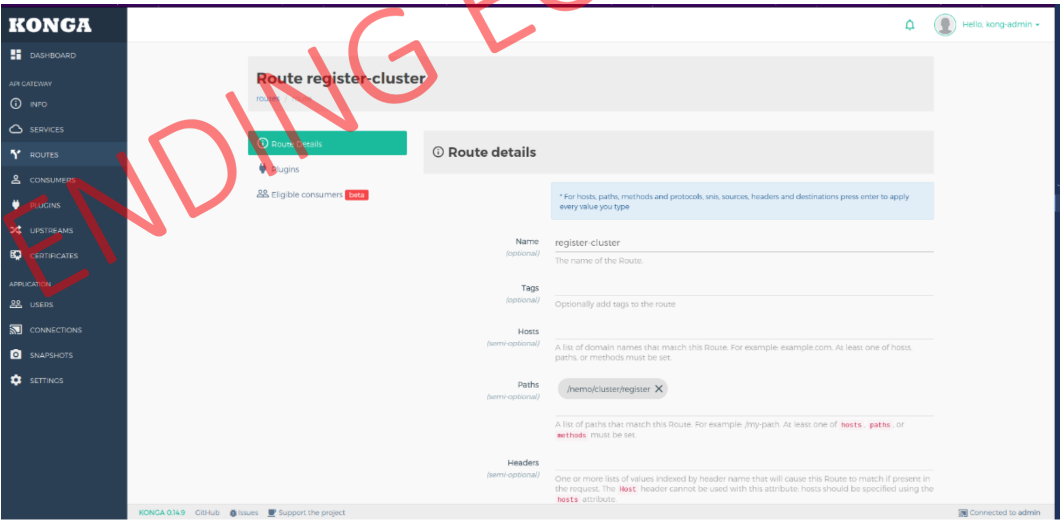
clientsecret: [redacted]

realm: nemo

✓ SUBMIT CHANGES

Figure 42: Configuring Keycloak (oAuth 2.0) plugin in NAC through Konga

As an upstream NEMO service, the current version of the MOCA API has been used. In the following, the NAC configuration and access control for two endpoints is presented, responsible for cluster registration (*register-cluster* endpoint of MOCA) and retrieval of -already registered- cluster's information (*retrieve-cluster-details* endpoint of MOCA). The configuration of relevant services and routes is presented in Figure 43 to Figure 46.



Route register-cluster

Route details

* For hosts, paths, methods and protocols, snis, sources, headers and destinations press enter to apply every value you type

Name (optional): register-cluster

The name of the Route.

Tags (optional):

Optionally add tags to the route

Hosts (semi-optional):

A list of domain names that match this Route. For example: example.com. At least one of hosts, paths, or methods must be set.

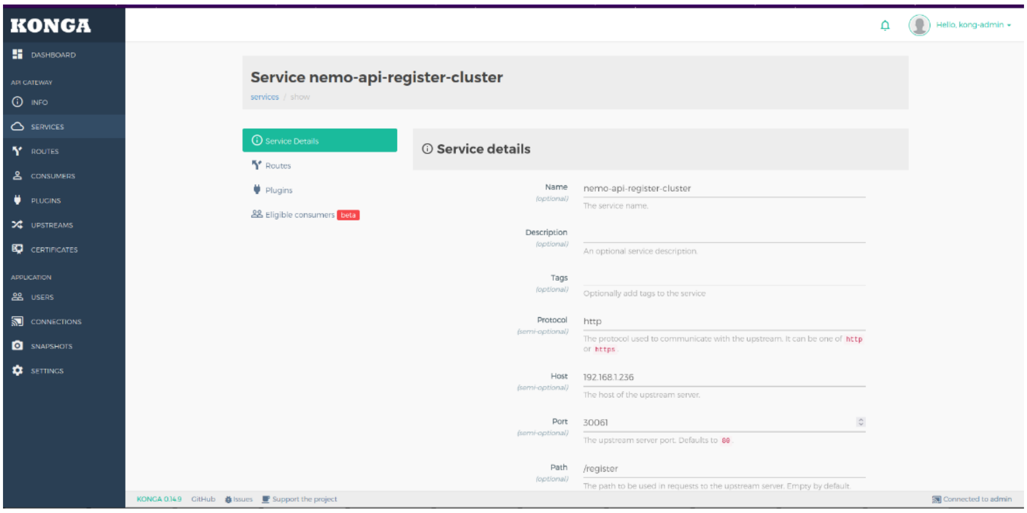
Paths (semi-optional): /nemo/cluster/register

A list of paths that match this Route. For example: /my-path. At least one of `hosts`, `paths`, or `methods` must be set.

Headers (semi-optional):

One or more lists of values indexed by header name that will cause this Route to match if present in the request. The `Host` header cannot be used with this attribute: hosts should be specified using the `hosts` attribute.

Figure 43: Registering route for *register-cluster* endpoint of MOCA



Service nemo-api-register-cluster

services / show

Service details

Name (optional): nemo-api-register-cluster
The service name.

Description (optional):
An optional service description.

Tags (optional):
Optionally add tags to the service

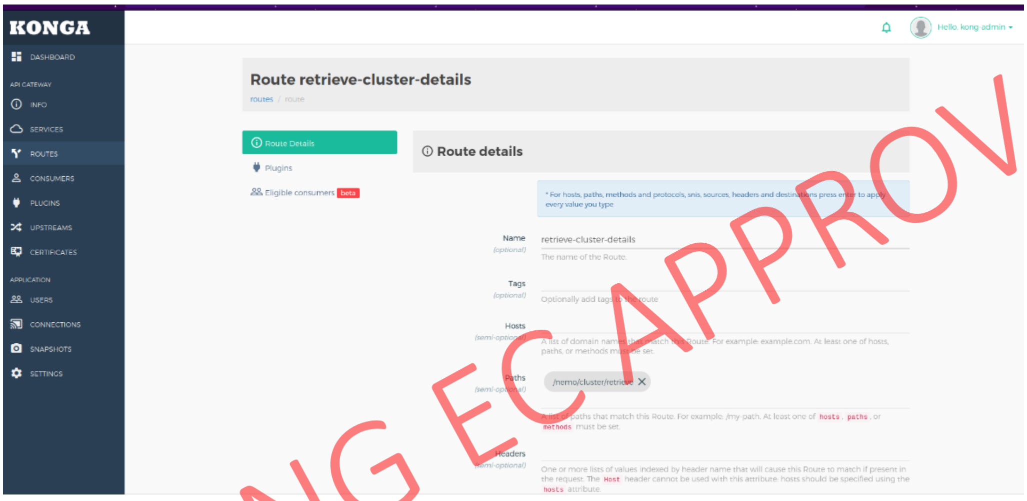
Protocol (semi-optional): http
The protocol used to communicate with the upstream. It can be one of `http` or `https`.

Host (semi-optional): 192.168.1.236
The host of the upstream server.

Port (semi-optional): 30061
The upstream server port. Defaults to `80`.

Path (optional): /register
The path to be used in requests to the upstream server. Empty by default.

Figure 44: Configuring API Gateway service for protecting the *register-cluster* endpoint of MOCA



Route retrieve-cluster-details

routes / route

Route details

* For hosts, paths, methods and protocols, snis, sources, headers and destinations press enter to apply every value you type

Name (optional): retrieve-cluster-details
The name of the Route.

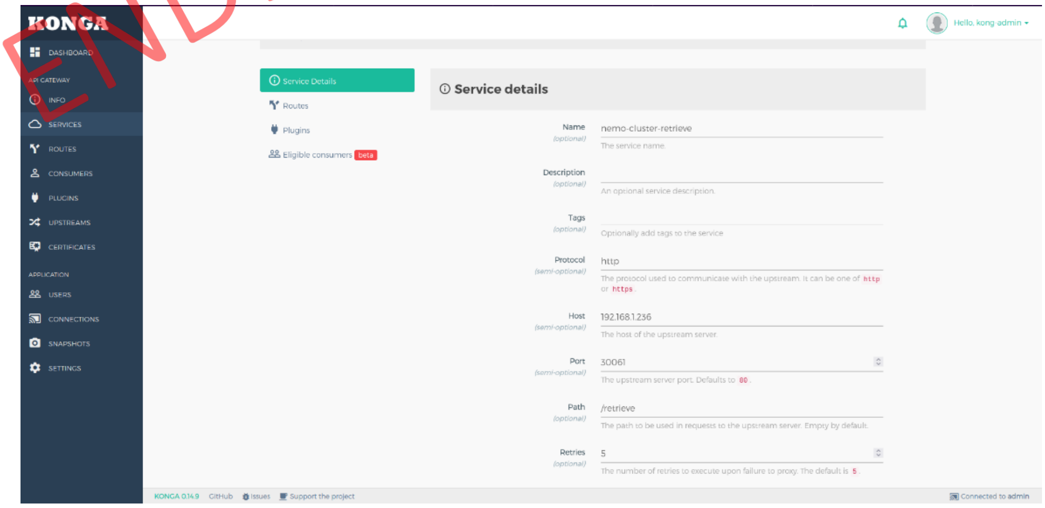
Tags (optional):
Optionally add tags to the route

Hosts (semi-optional):
A list of domain names that match this Route. For example: example.com. At least one of hosts, paths, or methods must be set.

Paths (semi-optional): /nemo/cluster/retrieve
A list of paths that match this Route. For example: /my-path. At least one of `hosts`, `paths`, or `methods` must be set.

Headers (semi-optional):
One or more lists of values indexed by header name that will cause this Route to match if present in the request. The `Host` header cannot be used with this attribute: hosts should be specified using the `hosts` attribute.

Figure 45: Registering route for *retrieve-cluster-details* endpoint of MOCA



Service details

Name (optional): nemo-cluster-retrieve
The service name.

Description (optional):
An optional service description.

Tags (optional):
Optionally add tags to the service

Protocol (semi-optional): http
The protocol used to communicate with the upstream. It can be one of `http` or `https`.

Host (semi-optional): 192.168.1.236
The host of the upstream server.

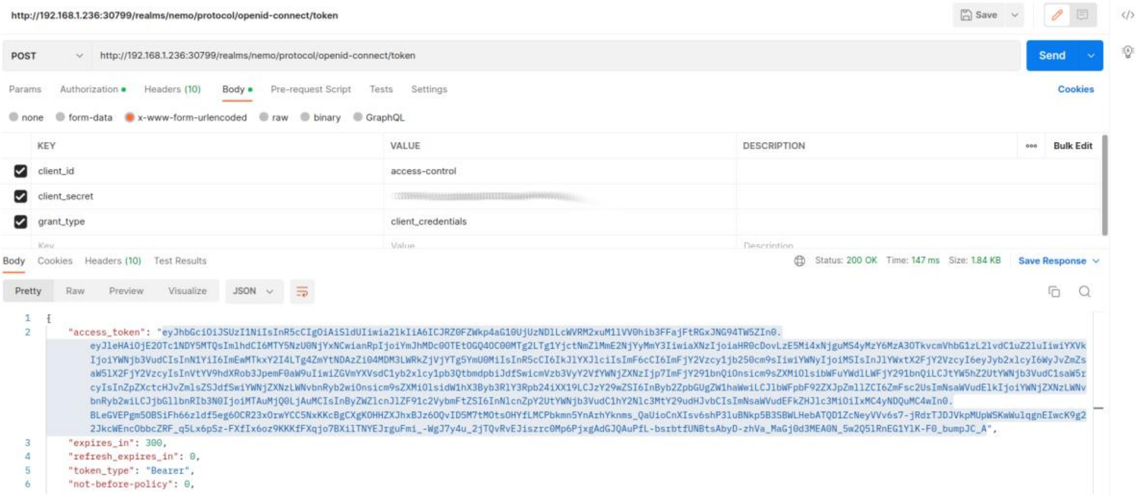
Port (semi-optional): 30061
The upstream server port. Defaults to `80`.

Path (optional): /retrieve
The path to be used in requests to the upstream server. Empty by default.

Retries (optional): 5
The number of retries to execute upon failure to proxy. The default is `5`.

Figure 46: Configuring API Gateway service for protecting the *retrieve-cluster-details* endpoint of MOCA

The NAC client application attempting to access the protected MOCA endpoints is simulated through POSTMAN, as depicted in Figure 47 for the client receiving an access token.



http://192.168.1.236:30799/realms/nemo/protocol/openid-connect/token

POST http://192.168.1.236:30799/realms/nemo/protocol/openid-connect/token

Body: ○ none ○ form-data ● x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

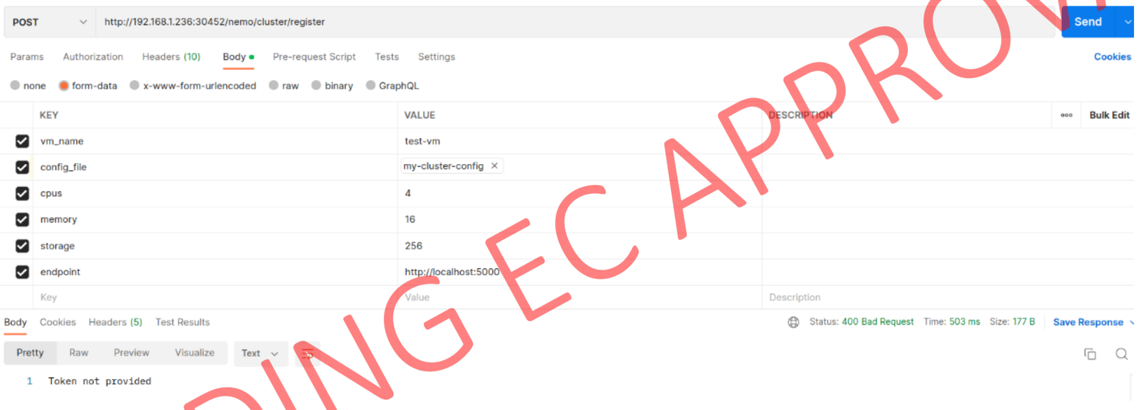
	KEY	VALUE	DESCRIPTION
☑	client_id	access-control	
☑	client_secret	[redacted]	
☑	grant_type	client_credentials	

Status: 200 OK Time: 147 ms Size: 1.84 KB

```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJRZ0FZWkp4aG10UjUzNDlLcWVRM2xuM1lVV0hib3FFajFtRGxJNG94TW5ZIn0.
        eyJleHAiOjE2OTc1NDY5MTQsImlhdCI6MTY5NzU0NjYxNCwianRpIjoiYmJhMDc0OTEtOGQ4OC00MTg2LTg1YjctNmZlMmE2NjYyMmY3IiwiaXNzIjoiaHR0cDovLzE5Mi4xNjguMS4yMzY6MzA3OTkvcmVhbG1zL2lvdC1uZ2luIiwiYXVk
        IjoiYWNjb3VudCIsInN1YiI6ImEwMTkxY2I4LTg4ZmYtNDAzZi04MDM3LWRkZjVjYTg5YmU0MiIsInR5cCI6IkJlYXJlciIsImF6cCI6ImFjY2Vzcy1jb250cm9sIiwiYWNyIjoiMSIsInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJvZmZs
        aW5lX2FjY2VzcyIsInVtYV9hdXRob3JpemF0aW9uIiwiZGVmYXVsdC1yb2xlcy1pb3QtbmdpbiJdfSwicmVzb3VyY2VfYWNjZXNzIjp7ImFjY291bnQiOnsicm9sZXMiOlsibWFuYWdlLWFjY291bnQiLCJtYW5hZ2UtYWNjb3VudC1saW5r
        cyIsInZpZXctcHJvZmlsZSJdfSwiYWNjZXNzLWNvbnRyb2wiOnsicm9sZXMiOlsidW1hX3Byb3RlY3Rpb24iXX19LCJzY29wZSI6InByb2ZpbGUgZW1haWwiLCJlbWFpbF92ZXJpZmllZCI6ZmFsc2UsImNsaWVudElkIjoiYWNjZXNzLWNv
        bnRyb2wiLCJjbGllbnRIb3N0IjoiMTAuMjQ0LjAuMCIsInByZWZlcnJlZF91c2VybmFtZSI6InNlcnZpY2UtYWNjb3VudC1hY2Nlc3MtY29udHJvbCIsImNsaWVudEFkZHJlc3MiOiIxMC4yNDQuMC4wIn0.
        BLeGVEPgm5OBSiFh66zldf5eg6OCR23xOrwYCC5NxKKcBgCXgKOHHZXJhxBJz6OQvID5M7tMOtsOHYfLMCPbkmn5YnArhYknms_QaUioCnXIsv6shP3luBNkp5B3SBWLHebATQD1ZcNeyVVv6s7-jRdrTJDJVkpMUpWSKwWulqgnEIwcK9g2
        2JkcWEncObbcZRF_q5Lx6pSz-FXfIx6oz9KKKfFXqjo7BXilTNYEJrguFmi_-WgJ7y4u_2jTQvRvEJiszrc0Mp6PjxgAdGJQAuPfL-bsrbtfUNBtsAbyD-zhVa_MaGj0d3MEA0N_5w2Q5lRnEG1YlK-F0_bumpJC_A",
    "expires_in": 300,
    "refresh_expires_in": 0,
    "token_type": "Bearer",
    "not-before-policy": 0,
```

Figure 47: Simulating client app requesting access token in POSTMAN

Then, the client application is presented to denied access when no or inactive tokens are provided (Figure 48, Figure 49, Figure 51, Figure 52), while access is granted when a valid token is provided (Figure 50 & Figure 53).



POST http://192.168.1.236:30452/nemo/cluster/register

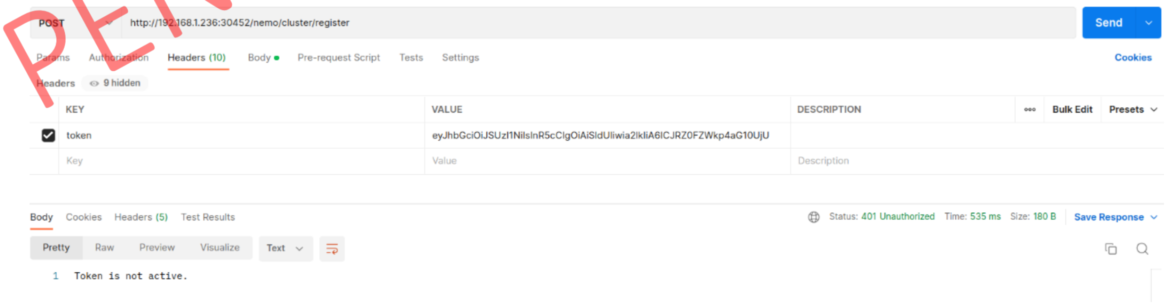
Body: ○ none ● form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL

	KEY	VALUE	DESCRIPTION
☑	vm_name	test-vm	
☑	config_file	my-cluster-config	
☑	cpus	4	
☑	memory	16	
☑	storage	256	
☑	endpoint	http://localhost:5000	

Status: 400 Bad Request Time: 503 ms Size: 177 B

```
Token not provided
```

Figure 48: NAC client app fails to access the *register-cluster* endpoint of MOCA due to token not provided



POST http://192.168.1.236:30452/nemo/cluster/register

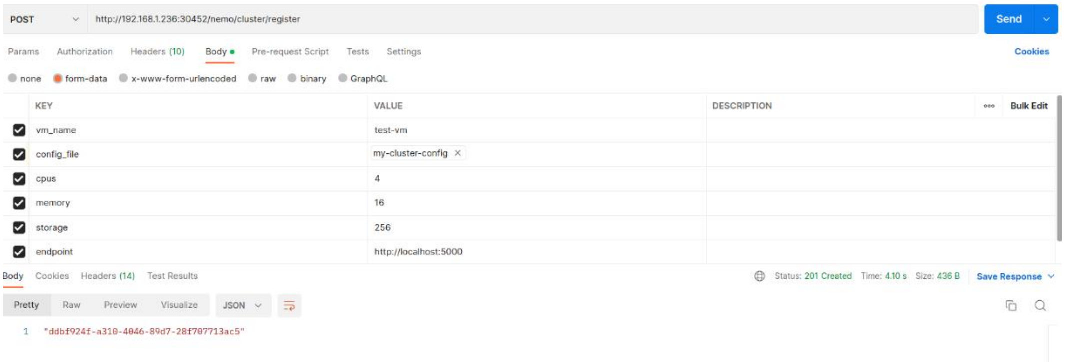
Headers ⊘ 9 hidden

	KEY	VALUE	DESCRIPTION
☑	token	eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJRZ0FZWkp4aG10UjU	

Status: 401 Unauthorized Time: 535 ms Size: 180 B

```
Token is not active.
```

Figure 49: NAC client app fails to access the *register-cluster* endpoint of MOCA due to provided token being inactive



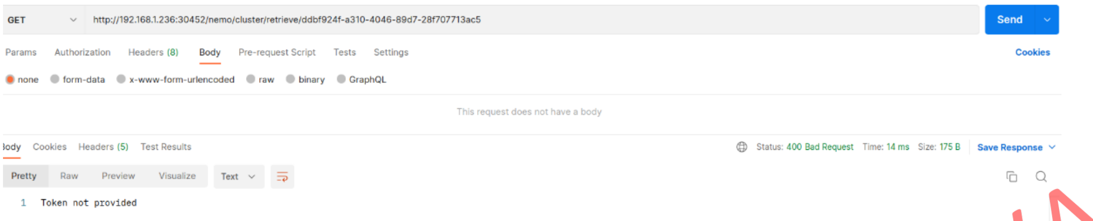
http://192.168.1.236:30452/nemo/cluster/register

	KEY	VALUE	DESCRIPTION
☑	vm_name	test-vm	
☑	config_file	my-cluster-config	
☑	cpus	4	
☑	memory	16	
☑	storage	256	
☑	endpoint	http://localhost:5000	

Status: 201 Created Time: 4.10 s Size: 436 B

```
"ddbf924f-a310-4046-89d7-28f707713ac5"
```

Figure 50: NAC client app successfully registers a cluster in MOCA, as AAA controls have been passed



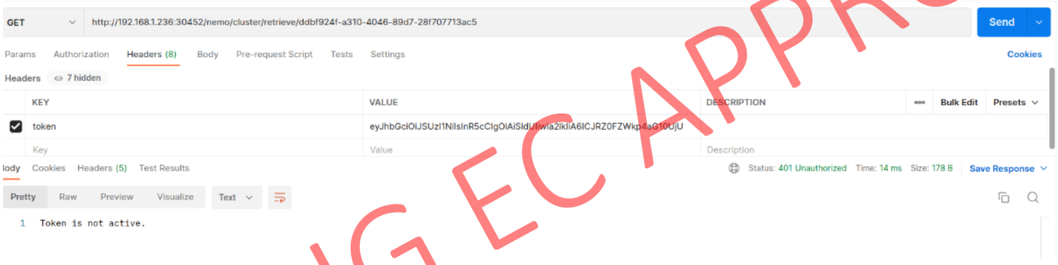
http://192.168.1.236:30452/nemo/cluster/retrieve/ddbf924f-a310-4046-89d7-28f707713ac5

This request does not have a body

Status: 400 Bad Request Time: 14 ms Size: 175 B

```
Token not provided
```

Figure 51: NAC client app fails to access the *retrieve-cluster-details* endpoint of MOCA due to token not provided



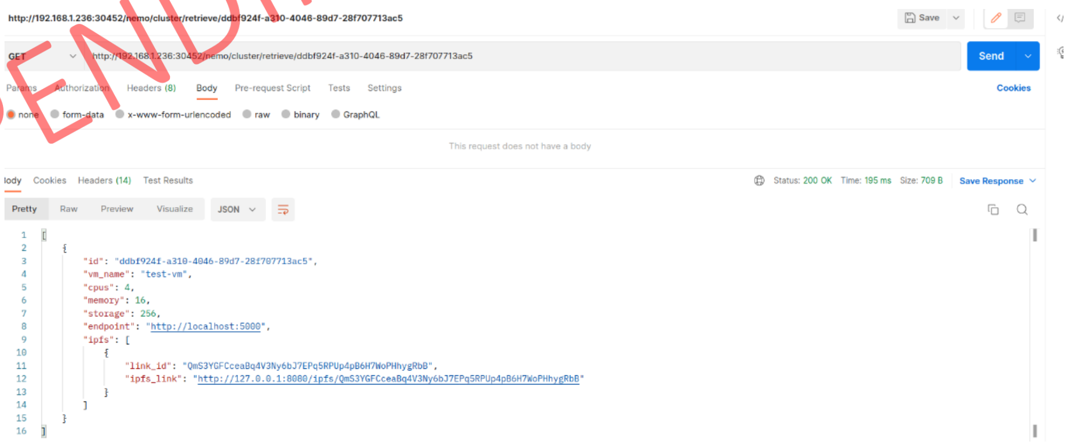
http://192.168.1.236:30452/nemo/cluster/retrieve/ddbf924f-a310-4046-89d7-28f707713ac5

	KEY	VALUE	DESCRIPTION
☑	token	eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJRZ0FZWkp4aG10UjU	
	Key	Value	Description

Status: 401 Unauthorized Time: 14 ms Size: 178 B

```
Token is not active.
```

Figure 52: NAC client app fails to access the *retrieve-cluster-details* endpoint of MOCA due to token provided being inactive



http://192.168.1.236:30452/nemo/cluster/retrieve/ddbf924f-a310-4046-89d7-28f707713ac5

This request does not have a body

Status: 200 OK Time: 195 ms Size: 709 B

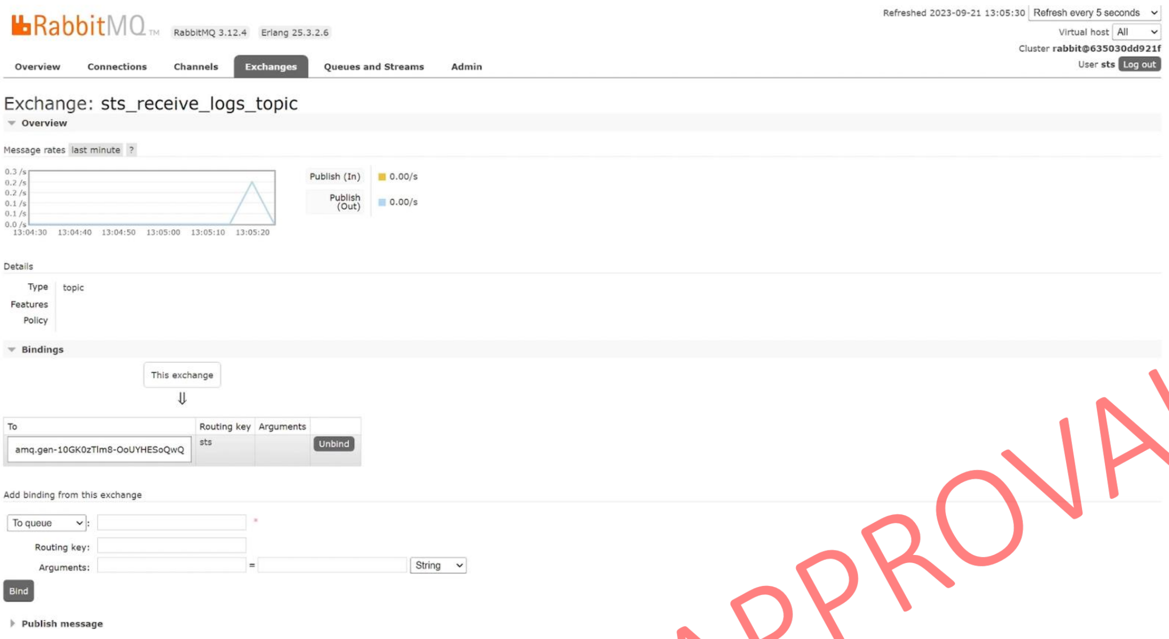
```
[
    {
        "id": "ddbf924f-a310-4046-89d7-28f707713ac5",
        "vm_name": "test-vm",
        "cpus": 4,
        "memory": 16,
        "storage": 256,
        "endpoint": "http://localhost:5000",
        "ipfs": [
            {
                "link_id": "QmS3YGFCceaBq4V3Ny6bJ7EPq5RPUp4pB6H7WoPHhygRbB",
                "ipfs_link": "http://127.0.0.1:8080/ipfs/QmS3YGFCceaBq4V3Ny6bJ7EPq5RPUp4pB6H7WoPHhygRbB"
            }
        ]
    }
]
```

Figure 53: NAC client app receives response by the *retrieve-cluster-details* endpoint of MOCA, as valid token has been provided

6.5.2 Intercommunication Management Module

The Message broker of NEMO which is based in RabbitMQ is currently placed in a temporary open GitHub repository[86] and it will be moved to the project repository when fully integrated with the rest of security modules.

Below the interface of the intercommunication management module.



Refreshed 2023-09-21 13:05:30 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@635030dd921f**

User **sts**

Exchange: sts_receive_logs_topic

Publish (In) 🟨 0.00/s

Publish (Out) 🟦 0.00/s

Details

Type	topic
Features	
Policy	

To	Routing key	Arguments	
amq.gen-10GK0zTlm8-OoUYHESoQwQ	sts		Unbind

Add binding from this exchange

To queue: *

Routing key:

Arguments: = String

Figure 54 : Intercommunication Management Interface

[86] https://github.com/mikesmirlis/rabbitmq-poc

Conclusions

In this document, we have presented the outcomes of the work on the *NEMO Kernel* so far. Following the structure of the project, this first deliverable presents the background topics and state-of-the-art, but most important the plans and architecture for the solutions that will form the NEMO Kernel. In summary, the project's progress in various tasks showcases significant strides towards establishing a robust and secure computing environment within the NEMO ecosystem.

Task 3.1 has paved the way for a Secure Execution Environment, enhancing Kubernetes with isolated Unikernel technology and trusted execution runtimes. With the completion of background research and solution architecture, coupled with the prototype development, the project is poised for the crucial phase of component development and integration into the NEMO Kernel. Simultaneously, Task 3.2 emphasizes the project's ethical commitment through the Privacy and Policy Enforcement Framework (PPEF), ensuring GDPR compliance and upholding the highest standards of data protection for NEMO-hosted services. Task 3.3 delves into cybersecurity aspects, focusing on authentication, access control, and real-time operating system monitoring. These cutting-edge measures enhance the overall security posture of NEMO services. Lastly, Task 3.4 highlights the pivotal role of the NEMO meta-Orchestrator, which stands as the operational core of the ecosystem. Empowered by advanced Orchestration Engine and Integration Component, it efficiently coordinates intricate workflows across IoT, Edge, and Cloud domains, reinforcing the project's dedication to seamless computing.

By stating a detailed proof-of-concept strategy, the whole concept of the NEMO-Kernel becomes verifiable. This strategy contains details on the deployment, migration and monitoring of the workloads, as well as details and instructions on the secure communication channels.

With all four tasks in motion, NEMO is on track to revolutionize computing paradigms, ensuring both security and efficiency across diverse domains.

References

[1] NEMO, “D1.1 - Definition and analysis of use cases and GDPR compliance,” HORIZON - 101070118 - NEMO Deliverable Report, 2023.

[2] NEMO, “D1.2 - NEMO meta-architecture, components and benchmarking. Initial version,” HORIZON - 101070118 - NEMO Deliverable Report, 2023.

[3] Canonical, “Kubernetes and cloud native operations report 2022,” 2022.

[4] Cloud Native Computing Foundation, “Confidential Containers Project,” 2023. [Online]. Available: https://github.com/confidential-containers.

[5] Open Container Initiative, “Open Container Initiative Project Repository,” 2023. [Online]. Available: https://github.com/opencontainers/runtime-spec/blob/main/spec.md.

[6] Institute for Automation of Complex Power Systems, “runh Project Website,” [Online]. Available: https://github.com/hermit-os/runh. [Accessed 10 2023].

[7] Kata Containers Community, “Kata Containers Project Repository,” [Online]. Available: https://github.com/kata-containers/kata-containers. [Accessed 10 10 2023].

[8] H. B. a. D. T. a. M. R. a. N. S. a. W. Pan, *Remote ATtestation procedureS (RATS) Architecture,* RFC Editor, 2023, p. 46.

[9] T. Hoffman, “Introduction to CRIU and Live Migration,” [Online]. Available: https://medium.com/@talhof8/introduction-to-criu-and-live-migration-cd4a6d11afb6.

[10] The CRIU Project, “CRIU: Usage scenarios,” [Online]. Available: https://criu.org/Usage_scenarios.

[11] S. Mohanty, “Kubernetes Checkpointing — A Definitive Guide!,” [Online]. Available: https://faun.pub/kubernetes-checkpointing-a-definitive-guide-33dd1a0310f6.

[12] J. H. J. J. J. J. J. H. H. M. Bongjae Kim, “A Dynamic Checkpoint Interval Decision Algorithm for Live Migration-Based Drone-Recovery System,” *drones,* vol. 7, no. 5, 2023.

[13] C. F. Lung, “Migrating pods between Kubernetes nodes (without killing them),” 23 04 2023. [Online]. Available: https://chuniversiteit.nl/papers/seamless-pod-migration-in-kubernetes.

[14] NEMO, “D5.1 - Living Labs and Data Management Plan (DMP). Initial version,” HORIZON - 101070118 - NEMO Deliverable Report, 2023.

[15] S. P. E. R. Chalee Vorakulpipat, “Usable comprehensive-factor authentication for a secure time attendance system,” *PeerJ Computer Science,* 2021.

[16] Gartner Peer Insigths, “Best Cloud-Native Application Protection Platforms Reviews 2023,” 2023. [Online]. Available: https://www.gartner.com/reviews/market/cloud-native-application-protection-platforms.

[17] Wikipedia, “Modular programming,” [Online]. Available: https://en.wikipedia.org/wiki/Modular_programming.

[18] C. Baldwin and K. Clark, Design Rules, Volume 1, Cambridge, MA: The MIT Press, 2000.

[19] Apple Developer, “Code Loading Programming Topics - Plug-in Architectures,” [Online]. Available: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/LoadingCode/Concepts/Plugins.html. [Accessed 17 10 2023].

[20] Redhat, “What does an API gateway do?,” 2019. [Online]. Available: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do. [Accessed 17 10 2023].

[21] Envoy, “Envoy proxy,” Lyft, [Online]. Available: https://www.envoyproxy.io. [Accessed 17 10 2023].

[22] Kong, “Kong Gateway,” [Online]. Available: https://docs.konghq.com/gateway/latest/. [Accessed 17 10 2023].

[23] AWS, “What is Amazon API Gateway?,” Amazon, [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html. [Accessed 17 10 2023].

[24] NGINX, “API Gateway,” [Online]. Available: https://www.nginx.com/learn/api-gateway/. [Accessed 17 10 2023].

[25] L. Crilly, “Deploying NGINX as an API Gateway, Part 1,” NGINX, 20 01 2021. [Online]. Available: https://www.nginx.com/blog/deploying-nginx-plus-as-an-api-gateway-part-1/. [Accessed 17 10 2023].

[26] Kong Inc., “Kong Gateway,” 2021. [Online]. Available: https://docs.konghq.com/gateway/. [Accessed Nov. 2021].

[27] OpenID, “OpenID Connect,” 2022. [Online]. Available: https://openid.net/connect/.

[28] Keycloak, “https://www.keycloak.org,” [Online]. Available: https://www.keycloak.org.

[29] D. Jones, “Container vs. Virtual Machines (VMs): What's the difference?,” NetApp, [Online]. Available: https://www.netapp.com/blog/containers-vs-vms/.

[30] C. Richardson, “Microservices.io,” [Online]. Available: https://microservices.io/.

[31] IEEE Innovation at Work, “Real-Life Use Cases for Edge Computing,” [Online]. Available: https://innovationatwork.ieee.org/real-life-edge-computing-use-cases/.

[32] P. Tselentis, “Konga,” 2020. [Online]. Available: https://pantsel.github.io/konga/.

[33] Kong, “Services and Routes,” [Online]. Available: https://docs.konghq.com/gateway/latest/get-started/services-and-routes/. [Accessed 17 10 2023].

[34] Kong Inc., “Comprehensive Getting Started Guide,” 2021. [Online]. Available: https://docs.konghq.com/gateway/2.6.x/get-started/comprehensive/. [Accessed Nov. 2021].

[35] Kong, “Write plugins in python,” [Online]. Available: https://docs.konghq.com/gateway/latest/plugin-development/pluginserver/python/. [Accessed 17 10 2023].

Annexes

Comparison Between Open Source and Vendor Solution of the Tetragon Kernel eBPF Probe

	Tetragon Open Source Software	Tetragon Enterprise	Tetragon Enterprise and Cilium Enterprise
Combined Runtime & Network Visibility	No	Yes (Some features require that cilium be deployed and the cilium api be enabled.. Specifically the ipcache functionality.)	Yes (All telemetry is available to tetragon in this case)
Runtime Visibility	Yes	Yes	Yes
Process Runtime Behavior	Yes - process_exec (K8s related info, pid, namespaces, caps, process related info, parent) Automatically, no CRDs needed	Yes - full ancestry tree	Yes
Syscall Runtime Behavior	Yes - with kprobe, tracepoints (K8s related info, pid, namespaces, caps, process related info, parent) It's happening with clusterwide CRD[87]s. Pod label filtering is OSS	Yes	Yes
Network Visibility			
L3 / L4 Network Visibility	Limited. Users of tetragon oss can hook any function or syscall in the kernel. This will be difficult to accomplish. Also users need to come up with their own CRDs. We just provided one example.	Yes - we get the full network socket lifecycle automatically no need for CRDs (process_connect, process_close, process_accept, process_listen) Annotations: DNS Names, endpoint names, pods, labels, etc.	Yes - and with timescape historic views of the same.
L7 Visibility - HTTP	No	Yes - via an in kernel HTTP parser, we get a separate event: ● HTTP	

Table 5: (1/2) Difference between Tetragon Open Source and Isovalent Vendor solution. This is taken as an example of the differences between OSS and vendor solutions in the CNAPP Business.

[87] Custom Resource Definition

	Tetragon OSS	Tetragon Enterprise	Tetragon Enterprise and Cilium Enterprise
		Includes all the process related and K8s aware metadata,	
L7 Visibility - DNS	No	Yes - via DNS parser in kernel, separate event: DNS - Watermarks -metrics	
L7 Visibility - HTTPS	No	Yes - via kTLS => kernel implementation of TLS, we don't do it in userland like Pixie does with uprobes and TLS libraries	
L7 Visibility - TLS	No	Yes - via TLS parser in kernel, full TLS handshake analysis. We get a separate event: ●TLS (client/server version, cipher, process related info, K8s info)	
SIEM Export	No. Events are sent to a file but we don't provide an integration like with enterprise.	Yes - fluentd with all the plugins fluentd can support (Splunk, elk, sumologic etc)	
Timescape Support	No Events are sent to a file but we don't provide integration or timescape as part of the deployment.	Yes. Collects only tegragon events.	Yes. Collects both Tetragon and hubble events.
File Integrity Monitoring	Yes - via a user friendly policy similar to osquery has 3 new events: file_create, file_modify, file_delete	Yes: - Digest (SHA 256)	
Runtime Enforcement	Yes - with kprobes and tracepoints, pod label filters, in kernel enforcement	Yes	
Threat Detection			
Rule Engine (WIP)	No	Yes - Combined Runtime and Network Security audit/enforcement	
Security Baseline Policy (WIP)	No	Yes - K8s aware SELinux (lock down malicious system calls) - dynamic	

Table 6: (2/2) Difference between Tetragon Open Source and Isovalent Vendor solution. This is taken as an example of the differences between OSS and vendor solutions in the CNAPP Business.