NEMO

Next Generation Meta Operating System

D3.3 Nemo Kernel Final Version

Document Identification			
Status	Final	Due Date	31/03/2025
Version	1.1	Submission Date	15/04/2025

Related WP	WP3	Document Reference	D3.2
Related Deliverable(s)	D3.1 Introducing NEMO Kernel, D3.2 NEMO Kernel Initial Version	Dissemination Level (*)	PU
Lead Participant	ATOS	Lead Author	Rubén Ramiro
Contributors	ATOS, COMS, ICCS, INTRA, RWTH, SYN, STS, TID, TSG	Reviewers	Alberto del Rio (UPM) Alexandru Vilceloiu (SIM) Mircea Vasile (SIM)

Keywords:
continuum, meta-Orchestration, micro-services, privacy & policy, cybersecurity, embeded systems

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.

(*) 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	
Name	**Partner**
Rubén Ramiro Ignacio Prusiel Enric Pages	ATOS
Matija Cankar Tomaž Bračič Gregor Cerar	COMS
Orestis Lagkas	ICCS
Dimitrios Skias	INTRA
Stefan Lankes Jonathan Klimt Martin Kröning	RWTH
Yannis Papaefstathiou	STS
Ilias Seitanidis	SYN
Mattin A. Elorza Alejandro Muñiz	TID
Nicolas Peiffer	TSG

Document History			
Version	**Date**	**Change editors**	**Changes**
0.1	08/01/2025	Rubén Ramiro (ATOS)	Initial version TOC creation
0.2	24/01/2025	Rubén Ramiro (ATOS)	First version of the executive Summary and Introduction section.
0.3	15/02/2025	Rubén Ramiro (ATOS), Nicolas Peiffer (TSG), Yannis Papaefstathiou (STS), and Jonathan Klimt (RWTH)	Added first contributions into Sections 2, 3,4 and 5.
0.4	25/02/2025	Mattin A. Elorz (TID), Ilias Seitanidis (SYN), and Dimitrios Skias (INTRA)	Added the Pod & Deployment Migration subsection, rebuilt section 2, and added section 2.3 Nerves architecture for the FOTA system, made changes in the KPIs section, and updated the format and style
0.5	17/03/2025	Orestis Lagkas (ICCS), Jonathan Klimt (RWTH), and Matija Cankar (COMS)	Added Pod & Deployment Migration subsection, rebuilt on section 2 and add sub section 2.3 Nerves architecture for FOTA system, changes in KPIs section, and updated format and style.
0.6	23/03/2025	Orestis Lagkas (ICCS), Jonathan Klimt (RWTH),	Restructured and added new content Section 2, updated Section 3, updated KPIs section.

		Dimitrios Skias (INTRA), Mattin A. Elorz (TID)	
0.7	25/03/2025	Matija Cankar (COMS), and Yannis Papaefstathiou (STS)	Section 2.4 Nerves moved to Section 6, added changes section 4, and restructured, updated KPIs section, and updated format and style.
0.8	01/04/2025	Rubén Ramiro, Enric Pages, Ignacio Prusiel (ATOS), and Dimitrios Skias (INTRA)	KPIs section changed, section 5 updated and added content in KPIs Section 8, updated Section 3.
0.9	08/04/2025	Rubén Ramiro (ATOS)	Added new content to Section 5, changed the main structure of the document.
1.0	11/04/2025	Rubén Ramiro (ATOS)	Aligned document with format and style, added deliverable conclusion.
1.1	15/04/2024	ATOS	Quality check and submission to EC

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Rubén Ramiro (ATOS)	11/04/2025
Quality manager	Rosana Valle Soriano (ATOS)	15/04/2025
Project Coordinator	Enric Pages (ATOS)	15/04/2025
Technical Manager	Ilias Seitanidis (SYN)	14/04/2025

Table of Contents

Document Information 2
Table of Contents 4
List of Tables 6
List of Figures 7
List of Acronyms 9
Executive Summary 11
1 Introduction 12
1.1 Purpose of the document 12
1.2 Relation to other project work 12
1.3 Structure of the document 12
2 Micro-services Secure Execution Environment 13
2.1 Overview 13
2.2 Architecture and Approach 13
2.2.1 Unikernel Runtime for Kubernetes 14
2.2.2 Secure Pod Attestation and Deployment Leveraging Trusted Execution Environments 18
2.2.3 Pod & Deployment Migration 20
2.2.4 meta-Orchestrator integration 25
2.3 Evaluation 29
2.3.1 Unikernel image size overhead 29
2.3.2 Secure pod attestation 30
2.4 Unikernel Deployment via SEE-Interface 33
2.5 Conclusion 35
3 Privacy & Policy Enforcement Framework 36
3.1 Overview 36
3.2 Architecture and Approach 36
3.3 NEMO workload monitoring 37
3.3.1 Intents and Expectations 37
3.4 NEMO Cluster monitoring 40
3.5 PPEF interactions and interfaces 40
3.5.1 Intent-based API 40
3.5.2 LCM 41
3.5.3 meta-Orchestrator 41
3.5.4 CMDT 41
3.5.5 CFDRL 41
3.5.6 MOCA 41

3.5.7 RabbitMQ ... 41
3.6 Conclusion ... 41
4 Cybersecurity & Digital Identity Attestation ... 42
4.1 Overview ... 42
4.2 Architecture and Approach ... 42
4.2.1 Identity and Management Module ... 42
4.2.2 News CNAPP & Software Supply Chain ... 46
4.3 Conclusion ... 50
5 NEMO meta-Orchestrator ... 51
5.1 Overview ... 51
5.2 Architecture and Approach ... 52
5.2.1 meta-Orchestrator Hub API ... 52
5.2.2 meta-Orchestrator Agent (MO Agent) ... 57
5.2.3 Deployment Controller (DC) ... 59
5.3 Conclusion ... 65
6 Secure Firmware Management on Far-Edge ... 66
6.1 Nerves architecture for FOTA system ... 66
6.1.1 The architecture of FOTA ... 66
6.1.2 Security and System Isolation ... 67
6.1.3 Firmware update sequence ... 67
6.1.4 FOTA PMU Cloud Service API documentation ... 68
7 Measurement and Validation ... 70
7.1 Micro-Services Secure Execution Environment KPIs ... 70
7.2 PRESS, Safety & Policy enforcement framework KPIs ... 70
7.3 Cybersecurity & Digital Identity Attestation KPIs ... 71
7.4 NEMO meta-Orchestrator KPIs ... 72
8 Conclusions ... 74
9 References ... 75
10 Annexes ... 76
10.1 Guidelines for TDX and Confidential Containers Technology ... 76
10.1.1 Enable the TDX in the BIOS ... 76

List of Tables

Table 1: Migration time breakdown for an example micro-service (nginx). ... 23
Table 2: Computing Workload Intent. ... 38
Table 3: Energy Carbon Efficiency Intent. ... 38
Table 4: Security Intent. ... 39
Table 5: Federated Learning Intent. ... 39
Table 6: Machine Learning Intent. ... 39
Table 7: Network Intent. ... 39
Table 8: Cluster registration KPIs ... 40
Table 9: Cluster Metrics ... 64
Table 10: Micro-Service Secure Execution Environment KPIs ... 70
Table 11: PRESS, Safety & Policy enforcement framework KPIs ... 70
Table 12: Cybersecurity & DIA KPIs ... 71
Table 13: MO KPIs ... 72
Table 14: Kata container installation. ... 78

List of Figures

Figure 1: Architecture of the Secure Execution Environment extension collection for Kubernetes. 14
Figure 2: Unikernel integration in Kubernetes. 14
Figure 3: Example of a Dockerfile to build a container image of a simple webserver as a Hermit unikernel 15
Figure 4: Unikernel runtime configuration for runh 16
Figure 5: Registration YAML of the container runtime runh with Kubernetes. 16
Figure 6: Deployment YAML of an example Unikernel service 17
Figure 7: List of running pods in a Kubernetes Cluster. In the middle it can be seen that a Unikernel is running as a Pod. 18
Figure 8: Kernel Parameters for KBS Integration. 18
Figure 9. KBS Architecture 19
Figure 10: Migration request using the see-ctl program 22
Figure 11: Deployment migration request 22
Figure 12: Single pod migration request 22
Figure 13: Deployment migration demo. 23
Figure 14: Single pod migration (no downtime) demo. 24
Figure 15: Single pod migration (with downtime) demo. 24
Figure 16: AMQP messages for pod metrics 25
Figure 17: AMQP messages for pod metrics 25
Figure 18: RabbitMQ exchanges for SEE metrics 25
Figure 19: Creation of a new RabbitMQ queue for message retrieval. 26
Figure 20: Binding a RabbitMQ message queue to the exchange. 26
Figure 21: Receiving resource metrics via RabbitMQ 26
Figure 22: AMQP flow for resource configuration. 27
Figure 23: RabbitMQ queues relevant for resource configuration. 27
Figure 24: Creation of a NGINX Pod via the SEE interface in the RabbitMQ web-ui 27
Figure 25: SEE-Interface response of the NGINX pod creation in the RabbitMQ web UI. 28
Figure 26: SEE-Interface failure response when deploying a pod via the RabbitMQ web UI. 28
Figure 27: Ubuntu based baseimage for Hermit containers. 29
Figure 28: Alpine Linux based base image for Hermit containers. 29
Figure 29: Image composition of a Hermit container based on the Alpine Linux baseimage. 30
Figure 30: Image composition of the nginx-alpine container. 30
Figure 31: Kernel Parameters for KBS Integration. 30
Figure 32: Authentication Request. 31
Figure 33: Key Retrieval. 31
Figure 34. CoCo Resource Consumption. 31
Figure 35. Description of the successful pod. 32
Figure 36. Description of the unsuccessful pod. 33
Figure 37: Deployment of the SEE-Interface itself in the OneLab Kubernetes. 34
Figure 38: Deployment YAML for a Unikernel based webservice 35
Figure 39: Successfully deployed Unikernel -based webservice. 35
Figure 40: The PPEF architecture. 37
Figure 41: Intent Expectations. 38
Figure 42: PPEF PAC internal modules. 39
Figure 43: Test NGINX server 43
Figure 44: Test NGINX Ingress description. 43
Figure 45: NGINX Kong Service 43
Figure 46: NGINX Kong route 44
Figure 47: NGINX Prometheus plugin 44
Figure 48: Prometheus plugin details 44
Figure 49: NGINX total HTTP request count - initial deployment 45
Figure 50: NGINX total HTTP request count – refresh 45
Figure 51: The bandwidth change rate for NGINX 45
Figure 52: NGINX server latency histograms 45
Figure 53 NEMO D3.1 focus on the detection at runtime (step 7 Gartner DevSecOps) 46

Figure 54 NEMO D3.2 Focuses on Software Composition Analysis (Step 3 Gartner DevSecOps) and Software Signing (Step 5) 46
Figure 55 NEMO D3.3 focus on signature verification at runtime 47
Figure 56 A Guide to Kubernetes Admission Controllers. NEMO focuses on validation admission. 47
Figure 57 OCI Image Verification at Runtime: signature is valid, and policy let the OCI container image be pulled 48
Figure 58 OCI Image Verification at Runtime: signature is not valid, and policies says not to pull OCI container image (strict policy) 49
Figure 59 OCI Image Verification at Runtime: signature is not valid, but policy says pull the OCI container image but warn user (warn policy) 49
Figure 60: meta-Orchestrator Subcomponents 52
Figure 61: MO API Architecture. 53
Figure 62: MO API endpoints. 54
Figure 63: Retrieve spoke clusters from the NEMO Cluster Network endpoint. 55
Figure 64: Payload for joinManagedCluster endpoint. 55
Figure 65: Update replicas endpoint payload for triggering Horizontal Scaling 56
Figure 66:Payload to deploy SEE resources. 57
Figure 67: MO Agent sequence diagram 58
Figure 68: DC Sequence Diagram. 59
Figure 69: Gitlab Tagging. 60
Figure 70: Gitlab CI. 61
Figure 71: NEMO DockerHub. 61
Figure 72: Deployment Manifest. 62
Figure 73: Deployment Details. 62
Figure 74: The schema of FOTA system. 67
Figure 75: NEMO FOTA System Sequence Diagram. 68
Figure 76: Enable Intel TDX in Host OS 76
Figure 77: System BIOS settings. 77
Figure 78: Verify Intel TDX is Enabled on Host OS 77
Figure 79: Operator deployment. 77
Figure 80: wait for "Running" status 77
Figure 81: CC Runtime deployment 77
Figure 82: wait for "Running status" 77
Figure 83: Kubectl get runtimeclass 78
Figure 84: CoCo deployment 78
Figure 85: Image encryption. 79
Figure 86: Upload image. 79
Figure 87: Image Signing. 79
Figure 88: Security policy 80
Figure 89: Register security policy in KBS storage. 80
Figure 90: Deploying encrypted images. 80
Figure 91: Resolving PCCS fail. 81
Figure 92: PCCS config file. 83

List of Acronyms

Abbreviation / acronym	Description
3GPP	3rd Generation Partnership Project
AIoT	Artificiail Intelligence of Things
AMQP	Advanced Message Queuing Protocol
API	Application Programmin Interface
AS	Attestation Service
CFDRL	Cybersecure Federated Deep Reinforcement Learning
CICD	Continuous Integration and Continuous Delivery
CNNAP	Cloud-Native Application Protection Platforms
CO2	Carbon Dioxide
CoCo	Confidential Containers
DC	Deployment Controlller
DDoS	Distributed Denial of Service
DevSecOps	Development, Security and Operations
DFDs	Data Flow Diagrams
DiD	Defense in Depth
DIDs	Decentralized Identifiers
DoS	Denial of Service
Dx.y	Deliverable number y belonging to WP x
EC	European Commission
ECC	Elliptic Curve Cryptography
EDA	Event-Driven Architecture
EDR	End Detection and Response
FOTA	Firmware Over-The-Air
HSMs	Hardware Security Modules
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IBMC	Intent-Based Migration Controller
ID	Identification
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
KBS	Key Broker Service
KPI	Key Performance Indicator
LCM	Lifecycle Manager component
LTE	Long-Term Evolution
MFA	Multi-factor Authentication
mNCC	meta-Network Cluster Controller
MO	meta-Orchestrator
MOCA	Monetization and Consensus-based Accountability
MS	Milestone

NIS2	Network and Information Security Directive 2
OCI	Open Container Initiative
OCM	Open Cluster Management
PAC	Policy Agent Controller
PCCS	Provisioning Certificate Caching Service
PMUs	Phasor Measurement Units
PoLp	Principle of Least Privilege
PPEF	Privacy & Policy Enforcement Framework
PRESS	Privacy, data pRotection, Ethics, Security & Societal
QEMU	Quic Emulator
QGSD	Quaote Generation Service Daemom
RPC	Remote Procedure Call
RVPS	References Value Provider Service
SBOM	Software Bill of Materials
SDLC	Software Development Life Cycle
SEAM	Secure Arbitration Mode
SEE	Secure Execution Environment
SIM	Subscriber Identity Module
SLA	Service Level Agreement
SLO	Service Level Objective
SLSA	Supply-chain Levels of Software Artifacts
SSDLC	Secure Systems Development Lifecycle
TDs	Trusted Domains
TDX	Trusted Domain Extension
TEE	Trusted Execution Environment
TRL	Technology Readiness Levels
UUID	Universal Unique Identifier
VCs	Verifiable Credentials
WP	Work Package
YAML	YAML Ain't Markup Language

Executive Summary

This document provides the final advancements and integration results within Work Package 3 (WP3) of the NEMO project, focusing on a functional, secure, and scalable multi-cluster management system.

D3.3 aligns with the stages of the Software Development Life Cycle (SDLC), *D3.1: Introducing NEMO Kernel* [1] is related to planning, analysis, and design, *D3.2: NEMO Kernel Initial Version* development and some integrations, and finally [2], *D3.3: NEMO Kernel Final Version* refactoring and improving developments, adding functionalities, and closing integrations.

To fully understand this last stage, it is important to have read the D3.1 and D3.2 previously to align with the last milestone that affects WP3, the MS9: NEMO Components (Version 1.0), where the D2.3 [3] and D3.3 are the proofs of this final version.

The main achievements in this deliverable are listed below:

- A demonstration of the Secure Execution Environment (SEE) component in the end stage, providing a highly secure context where resources can be used and deployed. These advancements ensure strong isolation and integrity inside NEMO.
- The PPEF (Privacy & Policy Enforcement Framework) component has been fully integrated and has broadened the NEMO infrastructure, retrieving the platform's monitoring and observability roles; this last point is related to SLAs and SLOs.
- The Cybersecurity and Digital Identity Attestation component solidifies the deployment of cybersecurity measures, including robust authentication and access finalizing with CNNAP, providing an extra step of protection from development until deployment and runtime.
- The meta-Orchestrator (MO) is a functional component for managing resources across IoT, Edge, and Cloud environments. It is designed for scalability using ML feedback from the Cybersecure Federated Deep Reinforcement Learning (CFDRL) component and efficiently the resource management of the NEMO platform.
- To achieve a final integration of each component with the NEMO platform and its components.

D3.3 concludes WP3 by showcasing/demonstrating/presenting a solid and compact NEMO Kernel Space that now offers a smart and easy multi-cluster control, paving the way for future developments. This final deliverable highlights the definition of new functionalities from each component and their integration within the NEMO ecosystem, leading/resulting in a mature and advanced platform.

1 Introduction

Deliverable D3.3 concludes the reporting of the work conducted within the scope of WP3, consolidating and validating important steps taken since the release of D3.2, whose objective aims to demonstrate the latest updates and explain how each component contributes to NEMO in various aspects.

1.1 Purpose of the document

D3.3 describes the last features, tools, and integrations developed to achieve a secure, efficient, and integrated system within the NEMO.

The deliverable sets the final stage for gathering all the development efforts and validating the components involving much more the integration and the testing parts to ensure a robust and mature version of the NEMO Kernel, to contribute to the project's goals of advancing cloud-native computing.

Moreover, this document will validate all the job-done evaluations of the KPIs related to each of the four WP3 components.

1.2 Relation to other project work

This last D3.3: NEMO Kernel Final Version represents an incremental iteration to D3.1 and D3.2 within the WP3. In addition, it is tightly connected with WP2. Particularly, D3.3 shares the main goals, structure and milestone (MS9) objectives within the project with D2.3 Enhancing NEMO Underlying Technology.

Integrations and WP3 components functionalities are related to WP4, and the results demonstrated in D4.3 Advanced NEMO platform & laboratory testing results [5] at MS10 NEMO Integrated (Version 1.0). The work reported in D3.3 also relates to WP4 and the integration activities.

1.3 Structure of the document

This document is structured in a modular format, allowing the reader to see the progress and contributions from each task within WP3. It starts with an executive summary and introduction that describes the purpose and context. The following first four sections are related to the WP tasks:

Section 2 describes the architecture and evaluation of the Secure Execution Environment; Section 3 covers the Privacy and Policy Enforcement Framework, which is the main monitoring part of the project. Section 4 includes Cybersecurity and digital Identity Attestation, security measures, and subcomponents related to NEMO's safety and security. Section 5 introduces the meta-Orchestrator, its subcomponents, and architecture, going into deeper detail with the integration logic.

Section 6 extends this document to the far edge and deals with secure firmware updates. Section 7 explains the validation of each component with key performance indicators and finalizes the general conclusion and technical annexes.

2 Micro-services Secure Execution Environment

Modern cloud environments, characterized by increased scale and complexity, face significant cybersecurity challenges, making it crucial to enhance security and isolation. The rapid growth of cloud services has led to a proliferation of vulnerabilities, underscoring the need for robust data protection and application integrity measures that safeguard user privacy.

The first component of the NEMO Kernel, the Micro-services Secure Execution Environment (SEE) tackles precisely these challenges and provides a set of enhancements for cloud infrastructures regarding isolation, integrity, and flexibility.

2.1 Overview

Currently, the industry standard for cloud service execution is Kubernetes[1], which was originally developed by Google and in 2024 became open-source, which is the de facto standard for container-based cloud infrastructures. It can orchestrate thousands to millions of containers in a cluster and manages relevant aspects like networks or file storage as well.

For these reasons Kubernetes was selected as the backend infrastructure to be used for NEMO, with it orchestrating the services and components at the lowest level. It is, therefore, of uttermost importance to provide the necessary adaptation and integration points, so that other components, such as the meta-Orchestrator, can utilize the computing infrastructure.

Kubernetes is designed to provide isolation using containerization, an approach with little overhead and high flexibility. However, this approach is not sufficient for privacy focused workloads or the execution of highly sensitive services. Also, it is developed for spatially homogenous datacenters, which is an assumption that is not necessarily true in NEMO anymore.

For these reasons, the SEE was developed – a collection of enhancements for Kubernetes to provide the necessary infrastructure for the meta-OS. The SEE on the one hand provides an interface between the computing infrastructure and the high-level services, but also provides enhancements for Kubernetes, that are focused on enhancing the security and integrity of the workloads.

2.2 Architecture and Approach

Several limitations have been identified with the current capabilities of Kubernetes and the SEE has been built as a collection of extensions for Kubernetes. Luckily, Kubernetes is designed in an extensible way, providing open APIs that are built upon for achieving higher isolation and security for the NEMO services. This resulted in the SEE architecture, Figure 1, which consists of the following modules:

- The Unikernel runtime for Kubernetes registers as a new runtime in a cluster and provides the capability to execute highly isolated applications-specific virtual machines in a cluster, leveraging the existing orchestration functionalities.
- The migration extension utilizes the Kubernetes API for fine-grained Pod & Deployment migration. The extension itself is running as a service in the cluster itself, which is a common pattern for infrastructure related services.
- The SEE interface serves as a connector between other NEMO components and Kubernetes as well as the previously mentioned components. The default interface within NEMO components is AMQP v0.9.1 so this component provides an interface layer between the Kubernetes API, our other components and any other potential NEMO component, but is specifically meant to interface with the meta-Orchestrator.

[1] Kubernetes: https://kubernetes.io/docs/home/

- To reduce the trust requirements for the underlying infrastructure, as well as for enhanced integrity, we have evaluated the new area of confidential computing and have developed guidelines and best practices for utilizing confidential computing in NEMO clusters.

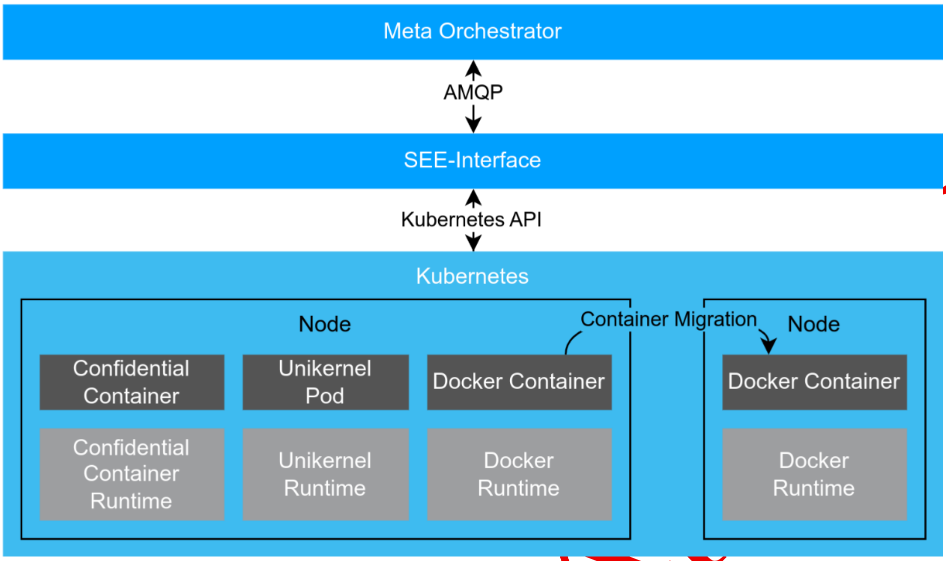


Figure 1: Architecture of the Secure Execution Environment extension collection for Kubernetes.

2.2.1 Unikernel Runtime for Kubernetes

Kubernetes, as well as the most well-known container engine Docker, does not start containers themselves. Instead, Figure 2, Docker is a high-level interface to build container images for forwarding requests to the container manager. In the case of Kubernetes, every node is running a daemon as node manager. That daemon provides high-level tasks like health monitoring but also forwards container spawning requests to the container manager.

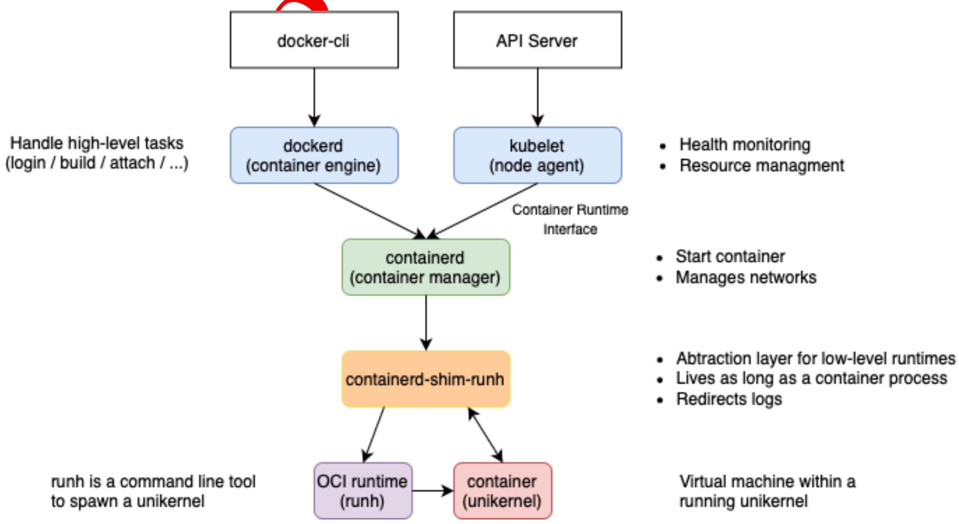


Figure 2: Unikernel integration in Kubernetes.

Multiple container managers such as Podman[2] and cri-o[3] exist, however as OneLab is using *containerd* as container manager, the implementation of the SEE focusses on this one. Containerd is an open-source project, which is more-or-less a spinoff of the original Docker project and designed to fulfil the requirements of Docker and to the standards of the Open Container Initiative (OCI)[4]. While containerd evolved as result of Docker redesign, cri-o has its roots in the Kubernetes community and is focused to solve the community requirements. Podman is a RedHat project and is realized a library and is not depending on a daemon running in the background. containerd manages the network interfaces and uses an OCI-compliant container spawner to create and to start a container. Unlike cri-o, containerd uses an additional abstraction layer between container spawner and container manager called *container runtime shim*.

To use the unikernel *Hermit* as a container replacement with strong isolation and small overhead, the container spawner runh[5] was extended for containerd and a new container runtime shim[6] was developed for the NEMO project. Several base images were also written, that include all necessary tools to start a unikernel. Namely these are the hypervisor QEMU[7] and the daemon virtiofsd[8] to provide local file system access. A user has only to extend these base images with their application to build a suitable image, as is shown in Figure 3. In that Figure, a simple webserver "httpd" is provided as unikernel image, as well as the loader hermit-loader to start the unikernel in the VM. This image uses the Alpine-based base image *hermit_env_alpine*. Alpine Linux is a security-oriented, lightweight Linux distribution. By using Alpine as Linux distribution, the image size is clearly smaller (~21 MiB) in contrast to Ubuntu distribution (~81 MiB).

```
FROM ghcr.io/hermit-os/hermit_env_alpine:latest
COPY hermit-loader-x86_64 hermit/hermit-loader
COPY httpd hermit/httpd
CMD ["/hermit/httpd"]
```

Figure 3: Example of a Dockerfile to build a container image of a simple webserver as a Hermit unikernel

The container spawner runh interprets the command line of the container image and checks if the command is starting a unikernel. If so, the command will be executed within a virtual machine, otherwise, the command will be started as a common Linux container.

The container spawner runh must be registered to containerd. Per default, containerd is using the spawner runc, which is designed to spawn Linux container. The following lines extend the configuration file /etc/containerd/config.toml to support runh, see Figure 4.

2 https://github.com/containers/podman
3 https://github.com/cri-o/cri-o
4 https://opencontainers.org/
5 https://github.com/hermit-os/runh
6 https://github.com/hermit-os/containerd-runh-shim
7 https://www.qemu.org/
8 https://gitlab.com/virtio-fs/virtiofsd

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runh]
    base_runtime_spec = ""
    container_annotations = []
    privileged_without_host_devices = true
    runtime_path = ""
    runtime_root = ""
    runtime_type = "io.containerd.runh.v2"
    pod_annotations = ["io.hermitcontainers.*"]

    [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runh.options]
```

Figure 4: Unikernel runtime configuration for runh

After this configuration, containerd can use runh besides the default spawner runc, but Kubernetes still must be informed about this change. This can be done via the runtime selection mechanism, which is based on the resources RuntimeClass[9]. To announce the spawner runh, the resource can be registered by applying the following file with the tool kubectl, see Figure 5:

```
[apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: runh
handler: runh
```

Figure 5: Registration YAML of the container runtime runh with Kubernetes.

After the registration of the new runtime class, Kubernetes will still use the default spawner runc and by adding the runtime class to the deployment the new spawner runh will be used. The following specification defines a deployment, which contains a simple webserver. The webserver is listening on port 9975 and the container image is publicly available at GitHub repository ghcr.io/hermit-os/httpd:latest. The annotation runtimeClassName: runh shows Kubernetes should use runh to spawn the container. To expose the deployment, a service must be registered. In this example, see Figure 6, the service acts also as load balancer and forward the request to deployment.

[9] https://kubernetes.io/docs/concepts/containers/runtime-class/

```yaml
kind: Service
apiVersion: v1
metadata:
  name: hermit-httpd-service
  namespace: hermit
spec:
  type: LoadBalancer
  ports:
    - name: hermit-httpd
      port: 9975
      targetPort: 9975
  selector:
    app: hermit-httpd-app
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hermit-httpd-app
  namespace: hermit
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hermit-httpd-app
  template:
    metadata:
      labels:
        app: hermit-httpd-app
    spec:
      runtimeClassName: runh
      containers:
      - name: hermit-httpd
        image: ghcr.io/hermit-os/httpd:latest
        ports:
        - containerPort: 9975
```

Figure 6: Deployment YAML of an example Unikernel service

After starting the service, the running processes are seeable on the Kubernetes cluster. Figure 7 shows two running container shims. One is the shim for the spawner runc (process id 3205162), which spawned a NodeJS webserver in common Linux container, while the other shim (process id 2059994) spawned a unikernel, which is running within the hypervisor QEMU.

```
root     3205162  0.0  0.0 1238228 15572 ?       Sl   Mar13   5:59 /usr/bin/containerd-shim-runc-v2 -namespace k8s.io -id 9b49d99b03350a26d
2938ea04fed -add
65535    3205183  0.0  0.0    996   640 ?       Ss   Mar13   0:00  \_ /pause
2102     3205294  0.0  0.0 116572 25516 ?       Ssl  Mar13   6:14  \_ ?
root     3205420  0.0  0.1 271728 60180 ?       Ssl  Mar13   2:50  \_ node webserver
root     3059994  0.0  0.0 958512 11220 ?       Sl   08:54   0:01 /home/stefan/containerd-runh-shim/target/release/containerd-shim-runh-v2
27ea7e194d2ecb23
root     3060016  0.0  0.0    996   640 ?       Ss   08:54   0:00  \_ /pause
root     3060048  1.0  0.3 2357904 100496 ?     Ssl  08:54   0:52  \_ qemu-system-x86_64 -display none -smp 1 -m 1G -serial stdio -device
04 -kernel /herm
root     3060062  0.0  0.0 142304  4224 ?       Sl   08:54   0:00      \_ virtiofsd --socket-path=/run/vhostqemu --shared-dir /root --sand
-handles=never
stefan   3065601  1.2  0.0  22864 13824 ?       Ss   08:58   0:59 /usr/lib/systemd/systemd --user
stefan   3065602  0.0  0.0  21148  3520 ?       S    08:58   0:00  \_ (sd-pam)
```

Figure 7: List of running pods in a Kubernetes Cluster. In the middle it can be seen that a Unikernel is running as a Pod.

2.2.2 Secure Pod Attestation and Deployment Leveraging Trusted Execution Environments

Confidential Containers (CoCo[10]) is a new way to run containers in a secure environment that protects both data and applications, even from the infrastructure provider.

To achieve usage protection, workloads (pods deployed in Kubernetes) are isolated via CoCo, so that neither the cluster nor infrastructure admins can access or manipulate the workloads and the data within, providing data in use protection. Moreover, it also integrates with advanced security hardware features like TEEs (Trusted Execution Environment), allowing it to run sensitive applications in an isolated environment.

CoCo uses Kata Containers[11] runtimes as runtime, leveraging hardware capabilities to add an extra layer of encryption and attestation, where attestation is one of the main components of CoCo. Before deploying a workload as a confidential container, attestation is used as a method to ensure that the TEE in which the container is to be deployed in a secure and trusted environment.

We are using a TDX[12] server as TEE. TDX allows us to create TDs (Trusted Domains), which are virtual and protected hypervisor environments. Trusted domains are used to isolate resources and workloads, allowing only trusted components to access them. This ensures the confidentiality and integrity of data even if there are intruders in the system.

On our server we have used an Ubuntu version 24.04 and Kubernetes version 1.29.9. To enable TDX, the Intel guide[13] has been used.

In order to install CoCo, we have followed the instructions outlined in the quickstart guide[14], installing version 11 of the Operator and CC runtime. After completing the CoCo installation, it was necessary to set up Trustee[15]. To do this, we have used cluster mode, which deploys the services as Docker containers.

To finish the installation, it is necessary to modify the kernel_params in the file */opt/kata/share/defaults/kata-containers/configuration-qemu-tdx.toml* to point to the IP of the KBS container where we have deployed the cluster:

```
kernel_params = "agent.aa_kbc_params=cc_kbc::<KBS_URI>:8080"
```

Figure 8: Kernel Parameters for KBS Integration.

Regarding the general architecture of CoCo, we have two main elements: the TEE and the attestation module.

---

[10] CoCo: https://confidentialcontainers.org/
[11] Kata Containers: https://katacontainers.io/
[12] Intel TDX: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html
[13] https://github.com/canonical/tdx
[14] https://github.com/confidential-containers/confidential-containers/blob/main/quickstart.md
[15] https://github.com/confidential-containers/trustee

The TEE is where the pod is deployed and on which the attestation module collects the necessary hardware measurements to verify that the environment is reliable. The attestation module consists of numerous services that connect to each other to verify the TEE to deploy the container on it.

In CoCo, the Trustee project provides attestation capability and key management engine. In addition, this project allows us to encrypt and sign the container image to be deployed, so that only the trusted environment can decrypt it.

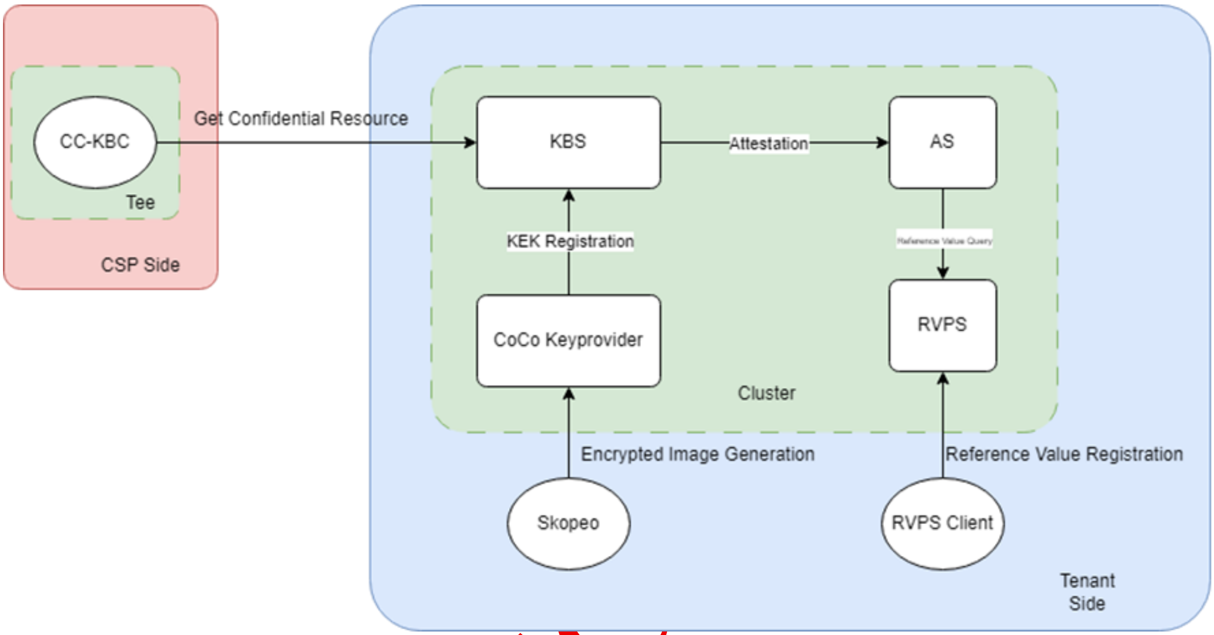


Figure 9. KBS Architecture

Figure 9 represents the general architecture deployed for the CoCo testing environment. The green squares are those provided by Trustee for the attestation process. The other main components are:

- TEE: Environment in which pods are deployed using confidential containers.
- Skopeo: Tool used to encrypt the image of the container. When an image is being encrypted, an attestation agent provides the secrets to skopeo, skopeo sends those secrets to the Keyprovider and finally, it registers them in the KBS[16].
- CoCo Keyprovider: It is responsible for providing the secrets to the KBS. Each time an image is encrypted, the private key is stored inside the KBS container.
- KBS (Key Broker Service): Service that communicates with the TEE and with the Attestation Service. If the Attestation Service confirms to the KBS that the environment is trusted, it is responsible for providing the private keys to the TEE to deploy the pod.
- RVPS Client: Tool to send the reference values to the RVPS. These reference values must be values that are known by the client, because the attestation service will perform the attestation based on the reference values that are injected into the RVPS. If no reference values are inserted into the RVPS, the attestation process will not have against which to compare the evidence provided by the enclave, and it will use a predefined reference value.
- RVPS (Reference Value Provider Service): Manages the reference values to verify the TEE evidence. Those reference values are sent to the AS to compare them with the evidence.

[16] https://github.com/confidential-containers/trustee/blob/main/kbs/docs/cluster.md

- AS (Attestation Service): Performs the attestation process. Responsible for collecting evidence and confirming that it is correct. The Attestation Service uses PCCS (Provisioning Certificate Caching Service) and QGSD (Quote Generation Service Daemon) to generate quotes describing the enclave status and validate the environment.

2.2.2.1 State of the art

Increasing usage of microservices-based architectures and cloud-native environments has created the need to build solutions to provide confidentiality of data in processing. For this purpose, different solutions have been used, such as the TEE technology that facilitates the execution of applications in trusted enclaves safely. In this situation, CoCo presents an innovative approach for confidentiality at the container level, making it more suitable for cloud environments.

CoCo is designed to integrate with cloud environments such as Kubernetes, thus simplifying the deployment of architecture. It is also vendor-neutral, allowing deployment in multi-cloud, on-premises, or hybrid environments. Unlike other alternatives such as Microsoft Azure Confidential Computing[17] or IBM Hyper Protect Services[18], which are dependent on certain vendors[19]. This therefore makes CoCo a technology that addresses both flexibility and security issues.

For the NEMO project, CoCo has proven to be one of the most robust tools to address confidentiality issues without neglecting the importance of scalability and integration with modern workflows.

Resulting from the investigations and experiences when working confidential computing technologies, we have developed a setup and best-practice guide for CoCo, which can be found in 10.1 Guidelines for TDX and Confidential Containers Technology.

2.2.3 Pod & Deployment Migration

Concept

The Micro-services Secure Execution Environment (SEE) supports fine grained workload migration at runtime across cluster nodes. This is achieved by the SEE Migration component. The Migration component is designed as a migration extension for Kubernetes and abstracts the in-cluster workload migration of SEE micro-services in pod and / or deployment level. Kubernetes is designed around the assumption of homogeneous clusters, such as computing centers. However, with edge and far-edge computing as new paradigms, this assumption does not hold anymore. Clusters could be geographically distributed which makes careful positioning of services necessary, so that applications can benefit from low latencies and high bandwidths.

The Migration component is part of the SEE and integrated as a separate interface (Migration interface) in the SEE interface. The Migration component is a service that accepts migration requests through the SEE interface in order to migrate container workloads between different nodes based on decisions of the meta-Orchestrator (e.g., shift load from node on region A to node on region B).

Implementation of the Migration Service

The migration component is implemented as a separate Kubernetes Service, representing the Migration endpoint, backed by a migration daemon deployed as a separate pod. The Migration daemon is implemented as a Python Flask application that exposes two separate endpoints (known as Flask app routes) that serve requests from the front-end Kubernetes Migration Service endpoint:

[17] Azure Confidential Computing: https://azure.microsoft.com/en-us/solutions/confidential-compute
[18] IBM Cloud Hyper Protect Crypto Services: https://www.ibm.com/products/hyper-protect-crypto
[19] Vendor Lock-In in Confidential Computing: https://medium.com/%40safelishare/building-multi-cloud-confidential-computing-the-danger-of-data-lock-in-cfe14893ddb3

- /health endpoint: An http GET method that simply returns "status: UP" if the migration daemon is up and running
- /migrate endpoint: An http POST method that accepts a migration request (JSON format) and calls the Migration functions that perform the actual migration. The method returns a JSON with a breakdown of the migration duration in milliseconds as follows:
  - Total migration time: The total time elapsed since the Migration Service received a Migration Request until the migration was completed.
  - Eviction time: The time elapsed since the Migration Service received a Migration Request until the workload was evicted from the source node.
  - Boot time: The time elapsed since the Migration Service received a Migration Request unti the workload was up and running in the target node.
  - Downtime: The time elapsed while the workload was not running in any node (e.g see method 3 below).

Migration functions: migration is performed using the Kubernetes API. The migration daemon program imports the Kubernetes Python Client in order to perform calls to the Kubernetes Control Plane. The migration is based on the Node Labeling / NodeSelector functionality of Kubernetes and covers three different cases - implemented as three different functions as shown in the following table:

Migration scenario	Migration component function
Deployment Migration	patch_depl_node_selector
Single POD Migration (new POD name / no downtime)	patch_keeppod_node_selector
Single POD Migration (same POD name / downtime)	patch_pod_node_selector

The reason for the distinction between the second and the third case is that each POD has a unique and immutable name in a Kubernetes cluster. In order to move it from one node to another without downtime it is required to first start the POD in the target node and then evict the POD from the old node. As two POD objects with the same name cannot co-exist in the same Kubernetes namespace these two different options are both implemented in the current version of the migration component.

Upon receiving a migration request the first step is to set a key/value label on the target node using the patch_node method of the k8s CoreV1Api client library. We call this label: target label. Depending on the migration scenario the 3 different functionalities are implemented as follows:

a) **Deployment Migration:** The patch_namespaced_deployment method of the k8s AppsV1Api client library is used in order to directly add the target label in the NodeSelector field of the Deployment Configuration. This step will trigger the Kubernetes Control Plane to migrate the Deployment pods to the required target node in order to satisfy the NodeSelector field. No downtime is involved in this case as Kubernetes terminates the old deployment pods after the new PODs are in "Running" state.

b) **Single POD migration (no downtime):** The daemon reads the running POD configuration using the read_namespaced_pod method of the CoreV1Api k8s client library. It starts a new POD named as <old_pod_name>-migr using the same configuration as the old (still) running POD with the additional NodeSelector field changed to match the target label. The Kubernetes Control Plane schedules the new POD to the desired target node. The migration script watches the new POD state using the Kubernetes client Library watch method. When the new POD state is changed to Running, the daemon evicts the old POD from the old node. As a result, there is no downtime.

c) **Single POD migration (downtime):** The target pod is evicted from the node where it is currently running using the delete_namespaced_pod method of the CoreV1Api k8s client library. The daemon then starts watching for events related to the deleted POD. When the DELETED event for the old POD is published the daemon deploys a new POD (with the same name as the old one) on the target node by adding target label in the Pod NodeSelector field as in the a) and b) cases.

Credentials: The migration daemon performs actions that change the cluster configuration (e.g., PODSs, deployments, nodes). In order for the Kubernetes client Library to successfully make the related method calls the daemon pod is related with the "see-migration" Kubernetes Service Account. This is a service account that is binded with the "cluster-admin" Kubernetes cluster role and grants the required permission to the migration daemon.

SEE Interface: Migration

The SEE Interface accepts and forwards migration requests to the Migration Service. The migration requests are read from YAML files, converted to JSON objects by the SEE interface and sent as HTTP requests to the migration service.

Usage: A migration request can be sent using the see-ctl program as follows in Figure 10 :

```
go run cmd/see-ctl/main.go do migrate -f migration-req.yaml
```

Figure 10: Migration request using the see-ctl program

where the migration-req.yaml is the YAML configuration describing the migration info as follows:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: migration-service
  annotations:
    node: "k8s-worker2" # target node
    deployment: "nginx-deployment" # deployment that we want to migrate
```

Figure 11: Deployment migration request

```yaml
apiVersion: v1
kind: Service
metadata:
  name: migration-service
  annotations:
    node: "k8s-worker1" # target node
    pod: "nginx" # the pod name that we want to migrate
    keep_pod_name: "true" # whether to keep the pod name
```

Figure 12: Single pod migration request

Demonstration

Table 1 and presents a breakdown of the migration time for the 3 different scenarios described above using as example workload a nginx server. The image is already present in the target nodes, i.e. the time for pulling the image from the network is not included. Note the downtime in the 3rd case as a result of first evicting the pod from the source node before creating the new one in the target node - the boot time includes the downtime. The downtime is dominated by the eviction time of the old pod. Also note that the boot time is affected by:

- the workload itself, e.g. for different images the boot times may vary.
- the Kubernetes Control Plane decision overhead.

Scenario	Total	Boot	Eviction	Downtime
Deployment (2 replicas)	7921ms	5694ms	7919ms	0ms
Pod (new name)	5402ms	3688ms	5402ms	0ms
Pod (same name)	6111ms	6111ms	2939ms	3171ms

Table 1: Migration time breakdown for an example micro-service (nginx).

Figure 13, Figure 14 and Figure 15 demonstrate the Migration Component functionality in all 3 cases, using the SEE Migration Interface. Figures also include the migration-pod logs showing the migration times breakdown, which is later encapsulated in the daemon response.

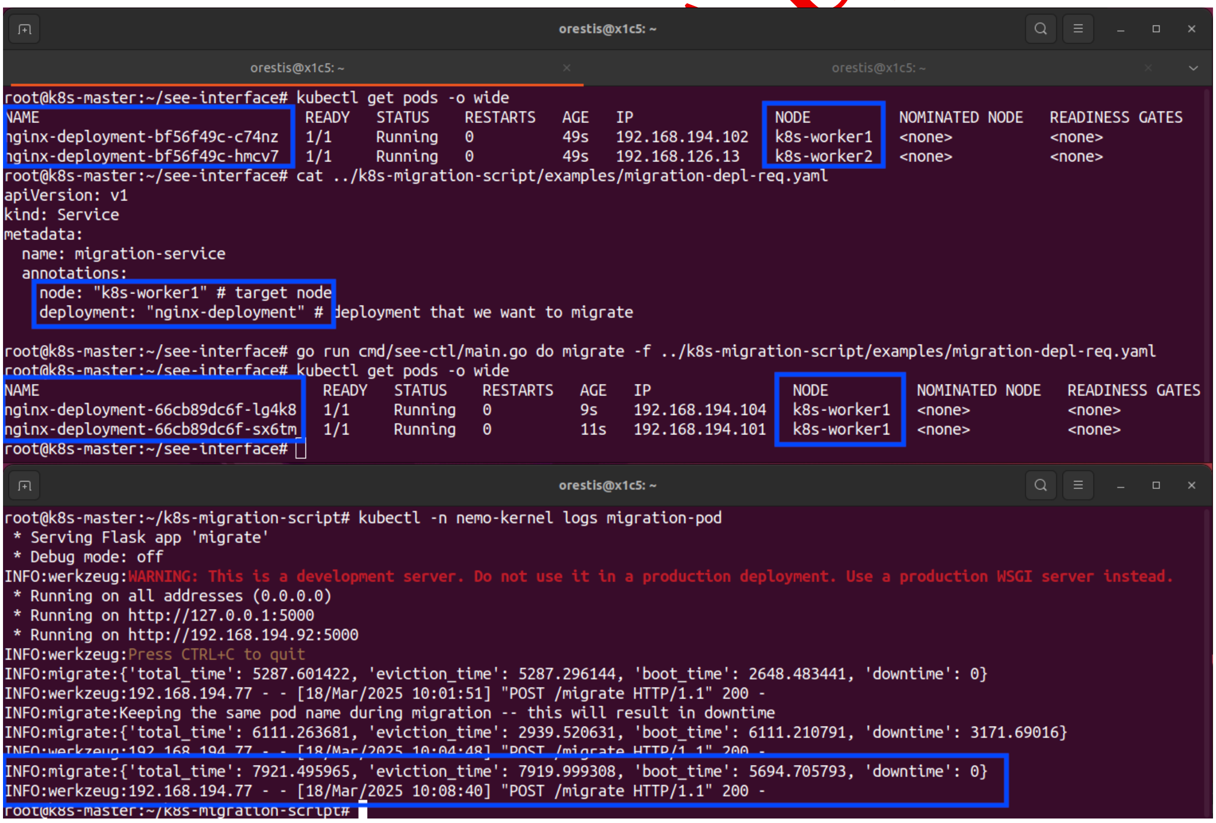


Figure 13: Deployment migration demo.

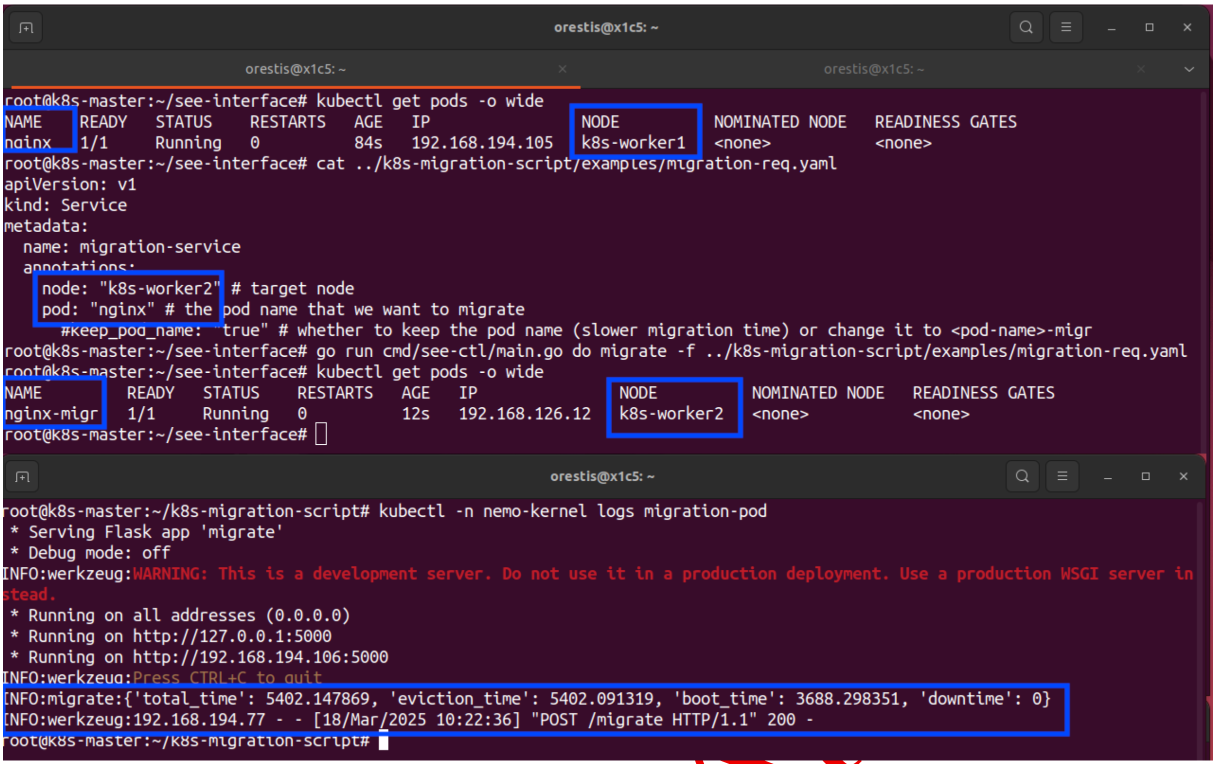


Figure 14: Single pod migration (no downtime) demo.

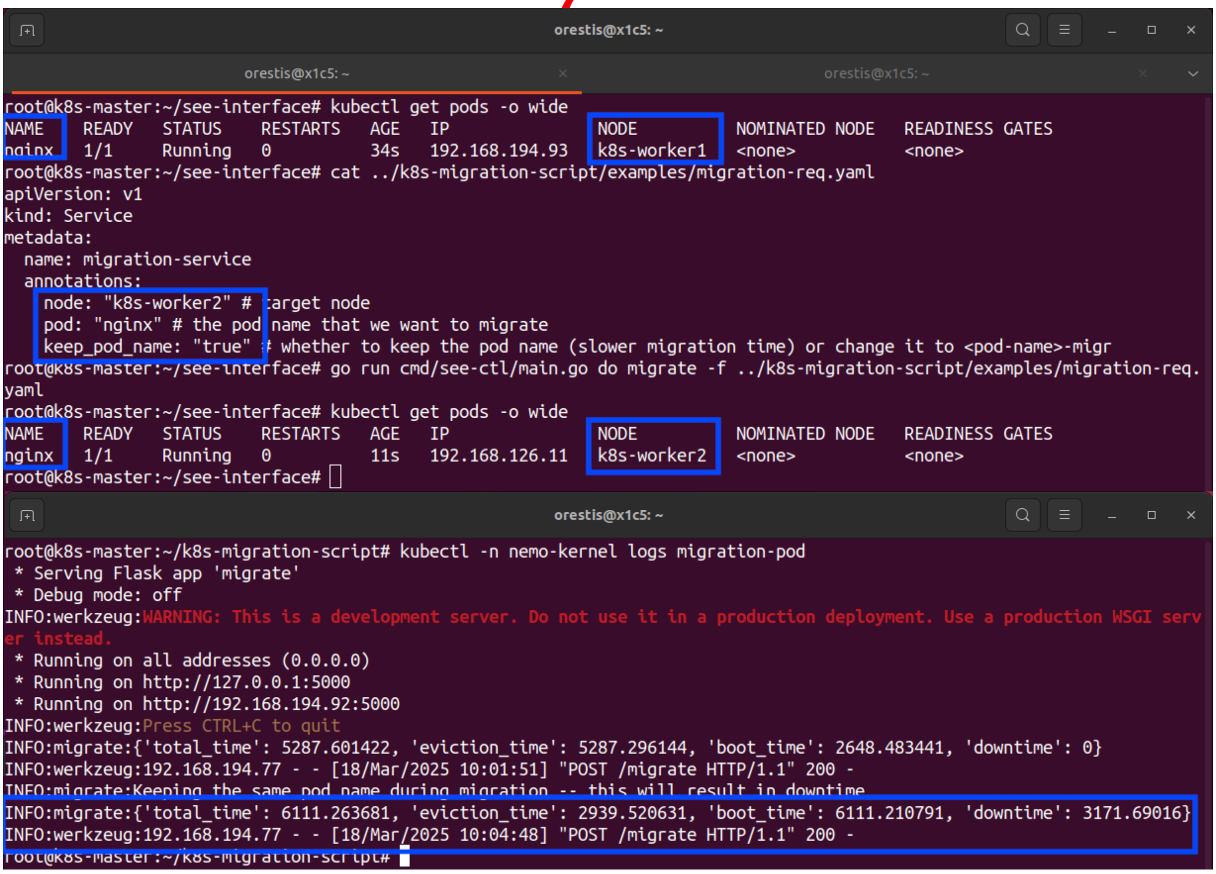


Figure 15: Single pod migration (with downtime) demo.

2.2.4 meta-Orchestrator integration

The SEE interface is the software component for interacting with the SEE via AMQP. As depicted in Figure 1, it is used by other components to interact with the other SEE components. The main points of interaction are retrieving both node and pod metrics as well as performing actions with the SEE, see Figure 16.

Retrieving metrics

The SEE interface retrieves both node and pod metrics from Kubernetes and publishes them to an AMQP exchange as shown in Figure 17.

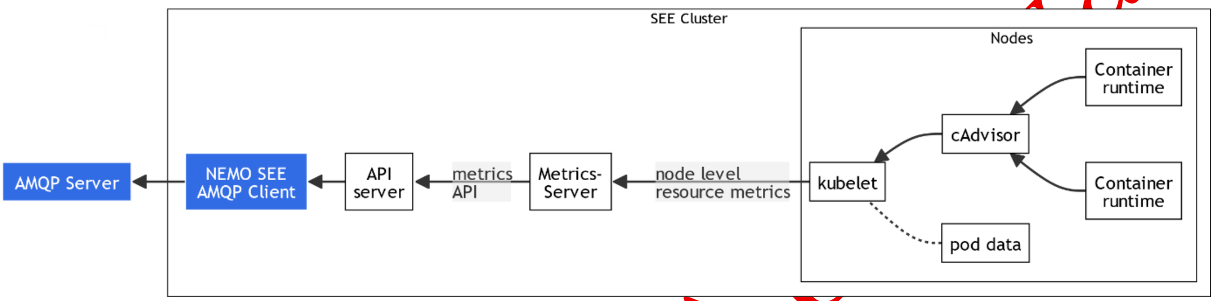


Figure 16: AMQP messages for pod metrics

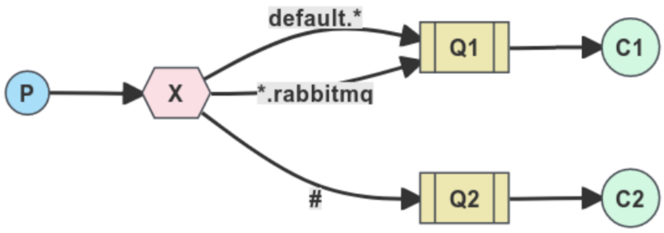
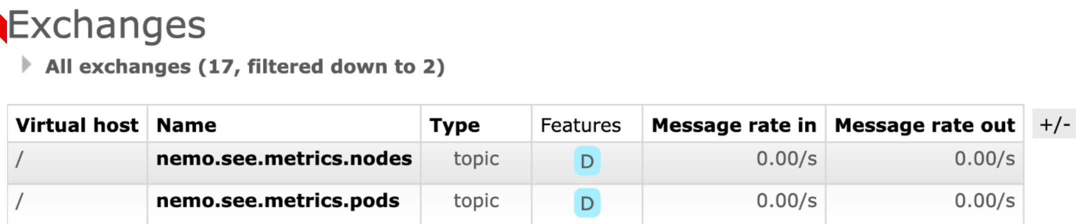


Figure 17: AMQP messages for pod metrics

This part examines how to retrieve node metrics from the cluster as an example. Retrieving pod metrics would work very similarly. We use the RabbitMQ management interface for demonstrating communicating with the SEE interface while software components use appropriate AMQP libraries.

First, we look for the relevant SEE interface exchanges:



Exchanges

All exchanges (17, filtered down to 2)

Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	nemo.see.metrics.nodes	topic	D	0.00/s	0.00/s	
/	nemo.see.metrics.pods	topic	D	0.00/s	0.00/s	

Figure 18: RabbitMQ exchanges for SEE metrics

These exchanges make node and pod metrics available as single messages per node and per pod. This allows consumers to natively select which nodes and pods they are interested in, leveraging the appropriate AMQP primitives for offloading this routing and filtering to the AMQP server.

To receive metrics, we add a new queue called “my.test.queue” to receive arbitrary messages on:

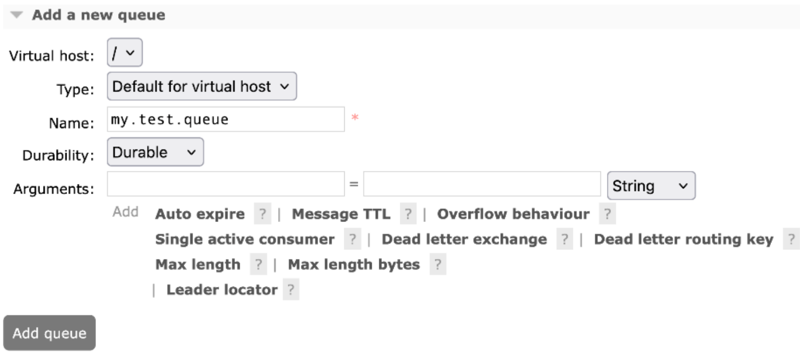


Figure 19: Creation of a new RabbitMQ queue for message retrieval.

For messages to arrive on this queue, we create a binding to forward messages from the exchange to the receiving queue. The routing key specifies which messages we are interested in. For pod metrics, the routing key is “<namespace>.<pod_name>”. If we were interested in the metrics for all pods in the default namespace for example, we would specify “default.*” as the routing key for the binding. For node metrics, the routing key is the node name. In this example, we bind all messages from the node metrics queue to our receiver queue.

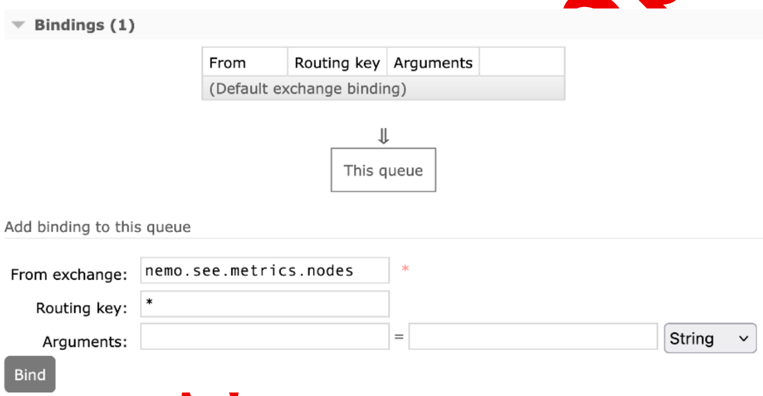


Figure 20: Binding a RabbitMQ message queue to the exchange.

The SEE interface periodically publishes metric messages to the exchanges. Once a new message that matches our binding has been published to the exchange, we can get the message from our receiver queue:

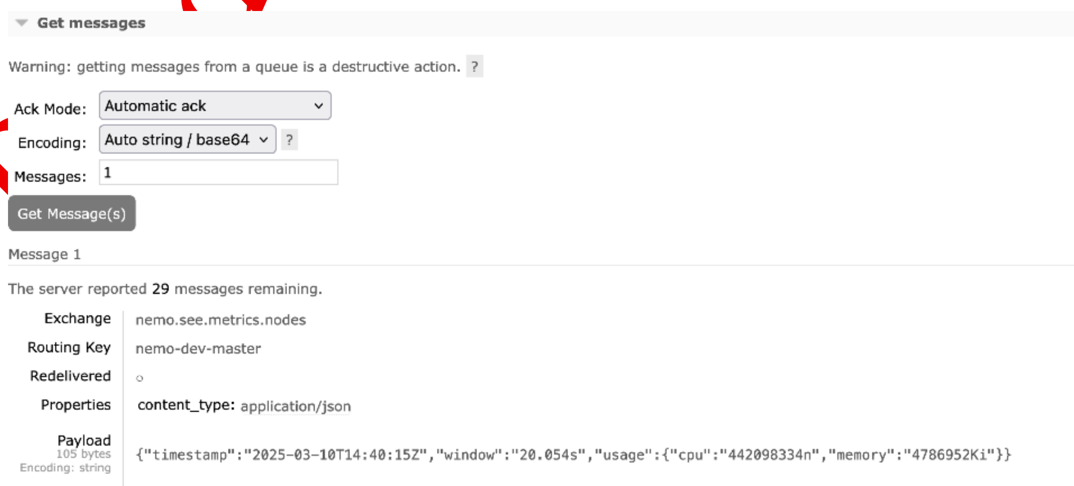


Figure 21: Receiving resource metrics via RabbitMQ

Perform an action in the SEE

The SEE interface can be driven by an AMQP RPC API as shown in Figure 22.

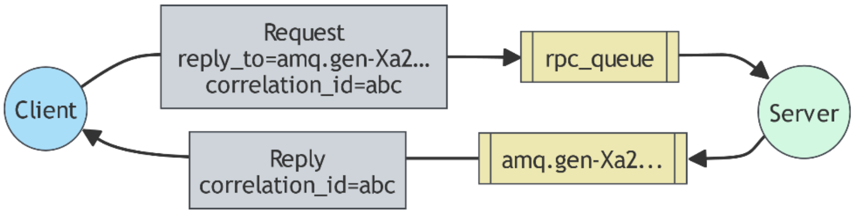
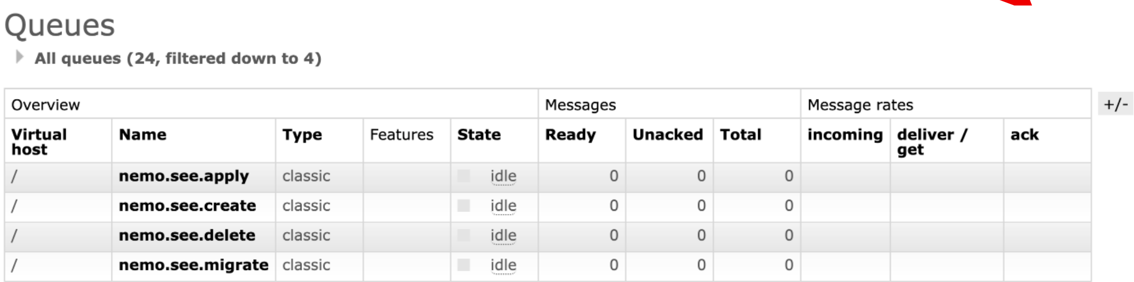


Figure 22: AMQP flow for resource configuration.

Walking through the process using the RabbitMQ management interface similarly to how we have done it for the metrics. First, we look for the relevant RPC queues:



Queues

All queues (24, filtered down to 4)

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	nemo.see.apply	classic		idle	0	0	0			
/	nemo.see.create	classic		idle	0	0	0			
/	nemo.see.delete	classic		idle	0	0	0			
/	nemo.see.migrate	classic		idle	0	0	0			

Figure 23: RabbitMQ queues relevant for resource configuration.

We see four RPC queues, Figure 23: one for creating resources, one for applying changes to resources, one for deleting resources, and one for migrating resources using SEE's migration component. Now, an NGINX pod can be created by publishing a message to the nemo.see.create RPC queue:

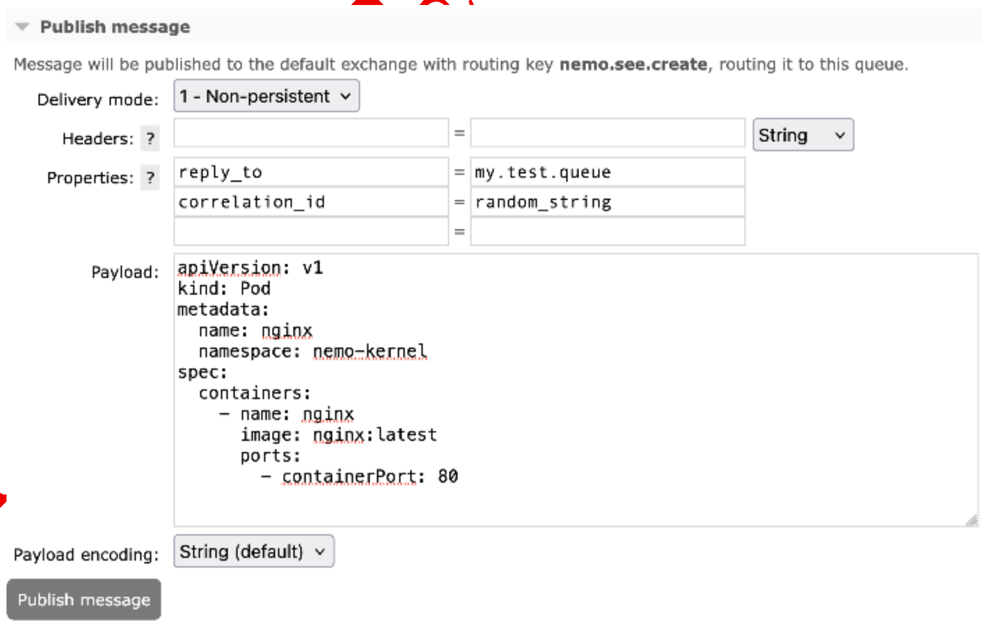


Figure 24: Creation of a NGINX Pod via the SEE interface in the RabbitMQ web-ui

In Figure 24 two properties are very important for RPC messages: the "reply_to" property and the "correlation_id" property. The "reply_to" property tells the SEE interface where to send the response to this RPC message. Its value should be a callback queue which is set up before by the RPC caller. This callback queue should usually be a non-durable queue with an AMQP-server-generated name to avoid collisions for these one-off responses. In our case, we reused the test queue from the previous section strictly for demonstration purposes. The other important property is the correlation ID. This ID is included in the response again so the caller can be sure the response corresponds to their request. The correlation ID can be any sufficiently random string, but we recommend UUIDv7.

Once we have sent the request, SEE interface will perform the corresponding action and send a reply to the referenced callback queue. So, in Figure 25, the response from our test queue is:

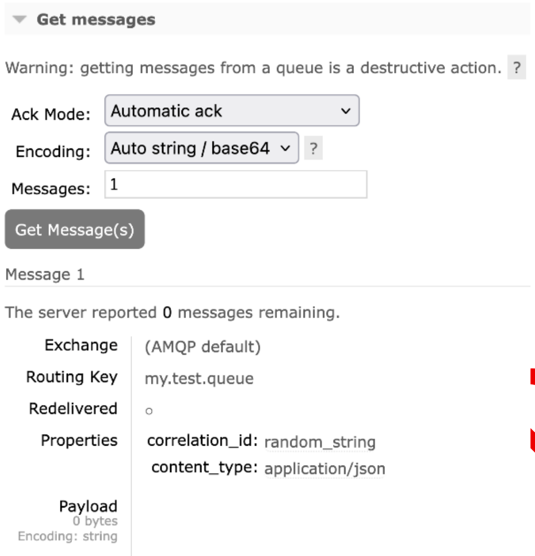


Figure 25: SEE-Interface response of the NGINX pod creation in the RabbitMQ web UI.

In this case the response is empty, which means success. When trying to create the same, now existing, pod again, we receive an error, Figure 26:

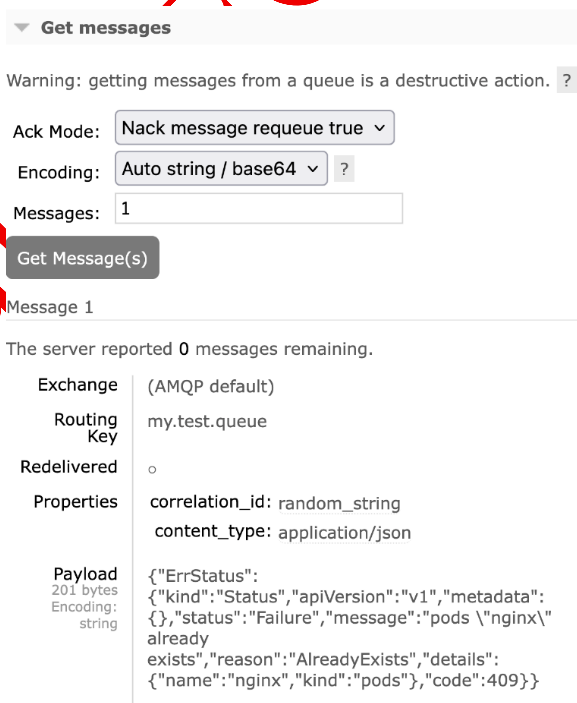


Figure 26: SEE-Interface failure response when deploying a pod via the RabbitMQ web UI.

2.3 Evaluation

2.3.1 Unikernel image size overhead

To investigate the claim of Unikernels having a low overhead compared to containers, we take a look at the image sizes and their composition in the cloud use case.

The Unikernel Hermit that is selected in NEMO is deployed in Kubernetes as a regular layered Docker image. This image contains the Kernel and the Kernel's bootloader, but also a minimal userspace installation and an instance of the VM hypervisor QEMU. It is the latter that can be considered overhead when comparing containers and Unikernels, therefore we try to quantify this. The Hermit project provides two base images with this setup[20], one based on the widespread Ubuntu image (Figure 27) and another one based on the lightweight Alpine Linux image (Figure 28). When investigating the content of these files, we can see that both base images still remain rather small, but the QEMU installation induces an overhead of 142MiB/81MiB.

```
│ ● Layers ├──────────────────────────────────────────────────────────────
Cmp   Size  Command
     78 MB  FROM blobs
     46 MB  RUN /bin/sh -c apt update # buildkit
     96 MB  RUN /bin/sh -c apt install -y --no-install-recommends libcap-ng0 libseccom
    3.7 MB  COPY /usr/local/cargo/bin/virtiofsd /usr/bin/virtiofsd # buildkit
       0 B  RUN /bin/sh -c chmod 0755 /usr/bin/virtiofsd # buildkit
```

Figure 27: Ubuntu based baseimage for Hermit containers.

```
│ ● Layers ├──────────────────────────────────────────────────────────────
Cmp   Size  Command
    7.8 MB  FROM blobs
     77 MB  RUN /bin/sh -c apk add --no-cache qemu qemu-system-x86_64 libseccomp libca
    3.7 MB  COPY /root/.cargo/bin/virtiofsd /usr/bin/virtiofsd # buildkit
       0 B  RUN /bin/sh -c chmod 0755 /usr/bin/virtiofsd # buildkit
```

Figure 28: Alpine Linux based base image for Hermit containers.

However, as Kubernetes is reusing layers of the Dockerfile, this only has to be considered once per host, independently of the amount of Unikernels running on that machine. The variable parts are the loader binary and the actual application. Figure 29 shows a resulting image along the disk usage of each layer. We can see that the Unikernel parts can provide a webserver in less than 5 MiB. As a comparison, the small nginx:alpine image[21] is shown in Figure 30. Skipping QEMU, the image size is smaller overall, but the plain webserver is more than 7 times the size than the one in the unikernel image.

[20] https://github.com/orgs/hermit-os/packages?repo_name=runh

[21] https://hub.docker.com/layers/library/nginx/alpine/images/sha256-799a9c761078cbbd04bdef1f3578741455114a29c55988e697bfceb97fa14682

```
| • Layers |
Cmp   Size  Command
     7.8 MB  FROM blobs
      77 MB  RUN /bin/sh -c apk add --no-cache qemu qemu-system-x86_64 libseccomp libcap-ng # buildkit
     3.7 MB  COPY /root/.cargo/bin/virtiofsd /usr/bin/virtiofsd # buildkit
        0 B  RUN /bin/sh -c chmod 0755 /usr/bin/virtiofsd # buildkit
     155 kB  COPY hermit-loader-x86_64 hermit/hermit-loader # buildkit
     157 kB  COPY hermit-loader-x86_64-fc hermit/hermit-loader-fc # buildkit
     4.5 MB  COPY httpd hermit/httpd # buildkit
```

Figure 29: Image composition of a Hermit container based on the Alpine Linux baseimage.

```
| • Layers |
Cmp   Size  Command
     7.8 MB  FROM 994456c4fd7b2b8
     4.0 MB  RUN /bin/sh -c set -x     && addgroup -g 101 -S nginx     && adduser -S -D -H -u 101 -h /var/cache/ngi
     1.6 kB  COPY docker-entrypoint.sh / # buildkit
     2.1 kB  COPY 10-listen-on-ipv6-by-default.sh /docker-entrypoint.d # buildkit
      389 B  COPY 15-local-resolvers.envsh /docker-entrypoint.d # buildkit
     3.0 kB  COPY 20-envsubst-on-templates.sh /docker-entrypoint.d # buildkit
     4.6 kB  COPY 30-tune-worker-processes.sh /docker-entrypoint.d # buildkit
      35 MB  RUN /bin/sh -c set -x     && apkArch="$(cat /etc/apk/arch)"     && nginxPackages="          nginx=${NGI
```

Figure 30: Image composition of the nginx-alpine container.

This result is indicative, that Unikernels can provide very small application images. The comparison looks different for different servers, and only a small example is shown here. But it is to be expected, that with larger applications, the overhead of bundling QEMU is outweighed by the small image size of the application.

2.3.2 Secure pod attestation

On our server we have used an Ubuntu version 24.04 and Kubernetes version 1.29.9. To enable TDX, the Intel guide has been used and in order to install CoCo, we have followed the instructions outlined in the quickstart guide, installing the version 11 of the Operator and CC runtime. After completing the CoCo installation, it was necessary to set up Trustee. To do this, we have used the cluster mode, which deploys the services as Docker containers[22].

To finish the installation, it is necessary to modify the kernel_params, see Figure 31, in the file */opt/kata/share/defaults/kata-containers/configuration-qemu-tdx.toml* to point to the IP of the KBS container where we have deployed the cluster:

```
kernel_params = "agent.aa_kbc_params=cc_kbc::<KBS_URI>:8080"
```

Figure 31: Kernel Parameters for KBS Integration.

In the following section startup time and cluster resource consumption will be considered for the measures.

2.3.2.1 Startup time and resource consumption

The time it takes to deploy the pod in CoCo is counted from the time the command to start the pod is executed until its status is "Running".

Figure 32 corresponds to the first log found when a pod is deployed.

[22] https://github.com/confidential-containers/guest-components/tree/main/attestation-agent/coco_keyprovider

```
[2024-10-16T11:26:04Z INFO kbs::http::attest] Auth API called.
```

Figure 32: Authentication Request.

This log confirms that the KBS has received a request. This marks the start of an authentication process to verify the identity of the requestor. If the attestation process is successful, the KBS will provide the private key to decrypt the image(s) of the pod container to be deployed. Figure 33 corresponds to the last log.

```
[2024-10-16T11:26:07Z  INFO   actix_web::middleware::logger]  172.18.0.1  "GET
/kbs/v0/resource/default/image-kek/2b5353ec-709a-4254-a311-f8ec8f2bff40
HTTP/1.1" 200 530 "-" "attestation-agent-kbs-client/0.1.0" 0.005777
```

Figure 33: Key Retrieval.

Therefore, the time it takes to confirm that the TEE is trusted and provides the secrets to the host to deploy the pod is around 3 seconds. Although once the host has the private key to decrypt the container image, k8s takes about 5-10 seconds to deploy the pod.

The pods deployed to enable the execution of CoCo consume a total of 150MiB of memory and 3m of CPU, with overall memory and CPU consumption being controlled.

```
confidential-containers-system   cc-operator-controller-manager-699d884f44-w2tsj   3m   28Mi
confidential-containers-system   cc-operator-daemon-install-792nj                  0m   69Mi
confidential-containers-system   cc-operator-pre-install-daemon-86jrg              0m   53Mi
```

Figure 34. CoCo Resource Consumption.

2.3.2.2 Deployment testing

Two different scenarios were successfully demonstrated. These are the details of those scenarios:

Successful pod deployment using CoCo

For this purpose, the image of the container containing the pod to be deployed has been encrypted. These keys have been stored in the KBS correctly using skopeo and CoCo Keyprovider. Therefore, if the attestation process is successful, the KBS will be able to find the private key associated to the public key of the encrypted image and will provide it to the TEE to decrypt the image and deploy the pod. The Figure 35 depicts the different characteristics of the pod deployed using CoCo. Providing us with information about the runtime used (kata-qemu-tdx), which the one used for Intel TDX, the encrypted image used and the state of the pod, among others.

```
mw@pandora-1:~/kbs$ kubectl describe pod encrypted-image-test-busybox
Name:                encrypted-image-test-busybox
Namespace:           default
Priority:            0
Runtime Class Name:  kata-qemu-tdx
Service Account:     default
Node:                pandora-1/192.168.159.209
Start Time:          Tue, 22 Oct 2024 09:24:56 +0000
Labels:              run=encrypted-image-test-busybox
Annotations:         cni.projectcalico.org/containerID: 2fa817e5133e2990997f1a05dd2e9b56a702927e78691f0f7e71a02e4cbf6bc2
                     cni.projectcalico.org/podIP: 172.16.19.118/32
                     cni.projectcalico.org/podIPs: 172.16.19.118/32
                     io.containerd.cri.runtime-handler: kata-qemu-tdx
Status:              Running
IP:                  172.16.19.118
IPs:
  IP:  172.16.19.118
Containers:
  busybox:
    Container ID:   containerd://8bb19fa78dd2c3d6bd037a96b03d1e33a480586c4498d83a8e23d54c8fd0ba0d
    Image:          docker.io/jorgealmansa/busybox:encrypted
    Image ID:       docker.io/jorgealmansa/busybox@sha256:505fc7788dda5e0a20a401dabd72ace72bb762a7d19cf60207b613173f27107f
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Tue, 22 Oct 2024 09:25:08 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-gdxn7 (ro)
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
Volumes:
  kube-api-access-gdxn7:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              katacontainers.io/kata-runtime=true
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:                      <none>
```

Figure 35. Description of the successful pod.

Unsuccessful pod deployment using CoCo

In this case, we encrypt the container image but do not store the secrets in KBS. Therefore, KBS is not able to find the private key and cannot provide it to the TEE to unlock the container image.

On Figure 36, the error message is “failed to create container task: failed to create shim task: failed to handle layer: failed to get decrypt key”. The container is trying to deploy an encrypted image, but it fails because it cannot find the private key to decrypt it. It also gives us information about the runtime being used, the image and image ID, the last state, etc.

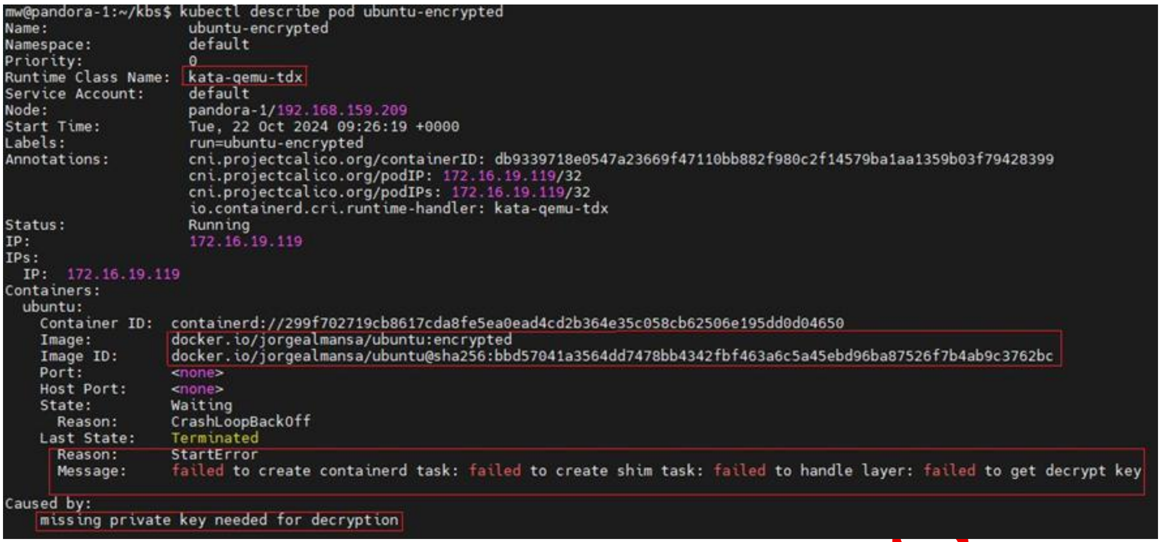


Figure 36. Description of the unsuccessful pod.

2.3.2.3 Custom images

Finally, a pod has been successfully deployed using a custom image, following the same deployment steps as before. In the image, there are several pre-installed packages, enabling the container to function as a SDN network controller or as a switch while also interacting with the Kubernetes API.

During the image encryption process, an issue has been encountered: two encrypted layers corresponded to identical plaintext layer, preventing decryption. This issue is documented in a pull request on the Confidential Containers GitHub repository [1].

Finally, CoCo offers us a good and easy-to-install way to protect workloads in cloud environments, by incorporating attestation mechanisms, key management and encryption technologies. Our implementation leverages TDX to create TDs, offering isolation and protection to make the environment even more reliable. The deployment process has demonstrated scalability, robustness, and efficiency, requiring minimal resources. Moreover, its integration with Kubernetes and support for custom images makes it a very versatile solution. All this makes CoCo a flexible and independent solution for organizations to protect their data without compromising performance or scalability.

2.4 Unikernel Deployment via SEE-Interface

As mentioned previously, the SEE-Interface itself runs as a pod in Kubernetes. Thus, first, there is a need to deploy the prebuild image via the *components.yaml* that is offered in the project's repository[23]. Figure 37 shows the successfully deployed the SEE interface to the OneLab cluster:

[23] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-kernel/secure-execution-environment/see-interface/-/blob/main/components.yaml

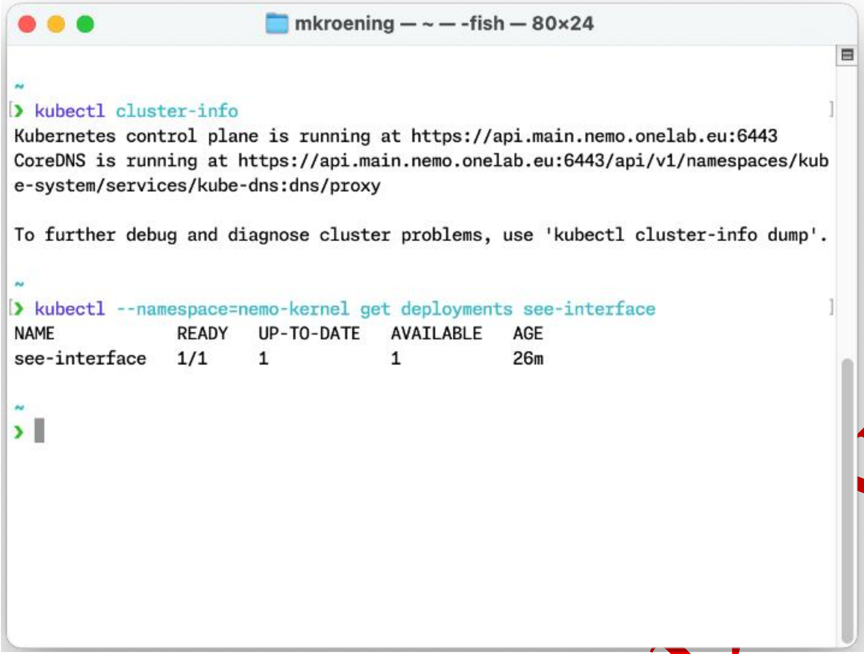


Figure 37: Deployment of the SEE-Interface itself in the OneLab Kubernetes.

To demonstrate the process of deploying a safe unikernel via the SEE-Interface, define the service in a deployment YAML file for Kubernetes, as represented in Figure 38. The service is a simple webserver that is executed as a unikernel. The main difference between a unikernel and a normal container-based pod in the deployment yamls is the runtimeClassName field. It must be set to the Unikernel runtime runh. Additionally, the nodeSelector field is set, as not all nodes in the OneLab test cluster have this runtime installed.

```yaml
kind: Service
apiVersion: v1
metadata:
  name: hermit-httpd-service
  namespace: hermit
spec:
  type: LoadBalancer
  ports:
    - name: http
      port: 9975
      targetPort: 9975
  selector:
    app: hermit-httpd-app
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hermit-httpd-app
  namespace: hermit
spec:
```

```yaml
  replicas: 1
  selector:
    matchLabels:
      app: hermit-httpd-app
  template:
    metadata:
      labels:
        app: hermit-httpd-app
    spec:
      runtimeClassName: runh
      containers:
      - name: hermit-httpd
        image: ghcr.io/hermit-os/httpd:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 9975
      nodeSelector:
        runtime: runh
```


Figure 38: Deployment YAML for a Unikernel based webservice

Finally, the pod is deployed, and the service is available via a browser, see below:

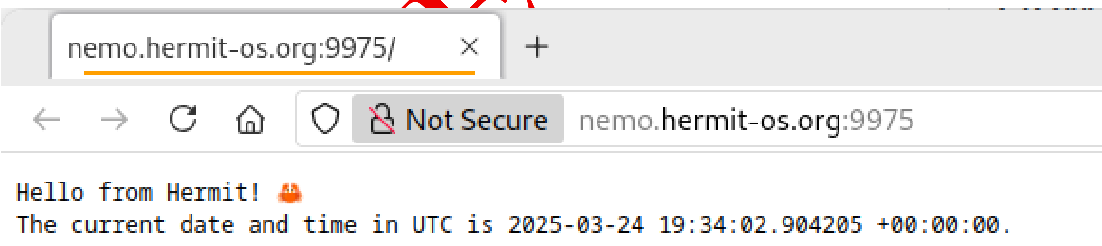


Figure 39: Successfully deployed Unikernel -based webservice.

2.5 Conclusion

The outcomes of task 3.1 provide several promising enhancements for classical cloud infrastructures. These are for once the Unikernel extension for Kubernetes, providing means of deploying highly specialized and well-isolated application images also on cloud scale. The migration extension allows fine-grained pod and deployment migration, which is relevant for locally distributed clusters to provide careful service placement, e.g., depending on latency or the CO2 level in the local power-grid. Last, an investigation was conducted into the use of confidential computing technologies and provided a setup and integration guide for the project and beyond, to allow trustful cloud infrastructures in NEMO. In combination with the underlying Kubernetes, these extensions form the Secure Execution Environment for service execution in the NEMO kernel. All components' development is complete, and they are successfully deployed in the OneLab cluster.

3 Privacy & Policy Enforcement Framework

3.1 Overview

The ultimate objective of NEMO meta-OS concerns the optimal management of hyper-distributed services over AIoT-Edge-Cloud continuum. This requires the appropriate definition of Service Level Objectives that would drive and, at the same time, safeguard the optimal operation of the deployed applications. NEMO meta-OS adopts an intent-based approach that drives the management of the NEMO stakeholders' application requirements and defines their optimal lifecycle management. The Privacy & Policy Enforcement Framework (PPEF) component materializes the NEMO meta-OS intent-based approach supporting the governance of workloads that adhere to high-performance and high-energy efficiency operations as manifested by the NEMO stakeholders.

The PPEF that was introduced in D3.1 and further evolved as described in D3.2 introduces the mechanism that safeguards the compliance and enforcement of different aspects of the application life cycle concerning security, privacy, cost, performance, and environmental impact aspects. D4.2 "*Advanced NEMO platform & laboratory testing results. Initial version*" [4] which was submitted in M27 and documented the first integrated NEMO meta-OS framework already incorporated, as part of the end-to-end integration scenarios that presented functional examples of the PPEF highlighting its role within the NEMO framework. This document's final version of the PPEF is detailed, providing the latest technical and functional updates and new insights. For the sake of completeness, technical information that were already presented in past deliverables might also be included here.

3.2 Architecture and Approach

In D3.2, section 3.2, the PPEF concept is introduced illustrating the high-level architecture and functional aspects of the tool that concerns the NEMO workload policies and the intents' enforcement and management activities. The present document incorporates the final specifications of the component describing the latest updates adopted by the component.

Moreover, in D3.2 a list of SLOs that are incorporated by the NEMO meta-OS are defined. These SLOs concern both the NEMO governed clusters and the NEMO hosted workloads and are defined by the NEMO provider during the registration of a cluster (infrastructure) or of a workload (application). These SLOs cover both static and dynamic information that describe an asset supported by NEMO. This section summarizes the intent/expectation/target list supported by the PPEF component that corresponds to dynamic properties which are monitored by the PPEF in the context of NEMO meta-OS.

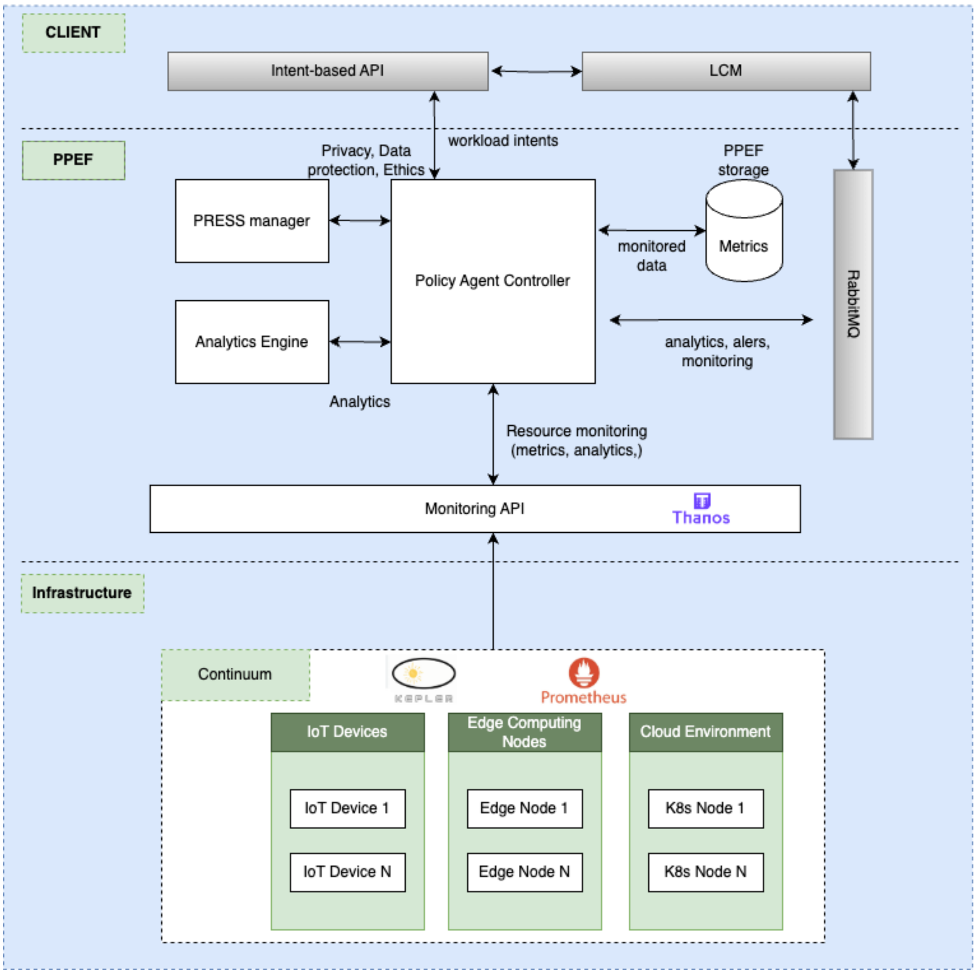


Figure 40: The PPEF architecture.

The PPEF is vertically oriented in the NEMO meta-OS architecture, implying both direct and indirect interfacing and interaction with core NEMO meta-OS components, including the intent-based API, the meta-Orchestrator, the CFDRL, and the Monetization and Consensus-based Accountability (MOCA).

3.3 NEMO workload monitoring

3.3.1 Intents and Expectations

NEMO was inspired by the 3GPP specification[24] #28.312 which covers the intent-driven management of services for mobile networks and has been adapted to suit the project's needs. In principle, an intent specifies the expectations, including requirements, goals and constraints for a specific service or workflow. The intent may provide information on a particular objective and related details. It is typically understandable by humans and needs to be interpreted by the machine without any ambiguity, focusing more on describing the "What" needs to be achieved but less on "How" those outcomes should be achieved, expressing the metrics that need to be achieved.

[24] https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3554

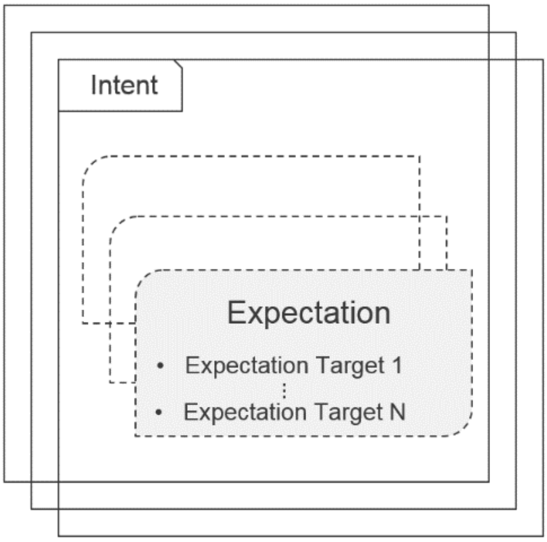


Figure 41: Intent Expectations.

As indicated in Figure 41, an *Intent* consists of a set of *Expectations* (Intent Expectations) which describe the requirements, goals, and context to be achieved. For a given expectation the desired characteristics of the service are the *expectation targets* to be achieved. The expectation targets are associated with metrics that measure the corresponding values.

In view of the final version of the PPEF significant implementation enhancements and corresponding code refactoring was necessary to optimize the operation of the component. In the context of the workload scheduling that is governed by the CFDRL component, workload migration and workload scaling (scale out) actions might be triggered. The latter introduced some added complexity to the PPEF logic that concerns the calculation of the *Computing Workload* monitoring which was addressed in the finalized PPEF.

The PPEF has defined six types of intents which correspond to the desired application behaviour that NEMO service provider assigns in business terms for a NEMO meta-OS hosted application (workload). Specifically, the NEMO meta-OS workload intents are the *Computing Workload Intent, the EnergyCarbonEfficiency Intent, the Security Intent, the FederatedLearning Intent, the Machine Learning Intent and the Network Intent*. The associated expectations that are mapped to the abovementioned intents are listed in tabulated format below in Table 2, Table 3, Table 4, Table 5, Table 6, and Table 7.

Table 2: Computing Workload Intent.

Expectation	*Description*	*Target Value Range*
CPU usage	*Usage in seconds*	*Integer value*
RAM usage	*Bytes in memory occupied*	*Integer value*

Table 3: Energy Carbon Efficiency Intent.

Expectation	*Description*	*Target Value Range*
Energy Consumption rate	*Joules per second (avg in 5')*	*Integer value*
Energy Efficiency	*Joules for every second of CPU time*	*Integer value*
Energy Consumption	*Total Joules consumed*	*Integer value*

Table 4: Security Intent.

Expectation	Description	Target Value Range
Federated Learning	*FL environment requirement*	*Yes/No*

Table 5: Federated Learning Intent.

Expectation	Description	Target Value Range
Security	*SEE requirement*	*Yes/No*

Table 6: Machine Learning Intent.

Expectation	Description	Target Value Range
Machine Learning	*ML environment required*	*Yes/No*
vRAM	*vRAM capacity in GB requirement*	*Integer value*

Table 7: Network Intent.

Expectation	Description	Target Value Range
Secure	*AccessList descriptor*	*IP*
UL Capacity	*Uplink capacity for 5G slice*	*IP, portNumber and portType*
DL Capacity	*Downlink capacity for 5G slice*	*IP, portNumber and portType*

The PPEF is responsible for the intent-based NEMO workload monitoring which is governed by the *Policy Agent Controller (PAC)* module which is the heart and mind of the PPEF environment. The PAC facilitates the management of the monitoring process of the NEMO workloads, which is driven by the intents of the NEMO user. More specifically, the PAC internally realizes two modules which tackle distinct aspects of the NEMO workload's intent monitoring lifecycle, Figure 42.

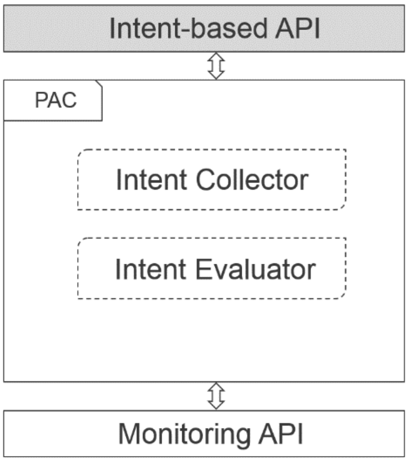


Figure 42: PPEF PAC internal modules.

The *Intent Validator* is a new feature that is introduced for the final version of the PPEF component in NEMO meta-OS. The target value that is assigned by the NEMO meta-OS service provider which corresponds to an *expectation/target* attribute is validated through a *validation filter*. This ensures that the monitoring thresholds correspond to the infrastructure specifications and are aligned with the *Target Value Range* of each intent/expectation.

The *Intent Collector* is responsible for collecting the intents that are associated with a NEMO workload through *the Intent-based API* and thus triggering the monitoring process.

The *Intent Evaluator* interfaces with the Monitoring API and collects the metrics corresponding with the intents defined for a particular NEMO workload. Then, it evaluates whether the metrics satisfy the targets set by the NEMO client. Subsequently, the updated values that have been collected are stored in the PPEF database and communicated through RabbitMQ to either the meta-Orchestrator or mNCC.

The PAC interfaces internally with the *PRESS manager*, the PPEF *Analytics Engine* and a database. In addition, it interfaces with the main communication channel of the NEMO meta-OS, RabbitMQ, enabling it to communicate its service monitoring analysis, metrics, and alerts to the meta-Orchestrator, CFDRL, MOCA, and mNCC components.

3.4 NEMO Cluster monitoring

PPEF is responsible for deploying monitoring tools which are responsible for collecting the cluster resource consumption measurements from the NEMO incorporated infrastructure that fall into the AIoT, Edge and Cloud continuum. The PPEF monitoring the CPU, RAM and HD resource consumption from each environment that is managed by the NEMO meta-OS, see Table 8.

Table 8: Cluster registration KPIs

KPI	Description	Target Value
Availability	The percentage of time that the cluster is up (99.9%, 99%, 90%)	Integer value
Green Energy	The percentage of RES powering the cluster. (0%,20%,40%,60%,80%,100%)	Integer value
Cost	The cost type of a cluster (low cost, high performance)	String value
CPU base rate	The CPU cost of the cluster by the CPU capacity of the cluster (in milli-tokens)	Integer value in milli-tokens
Memory base rate	The memory cost of the cluster by the memory capacity of the cluster (in milli-tokens)	Integer value in milli-tokens

3.5 PPEF interactions and interfaces

This section provides a high-level description of the interactions that concern the PPEF component within NEMO meta-OS. The integration results that correspond to the listed interactions are presented in D4.2 and will be further updated in D4.3. The latter will also include relevant integration activities that concern the 3rd parties that are introduced to the NEMO project through Open Call 1 and 2.

3.5.1 Intent-based API

The PPEF component interfaces with the Intent-based API for collecting the various intents that have been provided by the user in the framework of NEMO workload registration process. The interaction of the PPEF with the intent-based API is further supported by the addition of an intent validator that works as a filter over the attributes that are assigned as *expectation/target values* by the NEMO service provider. This process ensures the proper configuration of the intents that are consumed and monitored by the NEMO meta-OS.

3.5.2 LCM

The PPEF module interfaces with the LCM module to which it dispatches metrics that are monitored, and which are associated with NEMO cluster monitoring and workload intents. In addition, PPEF via its *Analytics Engine* is able to provide additional statistical insights into the collected measurements.

3.5.3 meta-Orchestrator

The PPEF dispatches to the meta-Orchestrator metrics that are monitored, and which are associated with NEMO cluster and workload intents.

3.5.4 CMDT

CMDT consumes the PPEF workload intent related information that corresponds to the NEMO-hosted workloads.

3.5.5 CFDRL

The PPEF module interacts with the CFDRL component and communicates in fixed time intervals the NEMO-hosted cluster and workload measurements that are collected via the monitoring tools deployed in the NEMO clusters. CFDRL capitalizes on the collected information for its decision-making functionality over the NEMO-hosted workloads.

3.5.6 MOCA

The PPEF communicates to the MOCA the monitored information that concerns both the NEMO governed clusters and the NEMO hosted workloads supporting the accounting and billing functionality that is offered by the MOCA.

3.5.7 RabbitMQ

The data collected by the PPEF component is communicated both to the intent-based API and to the NEMO components via the RabbitMQ module which establishes the main communication backbone of NEMO.

3.6 Conclusion

The final version of the PPEF is described here in the context of D3.3. The PPEF integration with the NEMO meta-OS has been presented in detail in D4.2 which details the first integrated NEMO meta-OS framework. The final integration results that concern the PPEF component will be further updated and described in D4.3. The final version of the PPEF is available in the project's Eclipse GitLab repository. The final development activities pertaining to the PPEF component along with the deployment and the proper configuration of the PPEF monitoring tools, namely Prometheus and Kepler, on the NEMO development and integration environments (development, staging production) hosted in OneLab facilities and in NEMO pilot related infrastructures.

The final version of the PPEF incorporates new features that were implemented in PPEF. Specifically, a new Intent (*Machine Learning Intent*) was included in the list of the intents that are available to the NEMO service provider, the expectation/target *validation filter* and the code refactoring and implementation enhancements that concern the service scale out process that is triggered by the CFDRL workload scheduling optimization functionality.

The PPEF component along with the rest of the NEMO meta-OS will be further validated both in the framework of the NEMO pilots and the associated use cases and in the context of the Open Call 1 and 2 integration and validation activities.

4 Cybersecurity & Digital Identity Attestation

4.1 Overview

The final version of the access control framework within NEMO implements a sophisticated Identity and Access Management (IAM) system. which enforces granular access rights for individual users and groups to specified resources. In that respect the Identity Management sub-module covers the lifecycle of user identities, including the creation and deletion of user identities, along with the processes of provisioning and de-provisioning user access rights. As has been proved in the last phase of the project the Identity Management modules successfully manage user identities efficiently and securely, from their initial establishment to their eventual removal. The Access Management sub-module performs authentication, authorization, and policy management. As proved in the last phase of the NEMO project, this module guarantees that only users with the appropriate permissions can access specific resources, while it also enforces and monitors, in a continuous manner, the access policies which are constantly adapting to the changing security requirements; thus, this module guarantees the security, confidentiality and integrity of the overall NEMO system.

The final version of the Network Intercommunication Security module utilizes a message broker incorporated with the most widely used open-source identity and management sub-system (Keycloak[25]) and the Identity and Management sub-module of NEMO. In that respect it enables secure and authenticated communication and synchronization among the NEMO modules supporting secure message routing, queuing, and transformation and thus allowing the loose coupling of the message sender and the receiver. The Network Intercommunication Security module, as it has been proved in the last phase of the project, supports full flexibility and efficient intercommunication of the NEMO modules while also triggering high reliability and scalability.

NEMO's source code projects on the Eclipse Foundation Gitlab, starting with the "meta-Orchestrator-api", will feature CICD recipes to build and produce cybersecurity metadata artefacts such as SBOM and cryptographic signatures. This enforcement of cybersecurity supply chain workflows, or SSDLC, strengthens the level of cybersecurity of NEMO's software solutions and helps in meeting requirements from the European Cyber Resilient Act and NIS2 directives.

4.2 Architecture and Approach

The overall architecture has not changed from the one that has been analytically described in D3.2, within the last period all the components have been fully verified and evaluated.

4.2.1 Identity and Management Module

The NEMO Access Control was initially integrated with the oAuth2.0 plugin for security. In this updated version the integration of NEMO Access Control with the Kong Prometheus plugin[26], which allows the exposure of workload network metrics, such as its bandwidth and latency, through a Prometheus instance is reported. The metrics are scraped by the PPEF and provided for querying. These metrics can prove useful to pinpoint any slowdowns in the workload, which can affect the overall performance and experience provided by the workload and possibly detect attack attempts (e.g. DoS attacks).

To better demonstrate the plugin, there is deployment of a simple NGINX server workload, which serves a simple login page, Figure 43.

[25] Keycloak: https://www.keycloak.org/
[26] https://docs.konghq.com/hub/kong-inc/prometheus/

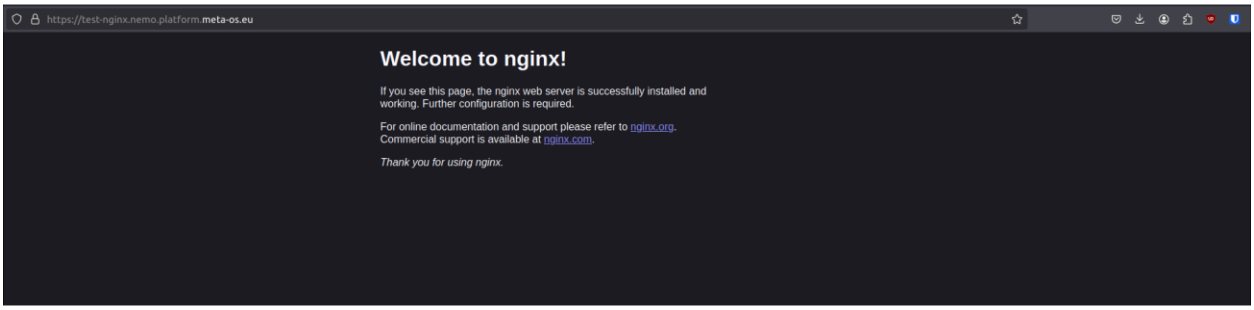


Figure 43: Test NGINX server

The workload has been registered automatically in Access Control, with the usage of the appropriate annotations in its K8S Ingress, further details can be found on deliverable D4.2. This automation method for registering workloads in NEMO Access Control has been described in more detail in deliverable D4.2. Figure 44 focuses on the *kong.com/plugins* annotation, which is responsible for registering the workload in Access Control's Kong service.

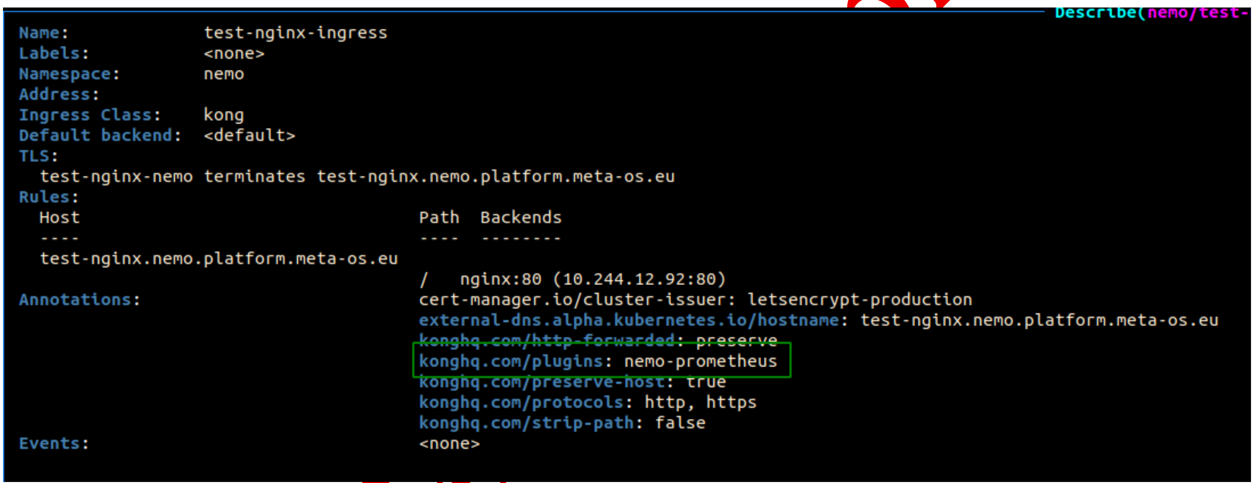


Figure 44: Test NGINX Ingress description.

Now, the necessary Kong service, route and plugin have been added successfully to Kong as demonstrated in Figure 45, Figure 46 and Figure 47.

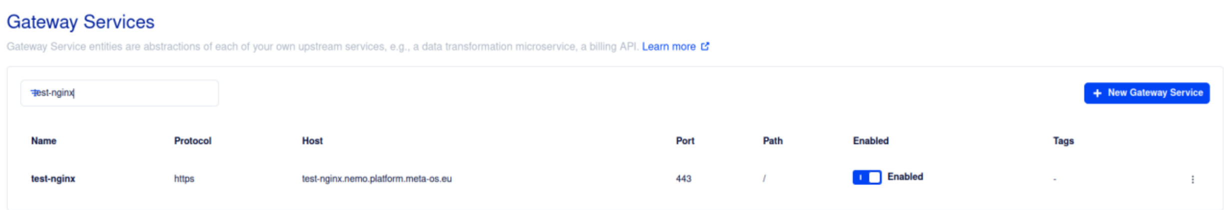


Figure 45: NGINX Kong Service

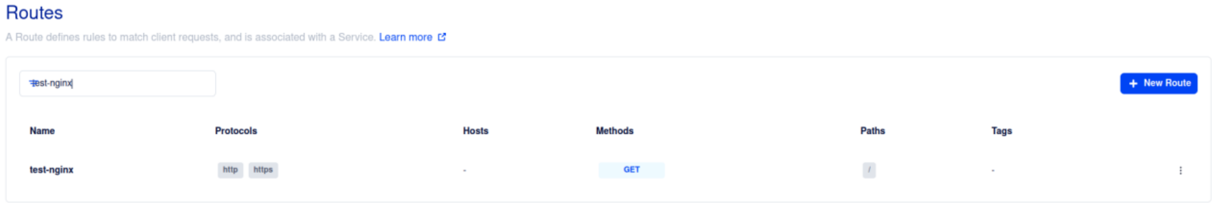


Figure 46: NGINX Kong route

Figure 47: NGINX Prometheus plugin

Figure 48 demonstrates the details of the Prometheus plugin applied to the workload. The plugin has enabled exporting the latency and bandwidth metrics of the workload, the status code metrics, which can expose the total number of requests to the workload.

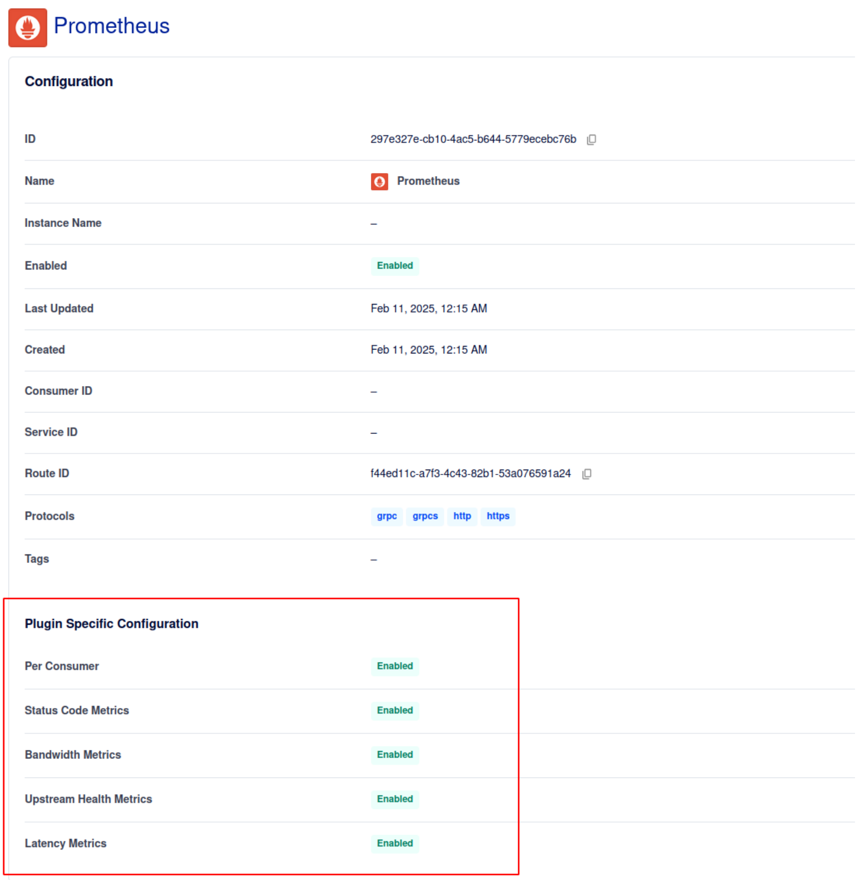


Figure 48: Prometheus plugin details

In this context, the PPEF can expose those metrics. If a query is carried out, for example, the total number of requests it can be observed, Figure 49, the total count, which at the start of the deployment is a total of 1.

Figure 49: NGINX total HTTP request count - initial deployment

If the NGINX server is refreshed a few times, it can be observed that the total count has been incremented (total=3) and that the PPEF, also, exports information for the total count of the different status codes (*200, 404*).

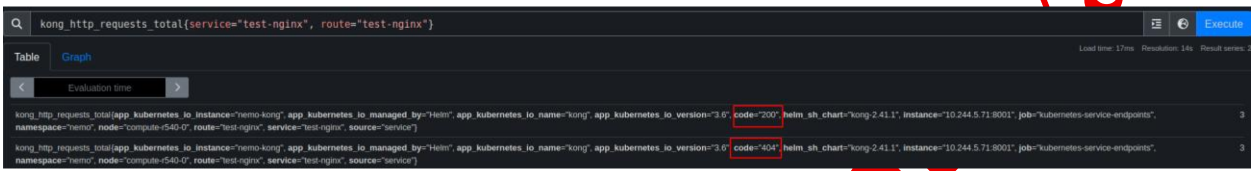


Figure 50: NGINX total HTTP request count – refresh

Figure 51 shows querying the change rate of the NGINX server's bandwidth, in bytes, for a time window of 1 day.

Figure 51: The bandwidth change rate for NGINX

Finally, Figure 52 shows how the latency histograms of the workload can be queried, to observe how much time it takes the Access Control Kong to process the request to the server (*ms*).

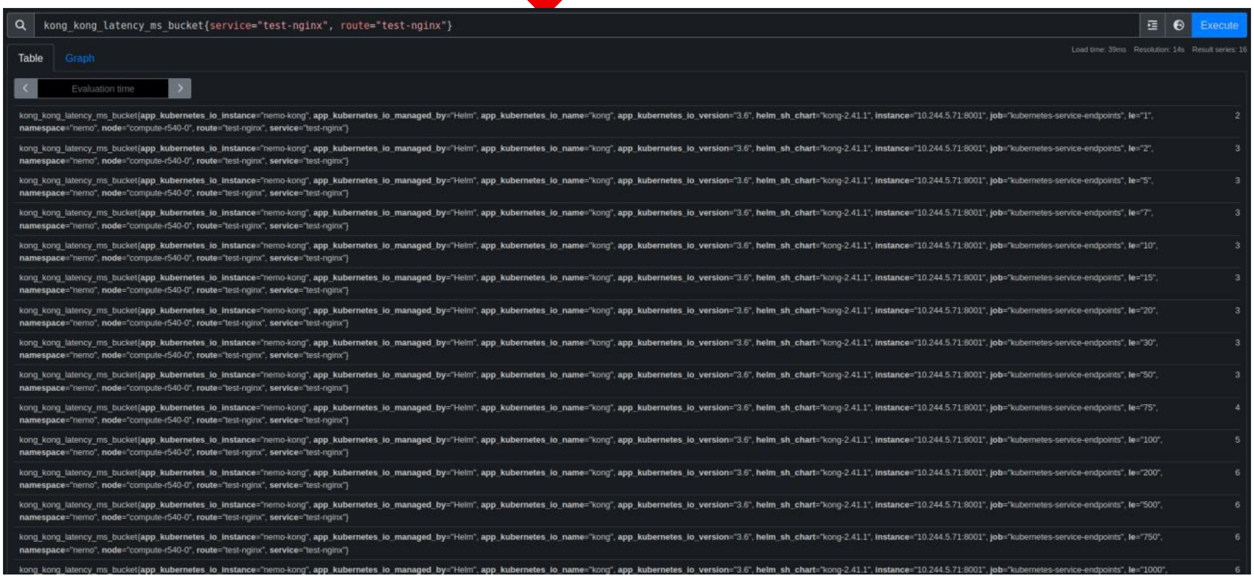


Figure 52: NGINX server latency histograms

4.2.2 News CNAPP & Software Supply Chain

This paragraph will first give a brief reminder of previous deliverables. Then it addresses the signature validation at runtime.

D3.1 describes CNAPPs - Cloud-Native Application Protection Platforms - in general and with a focus on runtime cybersecurity probes such as Falco.

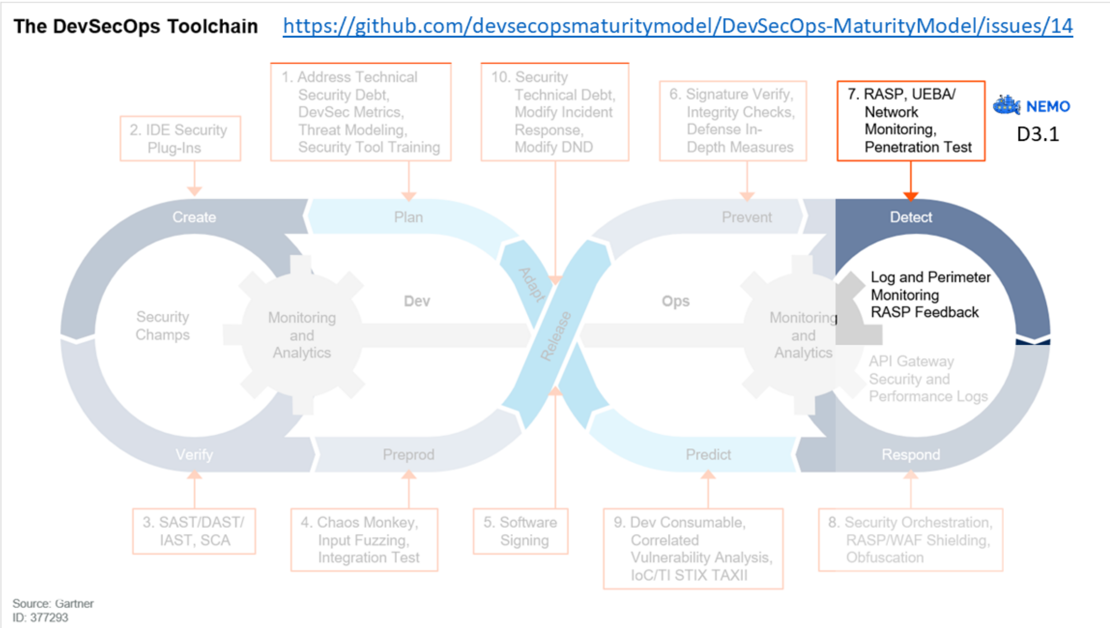


Figure 53 NEMO D3.1 focus on the detection at runtime (step 7 Gartner DevSecOps)

D3.2 took a “shift left approach” to address the topic of software supply chain security during development time with tools like software composition analyzers that create SBOM and attestation of provenance, as well as software signing tools. D3.2 gives the example of Goreleaser as a CICD tool to release Golang application with SBOM and signatures.

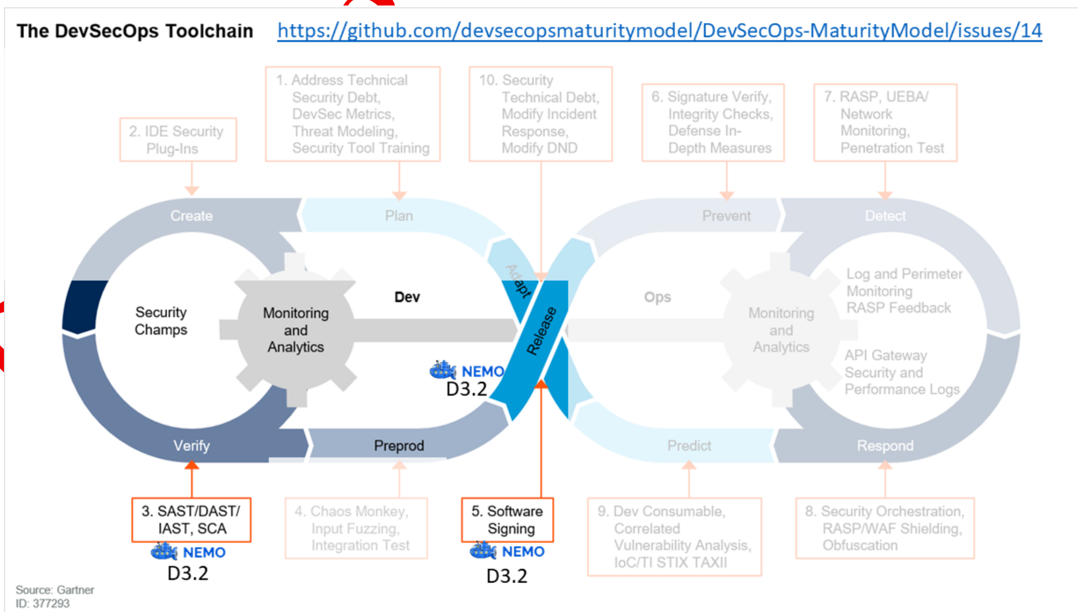


Figure 54 NEMO D3.2 Focuses on Software Composition Analysis (Step 3 Gartner DevSecOps) and Software Signing (Step 5)

D3.3 goes back to "runtime" and illustrates step 6 of Gartner DevSecOps, which is signature verification at runtime. These steps follow D3.2, which purpose is to generate the metadata such as OCI - Open Container Initiative – image signatures or attestation that will be verified at runtime.

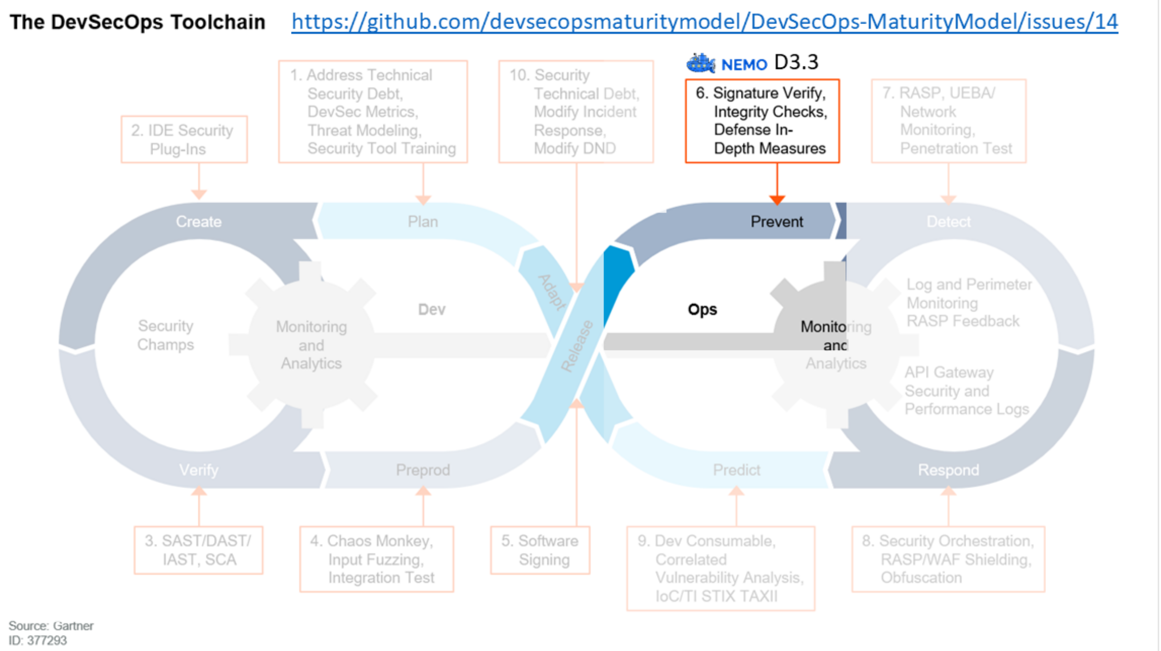


Figure 55 NEMO D3.3 focus on signature verification at runtime

In the NEMO project, this software signature verification at runtime uses Kubernetes Validation Admission and Admission Controller features from Figure 56.

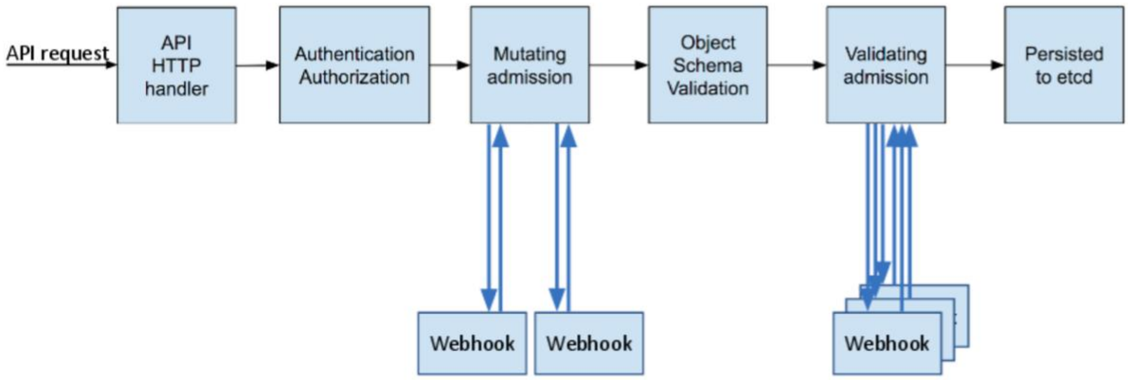


Figure 56 A Guide to Kubernetes Admission Controllers[27]. NEMO focuses on validation admission.

Figure 57, Figure 58 and Figure 59 show the principle behind this signature verification that uses Kubernetes Admission Controllers. The application to deploy can be anything, from NEMO meta-Orchestrator-api to KeyCloak, as long as they the apps provide the metadata like signatures.

[27] A Guide to Kubernetes Admission Controllers | Kubernetes

As can be observed in Figure 57, from left to right, a DevSecOps person is responsible for writing validation policy for the Kubernetes admission. The DevSecOps also configure or use the OCI image registry where software artefact and software signatures have been released. And the DevSecOps also writes the Kubernetes manifests that correspond to the deployment of an application.

The DevSecOps uses the Kubernetes client of its choice, including GitOps, to deploy the policy manifests on the Kubernetes cluster, and then the application manifests. Indeed, the policy and admission controller must be configured and deployed before the application. This enforced and protected the application. On Figure 57, the policy could be phrased like this:

“In order to be pulled, an OCI container image must have a valid signature verified by the admission controller from an accessible root of trust. If the OCI signature is valid, then the image is pulled. Otherwise, the DevSecOps chooses a strict policy which prevents the OCI container image from being pulled if the signature verification fails. This is the case in Figure 58. Or the DevSecOps could chose a less strict policy which pull the OCI container image even if the signature is not valid but warns the user about this with an alert message. This is the case in Figure 59.”

To choose between a strict policy “no pull if not a valid signature”, or a warning only policy “pull even if not a valid signature but warn the user”, this depends on the use case. In testing or pre-production environments, the signatures might not be generated by the CICD when it is not a software artefact from a release branch. In this situation, a warning policy is enough as testing the app is more important than protecting the apps. In a production environment, strict policy should be implemented.

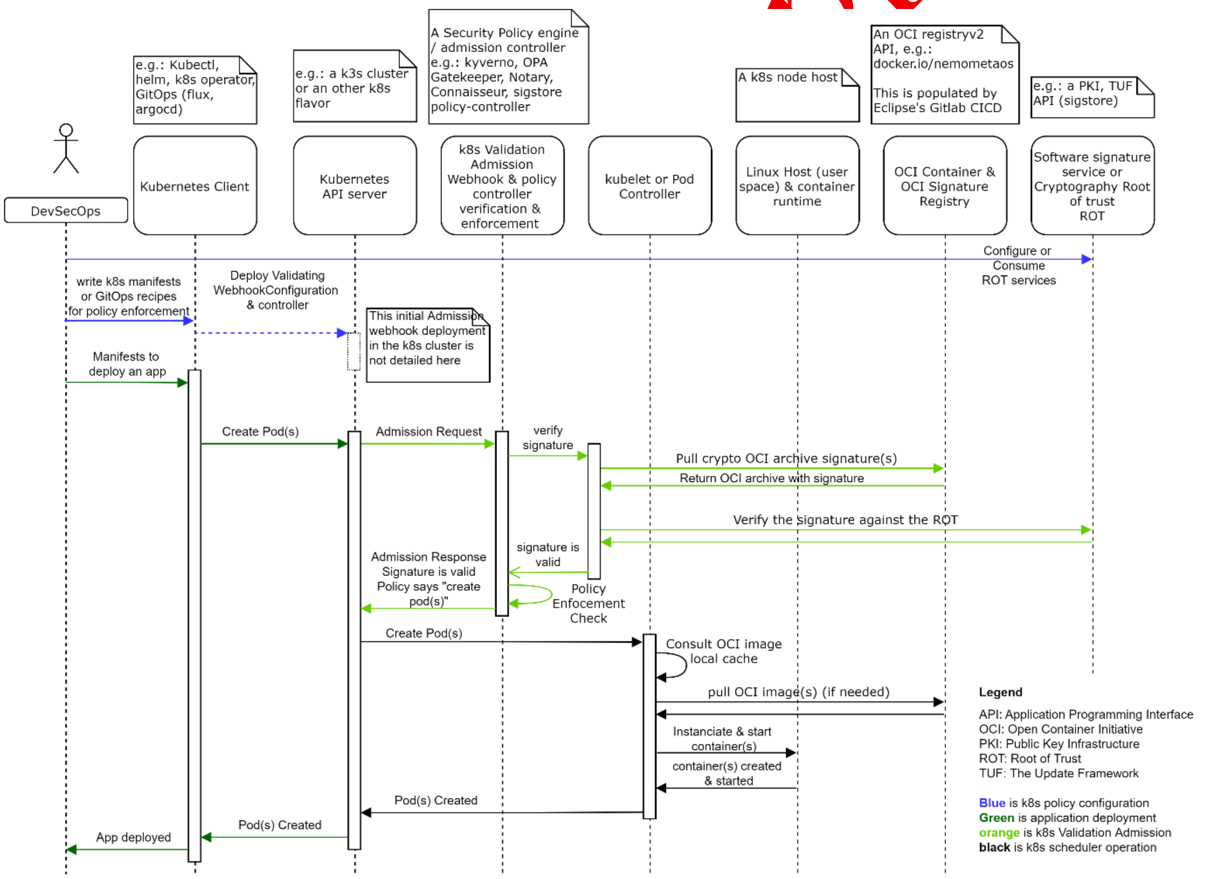


Figure 57 OCI Image Verification at Runtime: signature is valid, and policy let the OCI container image be pulled

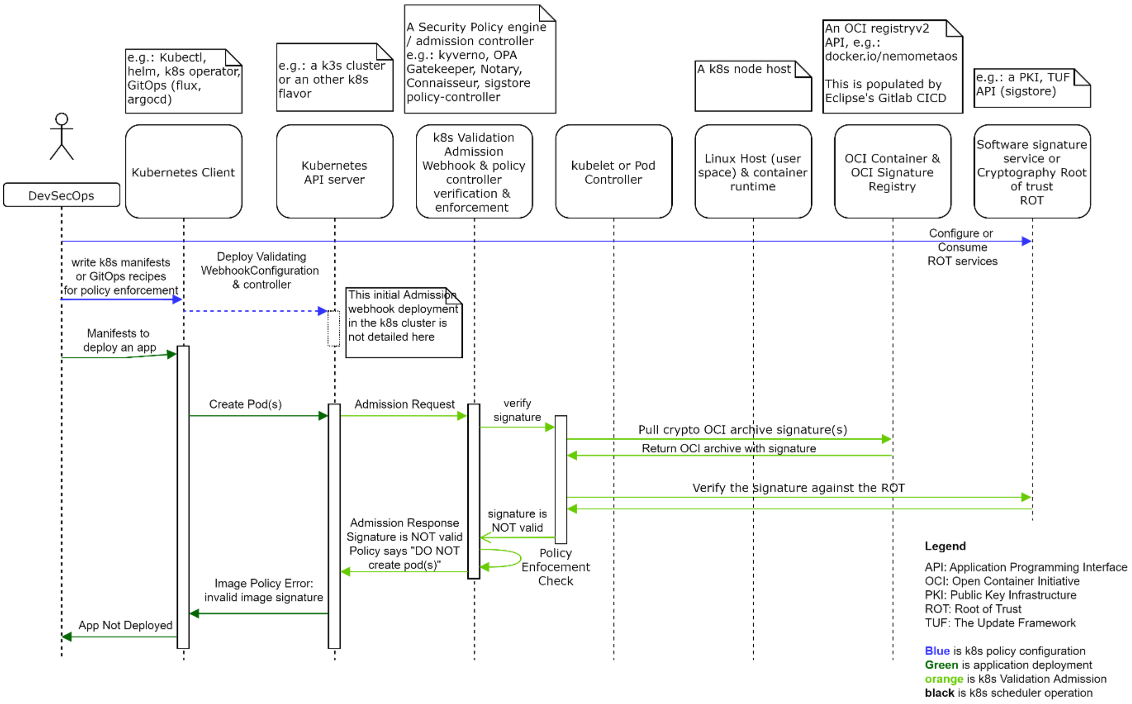


Figure 58 OCI Image Verification at Runtime: signature is not valid, and policies says not to pull OCI container image (strict policy)

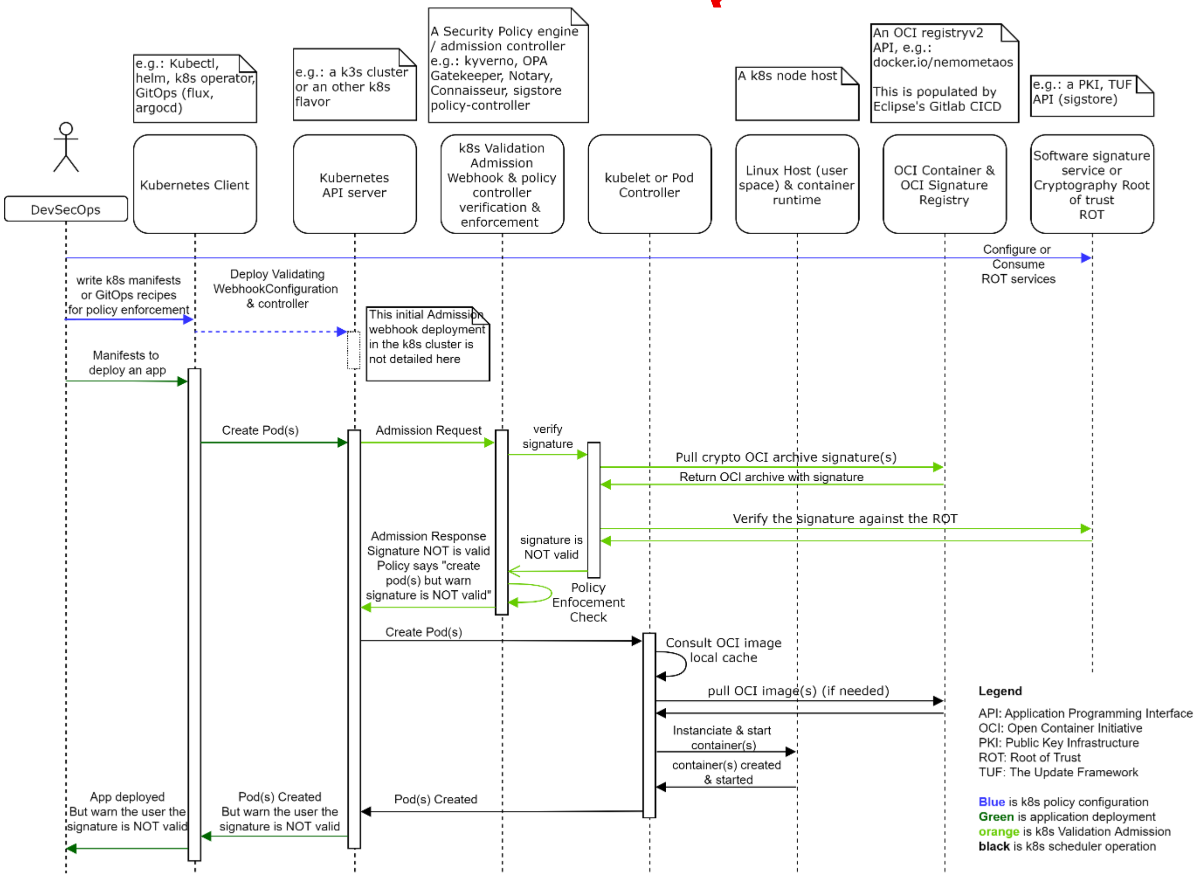


Figure 59 OCI Image Verification at Runtime: signature is not valid, but policy says pull the OCI container image but warn user (warn policy)

Future work might include doing the same verification with in-toto SLSA attestation of provenance, which includes both a signature and information about the software supply chain. One could write a policy to prevent installation of OCI image pulled from a wrong registry.

4.3 Conclusion

NEMO deliverables D3.1, D3.2, and D3.3 demonstrate the developments, validation, and ways to utilize the intercommunication system, the identity management system, and the principle of CNAPPs to protect applications during development and runtime. For CNAPPS, it has been demonstrated the full loop, where both development and runtime protect different aspects of an application's lifecycle.

The final version of the Cybersecurity and Digital Identity Attestation framework developed within the NEMO project consolidates essential security components to protect services operating across AIoT, Edge, and Cloud environments.

IAM system that offers precise control over who can access what. It is built to be flexible, adapt to security needs, and make access decisions based on context and risk.

Another important point concerns built-in telemetry powered by Prometheus, seamlessly integrated through Kong plugins. These two tools give real-time insight into system performance and health while also helping to discover early warning signs of potential issues, like service attacks, by analyzing traffic trends and user behavior.

The NEMO framework adopts a proactive and adaptable approach by combining identity management, contextual security enforcement, and continuous monitoring. The platform manages digital identities independently, secures communications, and ensures that all active components comply with strict security policies both now and in the future.

5 NEMO meta-Orchestrator

The NEMO meta-Orchestrator (MO) is an open-source software system designed to manage and optimize the distribution of computing workloads across the NEMO cluster network. This core component serves as the NEMO platform's central component, allowing efficient and smooth coordination between multiple NEMO components to allocate intelligently logical resources.

5.1 Overview

The meta-Orchestrator leverages different technologies and tools to achieve the goals, at the same time, the component splits itself into multiple subcomponents, as shown in Figure 60. Each subcomponent has a different main goal.

One of these goals is to deploy NEMO ad-hoc workloads built at a higher level from the vanilla Kubernetes manifests. These deployments are over a selection of clusters; this selection is the NEMO cluster network, or, in other words, all clusters that form IoT-Edge-Cloud devices for the NEMO platform also, considering that the role played by the MO is very important in managing this complex resource and service flow to enable NEMO to work effectively in a highly dynamic and heterogeneous environment.

On other hand, the MO controls the Placement[28] of the network cluster in order to optimize those workloads deployments and not overload the network, so also to prioritize green-energy cluster over the non-renewable energy clusters or fossil fuels energy clusters.

It is important to remark that in the initial stages of development of this component, Golang, RabbitMQ, Kubernetes, and REST API technologies have been chosen as the stack to meet its requirements. Interaction with other components will be mostly asynchronous through RabbitMQ queues, but synchronous HTTP direct communication via the REST API can also be used when needed.

At the top, the MO is a high-level controller over container orchestration clusters such as Kubernetes, coordinating resources from the IoT edge to the cloud continuum, ensuring workloads are deployed without issues. The NEMO MO comprises key subcomponents and services such as MO API, Deployment Controller (DC), and the IBMC (Intent-Based Migration Controller), which effectively orchestrate computing workflows.

[28] Placement: https://open-cluster-management.io/docs/concepts/content-placement/placement/

5.2 Architecture and Approach

As previously mentioned, the meta-Orchestrator's functionality is divided into multiple subcomponents to achieve a better software life cycle. This structured and maintainable software lifecycle, enabling better separation of concerns, scalability, and extensibility.

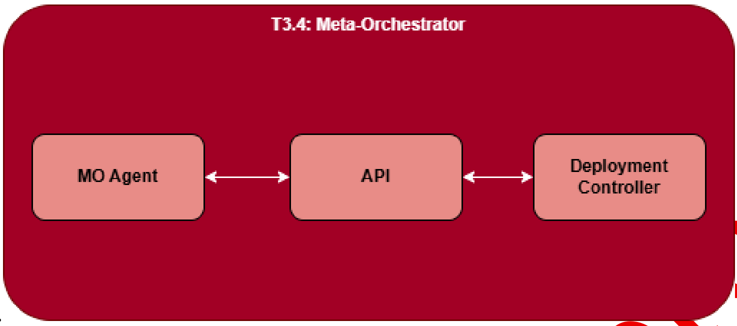


Figure 60: meta-Orchestrator Subcomponents

The MO is composed of three main subcomponents, Figure 60:

- MO API: This subcomponent handles the CRUD operations of the NEMO cluster network and operations about the horizontal scaling of the NEMO workloads. From D3.2 to this D3.3 some updates have been made regarding the API.
- MO Agent: This component is based on Event-Driven Architecture (EDA) and communicates with the rest of the NEMO components using RabbitMQ queues. The MO Agent handles multiple queues and depending on the queues and messages, calls different endpoints with different behaviours from the MO API.
- Deployment Controller (DC): This subcomponent specializes in workload deployments and has the logic for MO placements. It calls the API to get cluster-related metrics and decides which cluster to use for future workload deployments.

5.2.1 meta-Orchestrator Hub API

Since the first version of the API, the service has been evolving to adapt to the project needs. Consequently, the final architecture slightly varied as it is represented in Figure 61

In context and terms of Open Cluster Management tool (OCM[29]), this API is deployed inside the hub, based on the hub-spoke architecture[30], the hub is the central cluster where the decision-making happens. All the services and subcomponents related to MO are deployed and working inside the hub.

From D3.2 API to D3.3, some changes were made to adapt the service to the NEMO platform. In the next paragraphs the changes will be explained in deeper detail.

The MO Agent has taken over the reading functionality from the queues; this agent used to write into queues but now also reads them, takes messages, and calls the API endpoints depending on the messages and queues listening. Basically, this agent triggers different behaviours inside the MO, depending on what is receiving from other NEMO components such as the Intent-Base API, CFDRL or MOCA.

[29] OCM: https://open-cluster-management.io/
[30] Hub-spoke: https://open-cluster-management.io/docs/concepts/architecture/

Furthermore, MO API and MO Agent are services based on Kubernetes with inherited capabilities for modifying the Vertical and Horizontal scaling to handle petitions without losses. It is also possible to have multiple MO Agent instances inside the NEMO platform.

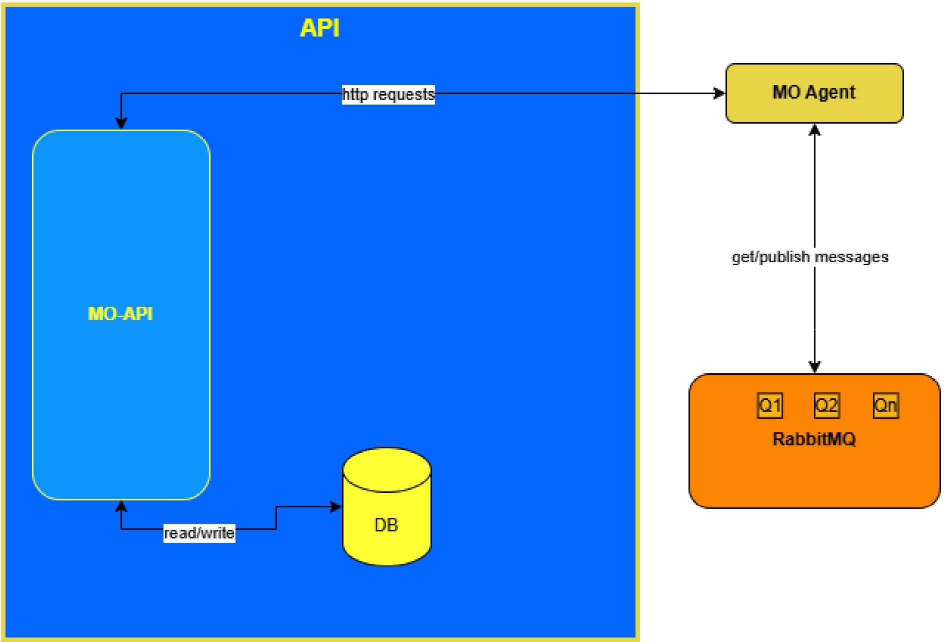


Figure 61: MO API Architecture.

The updates previously mentioned are:

- Removing the worker subcomponents from the MO, this part has been absorbed by the MO Agent, assuming its functionality inside the component.
- Adding new endpoints to handle the needs of the other components and at the same time the NEMO needs. See Figure 62.
- Added JWT Keycloak authentication, based on the Access Control component (Task 3.2)

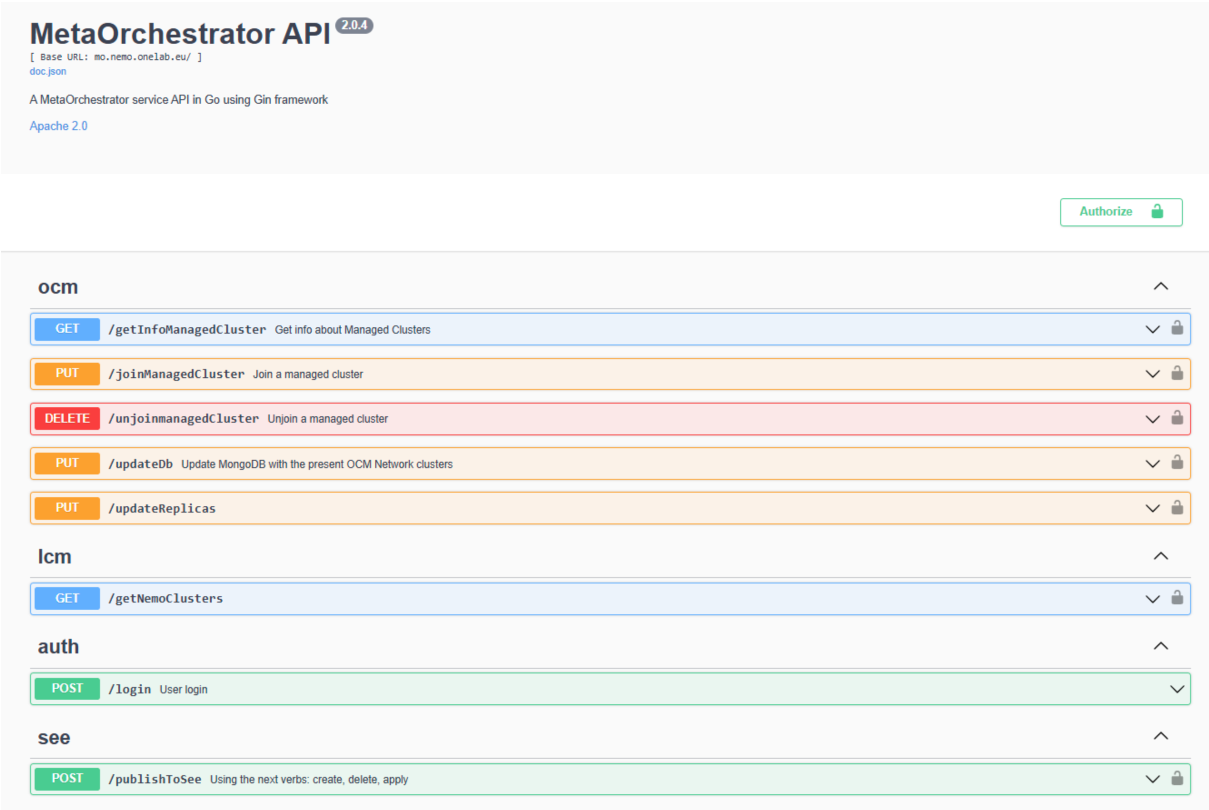


Figure 62: MO API endpoints.

5.2.1.1 API Endpoints

OCM

This section is related to the NEMO cluster network and CRUD cluster operations. Inside the code, the MO API uses the OCM libraries and tools to exploit multi-cluster-level operations.

- GET /getInfoManagedCluster: This endpoint gets all the clusters inside the NEMO cluster network. Previously, they must be joined using the */joinManagedCluster* endpoint. See Figure 63 to see the output.


```
{
            "Name": "pro-onelab",
            "UID": "ff0d5030-5d7c-487d-ad29-42c560251070",
            "Version": "v1.30.7",
            "ManagedAPI": "https://xxx.yyy.nemo.onelab.eu:6443",
            "Capacity": {
                "cpu": "32",
                "ephemeral-storage": "387753320Ki",
                "hugepages-1Gi": "0",
                "hugepages-2Mi": "0",
                "memory": "60826000Ki",
                "nvidia.com/gpu": "16",
                "pods": "550"
            },
```

```
            "Allocatable": {
                "cpu": "32",
                "ephemeral-storage": "357353459123",
                "hugepages-1Gi": "0",
                "hugepages-2Mi": "0",
                "memory": "60314000Ki",
                "nvidia.com/gpu": "16",
                "pods": "550"
            },
            "CreationTimestamp": "2025-04-03T12:24:13Z",
            "Availability": "99.9%",
            "Cpus": 32,
            "Memory": 62,
            "Storage": 1350,
            "GreenEnergy": "20%",
            "Cost": "low_cost",
            "CpuBaseRate": 10,
            "MemoryBaseRate": 10,
            "Status": "True"
}
```


Figure 63: Retrieve spoke clusters from the NEMO Cluster Network endpoint.

- PUT /joinManagedCluster: Join a cluster which was not present in the cluster network earlier. The endpoint is idempotent; only a cluster with that name can exist, no matter how many HTTP requests get triggered. The JSON payload contents key metrics like CPU count, memory, and storage capacities, qualitative contents, such as availability cost and green energy usage.


```
{
    "availability": "80%",
    "cluster_name": "dev-onelab",
    "cost": "low_cost",
    "cpu_base_rate": 10,
    "cpus": 20,
    "green_energy": "20%",
    "managed_api": "https://api.main.nemo.onelab.eu:6443",
    "memory": 200,
    "memory_base_rate": 10,
    "storage": 300
}
```


Figure 64: Payload for joinManagedCluster endpoint.

- DELETE /unjoinmanagedCluster: This endpoint is the opposite of previous endpoint and removes a chosen cluster from the NEMO cluster network. In the payload there are the fields “managed_api” to put the API Kubernetes API and the name of the cluster.

- PUT /updateDb: This endpoint updates the OCM and DB registries to align and ensure the same cluster on both sides. Basically, thake the OCM persistence and clone it into the DB.

- PUT /updateReplicas: this endpoint updates the number of replicas of the NEMO workloads. In the payload are shared the "cluster_name" where the workload is deployed, "workload_id" and "number_replicas". See Figure 65 below to see an example of the payload call.

```
{
  "cluster_name": "staging-cluster",
  "workload_id": "cbcb208a-d535-434b-bb35-217a64bd516c",
  "number_replicas": 3,
}
```

Figure 65: Update replicas endpoint payload for triggering Horizontal Scaling

LCM

This section is about the endpoint used in the Guided User Interface (GUI) of the NEMO project. This call returns all the names and Kubernetes's API URL for the cluster that MO can handle, understanding it as capable of performing CRUD operations with these clusters.

- GET /getNemoClusters: This endpoint returns the clusters that can be handled within the CRUD operations. The NEMO platform's LCM GUI uses this endpoint.

AUTH

This section is related to the authentication and security within the meta-Orchestrator API.

- POST /login: This endpoint is used by other NEMO services to properly authenticate and get a JWT token. This endpoint has been removed and replaced by the Keycloak authentication.

SEE

This section is related to SEE services such as the unikernel and integrations with MO.

- POST /publishToSee: this post triggers a SEE or unikernel deployment inside the NEMO platform and set the replying queue ("reply_to") in the "body" field there's go the SEE resource to deploy. See Figure 66.

```
{
    "reply_to": "amq.gen-Xa2",
    "verb": "create",
    "body": {
        "apiVersion": "v1",
        "kind": "Pod",
        "metadata": {
            "name": "nginx"
        },
        "spec": {
            "containers": [
                {
                    "name": "nginx",
                    "image": "nginx:latest",
                    "ports": [
```

```
                              {
                                  "containerPort": 80
                              }
                          ]
                      }
                  ]
              }
          }
      }
```


Figure 66:Payload to deploy SEE resources.

5.2.2 meta-Orchestrator Agent (MO Agent)

The main point of this component is to have a translator for the rest of the NEMO components using Event-Driven architecture (AMQP) and transform this into HTTP synchronous requests against the MO API or other APIs from projects.

As seen previously in the overall MO architecture, this MO subcomponent can also be defined as an asynchronous agent. MO Agent is handling multiple queues using the Golang library Watermill[31] that is designed to facilitate the construction of event-driven applications that provide robust tools for message-oriented architecture. From the MO Agent, Watermill simplifies publishing and reading the queues and adds an abstraction layer over the RabbitMQ library amqp091-go[32].

Inside the code, there are two handlers linked at two queues; these two queues have different logic and behavior; see Figure 67 for deeper details:

- **Cluster Registration or deregistration flow**: From LCM GUI, go through MOCA and after the MO Agent. The MO Agent is listening to a queue to get messages about the potential registration or deregistration of clusters.

- **Horizontal Scaling or Descaling (Number of Replicas/Pods):** At the top of Kubernetes, the MO API and the MO Agent have been built to respond to the CFDRL component's demands to increase or decrease the number of pods.

The MO agent is not just a task runner. It is shaping up as a general-purpose event-driven control planer for multi-cluster or hybrid cloud orchestration, being a flexible and compatible approach to automation, which could easily be extended and integrated into DevOps operations.

[31] Watermill.io: https://watermill.io/docs/getting-started/
[32] Ampq091.go: https://github.com/rabbitmq/amqp091-go

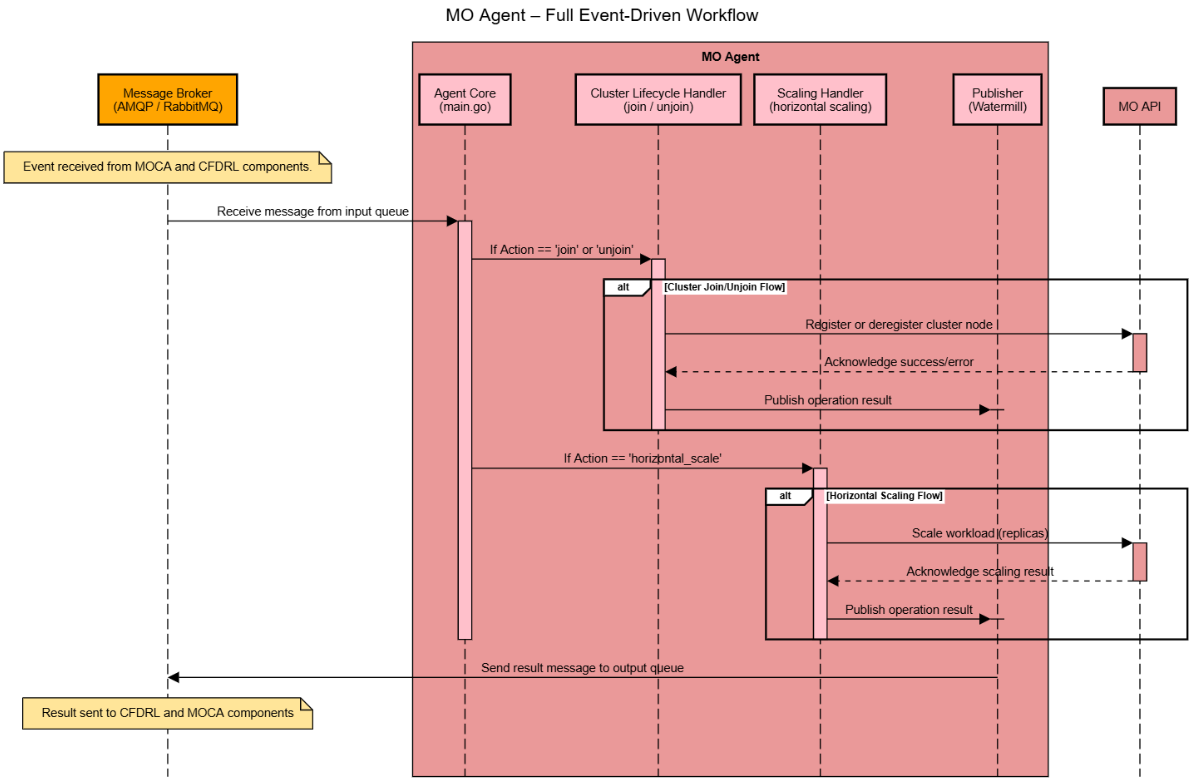


Figure 67: MO Agent sequence diagram

5.2.2.1 Sequence Diagram Steps

Based on the above figure, Figure 67. The workflow steps are:

1. **A message is emitted to the input queue**

An external NEMO component sends an operation request (e.g., join, unjoin, or horizontal scaling) to the **input queue** hosted on a message broker such as RabbitMQ. This message contains metadata identifying the action, target cluster, operation ID, type of operation (join/unjoin).

2. **MO Agent Core receives and parses the message**

The **Agent Core** (main.go) subscribes to the message queue using Watermill and listens for incoming events. Upon receiving a message, it parses the JSON payload and evaluates the action field to determine how to route the request.

The Agent Core routes the request to the correct handler

Based on the value of action, the Agent Core dispatches the message to one of two handlers:

- For join or unjoin, it invokes the **Cluster Lifecycle Handler**
- For horizontal_scale, it invokes the **Scaling Handler**

This routing logic is abstracted using a configurable function map.

3. **The handler constructs and sends a request to the MO API**

The appropriate handler (either Cluster Lifecycle or Scaling) builds an API request that reflects the message's intent. This typically includes the target cluster name, identifiers, resource parameters (e.g., replica count), or lifecycle directives. It then sends this request to MO API. The Agent finally queued for the reply to messages from the MO API into another different queue providing full feedback on the process.

5.2.3 Deployment Controller (DC)

As described in Section 5.1 Overview, the Deployment Controller (DC) is responsible for managing workload deployments. It leverages RabbitMQ to listen for incoming messages in a queue, where the Intent-API publishes workload instances.

Upon receiving a new message, the DC first extracts the instance_id, a unique identifier for the workload. Using this ID, it queries the Intent-API to retrieve the workload's current status and deployment cluster. If the status is "rendered", the deployment process is triggered.

Next, the DC retrieves cluster metrics from the MO-API for all available clusters. If the message received from the Intent API contains an intent, these metrics are used to identify a cluster that meets the specified requirements. If no cluster meets the criteria, or if the message lacks an intent, the DC selects the cluster with the highest green energy availability for deployment.

Once a cluster is selected, the manifests included in the Intent API message are encapsulated into an OCM[33] ManifestWork[34] and applied to the HUB[35] cluster within the namespace corresponding to the selected cluster. This triggers the propagation of the manifests to the target cluster, ensuring the successful deployment of the workload.

Finally, the DC sends a confirmation message back to the Intent API, updating the workload status to "deployed", thereby closing the deployment loop.

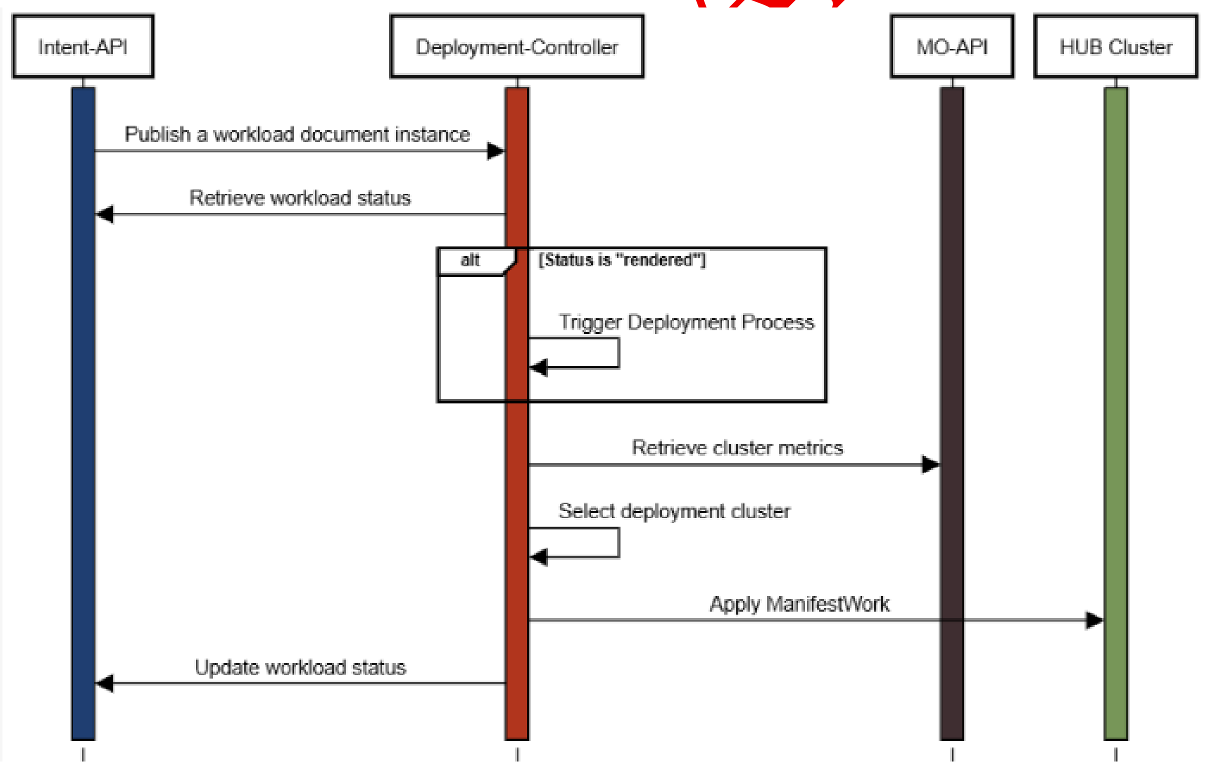


Figure 68: DC Sequence Diagram.

33 https://open-cluster-management.io/
34 ManifestWork | Open Cluster Management
35 Architecture | Open Cluster Management

5.2.3.1 CI/CD

The CI/CD pipeline for the MO follows the standard[36] established in the NEMO project, ensuring consistency and reliability across deployments. GitLab CI is utilized for continuous integration (CI), requiring each component to include a valid Dockerfile to enable deployment within the NEMO environment. Additionally, a .gitlab-ci.yml file must be present in each component's repository. This configuration allows a GitLab runner to automatically build a new container image whenever a commit is pushed. The newly built image is then stored in NEMO's Docker Hub registry[37], ensuring an up-to-date and versioned container repository.

For continuous deployment (CD), FluxCD[38] is employed to automate and streamline the deployment process. The deployment manifests for each component are maintained in the NEMO FluxCD repository[39]. Whenever a new version of a component is tagged in GitLab, FluxCD detects the update and automatically synchronizes the target clusters with the latest changes. This ensures that all deployed services remain current with minimal manual intervention, reducing operational overhead and improving system reliability.

This process is represented in the figures below. Figure 69 illustrates an example of creating a new tag for the Deployment Controller.

In Figure 70, this newly created tag triggers the GitLab runner, which initiates the CI process by building a new image of the component.

Once the build is completed, the updated image is pushed to Docker Hub, as shown in Figure 71. Following this, Figure 72 demonstrates how FluxCD automatically detects the new tag and updates the corresponding deployment manifest for the Deployment Controller.

Finally, Figure 73 shows the verification step, where the target cluster is accessed to confirm that the tag used in the deployed image matches the newly created one.

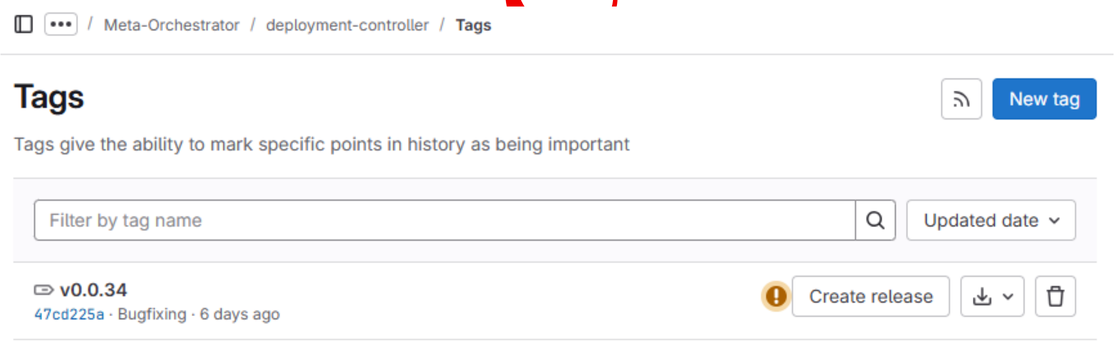


Figure 69: Gitlab Tagging.

[36] CI-CD Integration.md · main · Eclipse Research Labs / NEMO Project / Nemo HowTo · GitLab
[37] https://hub.docker.com/u/nemometaos
[38] https://fluxcd.io/
[39] NEMO / FLUX CD · GitLab

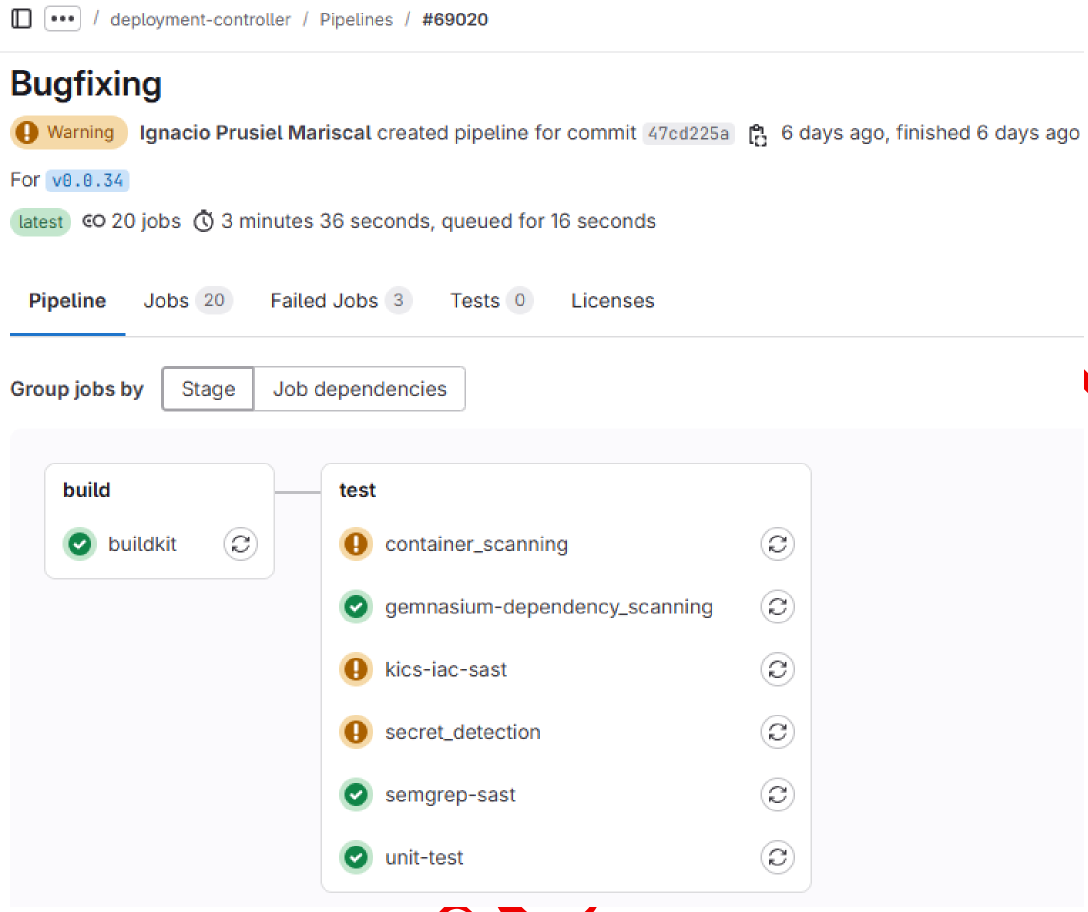
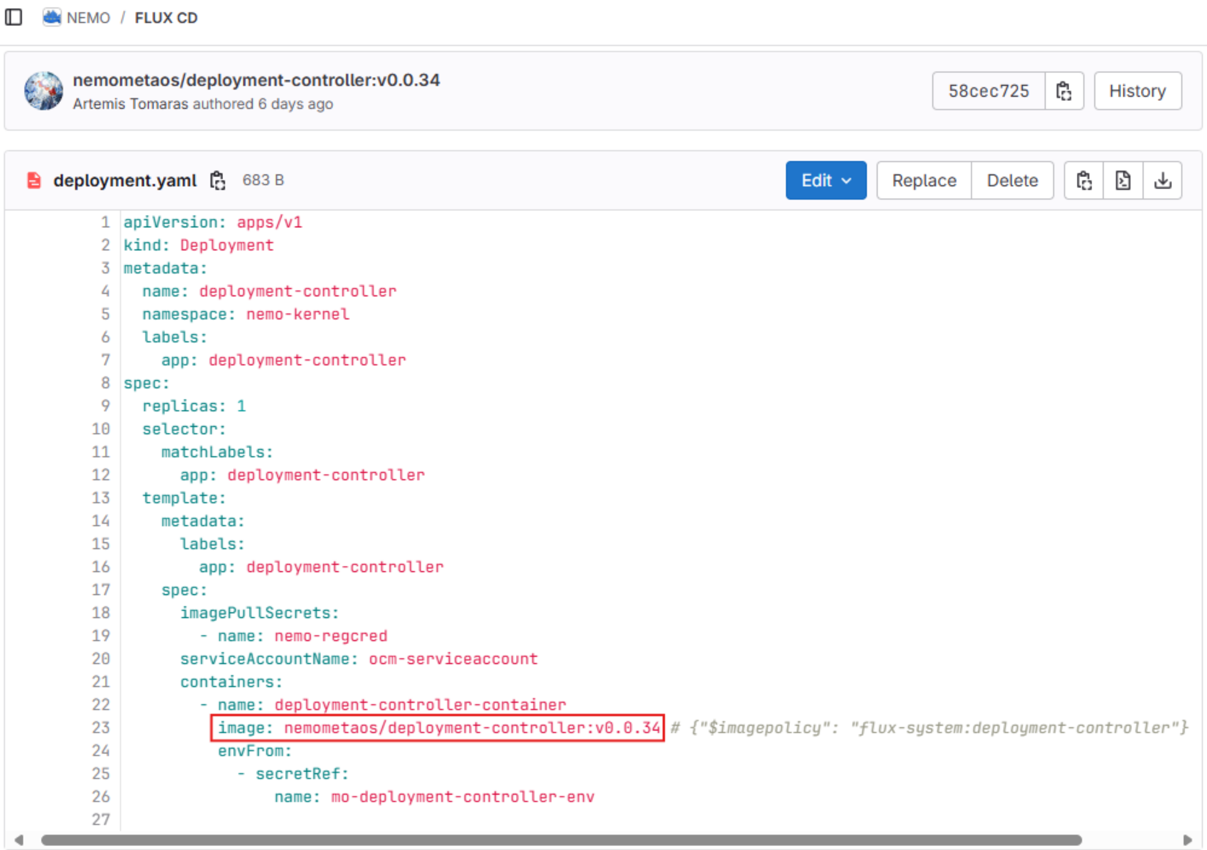


Figure 70: Gitlab CI.

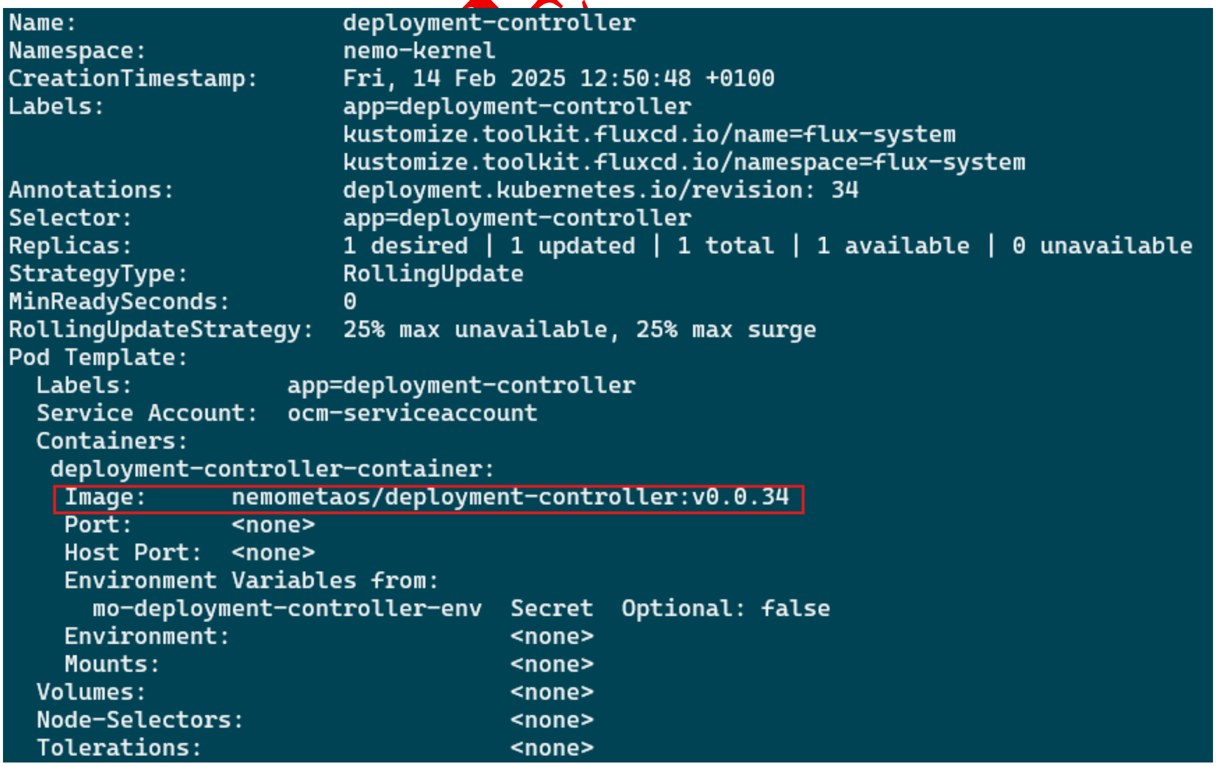
Figure 71: NEMO DockerHub.




```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-controller
  namespace: nemo-kernel
  labels:
    app: deployment-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deployment-controller
  template:
    metadata:
      labels:
        app: deployment-controller
    spec:
      imagePullSecrets:
        - name: nemo-regcred
      serviceAccountName: ocm-serviceaccount
      containers:
        - name: deployment-controller-container
          image: nemometaos/deployment-controller:v0.0.34 # {"$imagepolicy": "flux-system:deployment-controller"}
          envFrom:
            - secretRef:
                name: mo-deployment-controller-env

```

Figure 72: Deployment Manifest.



```
Name:                   deployment-controller
Namespace:              nemo-kernel
CreationTimestamp:      Fri, 14 Feb 2025 12:50:48 +0100
Labels:                 app=deployment-controller
                        kustomize.toolkit.fluxcd.io/name=flux-system
                        kustomize.toolkit.fluxcd.io/namespace=flux-system
Annotations:            deployment.kubernetes.io/revision: 34
Selector:               app=deployment-controller
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:           app=deployment-controller
  Service Account:  ocm-serviceaccount
  Containers:
   deployment-controller-container:
    Image:      nemometaos/deployment-controller:v0.0.34
    Port:       <none>
    Host Port:  <none>
    Environment Variables from:
      mo-deployment-controller-env  Secret  Optional: false
    Environment:                    <none>
    Mounts:                         <none>
  Volumes:                          <none>
  Node-Selectors:                   <none>
  Tolerations:                      <none>
```

Figure 73: Deployment Details.

5.2.3.2 Workload Migration (IBMC-DC)

The Intent-Based Migration Controller (IBMC) is responsible for handling workload migrations within NEMO whenever a new intent is created and published by the Intent-API.

This process directly impacts the Deployment Controller (DC), as the cluster where the workload is deployed changes. However, the issue is resolved in a straightforward manner. Once the migration is completed, meaning the workload's manifests have been successfully backed up in the source cluster and restored in the target cluster, the IBMC sends a message to the DC, instructing it to update the ManifestWork corresponding to the migrated workload.

To ensure continued workload management by OCM, the DC updates the ManifestWork namespace to match the name of the new target cluster. This guarantees that the workload remains properly orchestrated and monitored after migration.

5.2.3.3 Network configuration and creation between NEMO Cluster Network (DC-mNCC)

The MO supports creating virtual networks between pods of different NEMO clusters using the DC subcomponent. The meta–Network Cluster Controller (mNCC) creates an extra layer of communication between clusters using the L2S-M[40] tool. The mNCC communicates with the Intent-Base API and the MO; after exchanging messages, the MO establishes the connection by applying the changes in the managed NEMO's workload.

5.2.3.4 Workload Placement

Placement means scheduling workloads strategically in the best possible place, based on monitoring metrics retrieved. The MO API gathers some metrics as can be seen in Table 9 which includes resource availability, CPU usage, RAM usage, and energy sources. The placement strategy determines which place a given workload runs in.

Value and practical placement are crucial for system performance, resource utilization, and energy efficiency. With good placement reallocation, systems can achieve better performance and lower latency. Moreover, optimized placement helps minimize energy consumption during low-demand periods, for example.

The workload placement strategy needs to be dynamic in a cloud computing environment.

In the NEMO project, there are different levels of placement. NEMO and meta-Orchestrator handle the workloads at multi-cluster levels, meaning they are at a higher level that Kubernetes can manage. While Kubernetes seeks and finds the best nodes inside a cluster with different nodes, heterogeneous or homogeneous, NEMO and meta-Orchestrator handle the placement between different clusters around the NEMO Cluster Network.

Field	Type	Title	Description
cluster_name	string	Cluster name	The name of the Cluster that will be deployed. Must be between 1 and 42 characters.
cpus	number	CPUs	The number of CPUs of the Cluster.
memory	number	Memory	The RAM of the Cluster in GB.
storage	number	Storage	The disk storage of the Cluster in GB.
availability	string	Availability	The percentage of time that the cluster is up (99.9%, 99%, 90%).
green_energy	string	Green energy	The percentage of RES powering the cluster (0%, 20%, 40%, 60%, 80%, 100%).

[40] L2S-M: https://github.com/Networks-it-uc3m/L2S-M

Field	Type	Title	Description
cost	string	Cost	The cost type of a cluster (low cost, high performance). Enum
cpu_base_rate	number	CPU base rate	The CPU cost of the cluster by the CPU capacity of the cluster (in milliseconds).
memory_base_rate	number	Memory base rate	The memory cost of the cluster by the memory capacity of the cluster (in MBs).

Table 9: Cluster Metrics

Regarding the above cluster metrics, the Deployment Controller uses these metrics to place the workload in the best possible cluster from the NEMO cluster network that MO is handling and managing.

5.3 Conclusion

The NEMO meta-Orchestrator (MO), now at this new phase of the project, demonstrates to be a pivotal component of the NEMO platform that coordinates workloads across different scenarios while also working with event-driven communication architecture and has been built for decentralized cluster control and coordination, using tools like OCM, Golang, and RabbitMQ, allowing efficiently manage tasks across distributed systems whether it is handling edge devices or scaling services in real-time. Thanks to OCM, it uses a hub-spoke architecture for the decentralized distribution of resources and coordination, also supporting execution and governance that makes a system scalable, stable, and edge-friendly.

The subcomponents of the MO's architecture are the MO API, the asynchronous MO Agent, and the Deployment Controller (DC). The MO API facilitates smart workload placement based on key metrics (CPU, memory, green energy usage, and cost) and offers a secure orchestration compatible with a new safety authentication.

The MO as a service is available with integrations such as cross-cluster networking (mNCC), workload cluster migrations (IBMC), and secure use of Unikernel deployments (SEE) to isolate crucial parts of NEMO, which highlights the MO is ready to provide its services to different NEMO components and in extension to different NEMO uses cases.

6 Secure Firmware Management on Far-Edge

In addition to the secure execution environment for microservices discussed in Section 2, the Smart Grid use case extends the requirements to include firmware updates for far-edge devices. These devices operate remotely, away from the data centre, and connect to the system solely through wireless technologies. To address this, a dedicated firmware update system, known as FOTA (Firmware Over-The-Air), has been integrated into NEMO using meta-Orchestrator calls. This section presents the details of the FOTA system and its integration.

6.1 Nerves architecture for FOTA system

D3.2 introduced the Firmware Over-The-Air (FOTA) system for distributing and deploying firmware images to far-edge devices. These devices are often deployed in harsh environments with limited connectivity options, typically relying on public wireless networks such as LTE. This highlights the critical need for secure and atomic firmware upgrades. If an update fails, devices could become inoperable (bricked) and require manual intervention.

Within the NEMO project, far-edge devices function as Phasor Measurement Units (PMUs), which play a critical role in fault localization for the Grid Disturbance Mitigation System, as detailed in D5.3 [6] and D5.4 [7] . These devices are responsible for collecting high-frequency phase readouts, preprocessing the data, and transmitting both alerts and readouts to the main cloud node, where they are available for further analysis.

6.1.1 The architecture of FOTA

The FOTA architecture, illustrated in Figure 74, distinguishes between components deployed within the NEMO infrastructure and those located at the far edge. The core FOTA system is integrated into the NEMO installation via an API, providing functionalities for core operations, status inspection, and firmware updates. The cloud-based FOTA system manages all firmware-related operations, including monitoring the status of field devices, logging changes, and maintaining an artifact repository of all available firmware versions.

At the edge, the system employs a parallel firmware partitioning approach, where one partition remains active while the other is prepared for deployment. This configuration enables a blue-green deployment strategy, reducing the risk of failures and ensuring seamless firmware updates with minimal downtime.

Communication between the cloud and edge components occurs over an LTE wireless network. The control plane is reserved for NEMO instructions and firmware updates, while the data plane handles the collection and inspection of PMU data. A dedicated user interface (GUI) is planned for this system, though it is not included in the current schema. Further details on this implementation will be provided in D5.4 (*NEMO Living Labs Use Case Evaluation – Final Version*).

Due to the importance and security, the communication between cloud and edge part is decoupled from the common NEMO services. The architecture relies on decoupled MQTT broker, MinIO [41]storage and PostgreSQL [42]database. This approach allows us to easily maintain the system, scale the components by demand, and control the throughput. Additionally, decoupling improves the security of the NEMO infrastructure. Note that far edge devices are on public networks and use SIM cards of public network providers. This expands the surface of possible attacks as devices at the edge could be stolen, compromised or the card identities would be spoofed, which could lead to potential DDoS attacks to the system. The possibility of this event is low, but even in case of happening, the main NEMO services as brokers and storages would not be affected.

[41] MinIO: https://min.io/docs/minio/kubernetes/upstream/index.html?ref=docs-redirect

[42] PostgreSQL: https://www.postgresql.org/

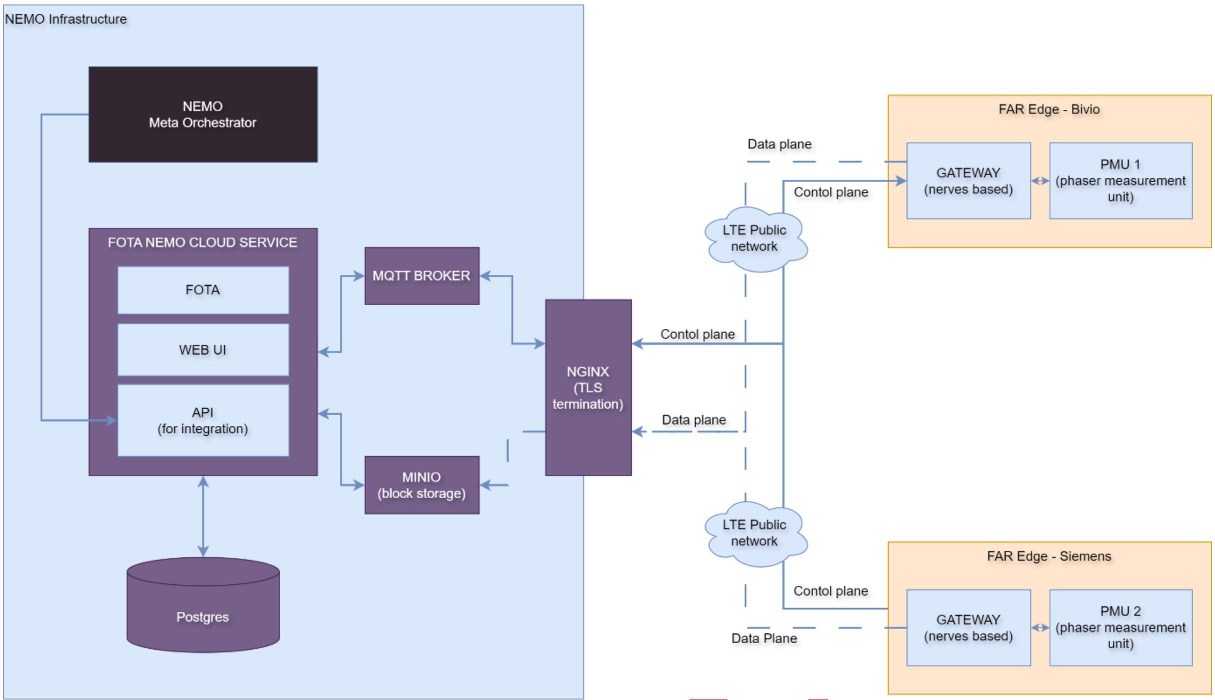


Figure 74: The schema of FOTA system.

6.1.2 Security and System Isolation

Due to security considerations, communication between cloud and edge components is isolated from other NEMO services. The system relies on a dedicated MQTT broker, MinIO for storage, and a PostgreSQL database, ensuring greater control over scalability, stability, and security. By decoupling these services, the system can be maintained and expanded more efficiently while also reducing potential attack vectors.

Since far-edge devices operate on public networks and use Subscriber Identity Module (SIM) cards from commercial providers, they present potential security risks, such as device theft, compromise, or SIM identity spoofing. Such incidents could lead to Distributed Denial of Service (DDoS) attacks on the system. Although the likelihood of these events is low, the main NEMO services, including brokers and storage, remain unaffected due to the segmented system design, ensuring continued stability and security.

6.1.3 Firmware update sequence

This sequence diagram, Figure 75 ,describes the FOTA update process within the NEMO infrastructure, ensuring secure firmware deployment to far-edge devices. The process begins with a new firmware version being stored in the FOTA NEMO Cloud Service. The NEMO meta-Orchestrator then requests a firmware update, prompting the FOTA system to securely transmit the update to both Bivio and Siemens Far Edge Gateways via the control plane.

Once the gateways receive the update, they confirm delivery by notifying the MQTT Broker. Following this, the FOTA system publishes an update message to the MQTT Broker, which then forwards the update request to the respective gateways. Each gateway then applies the firmware update locally and sends an update confirmation back to the MQTT Broker.

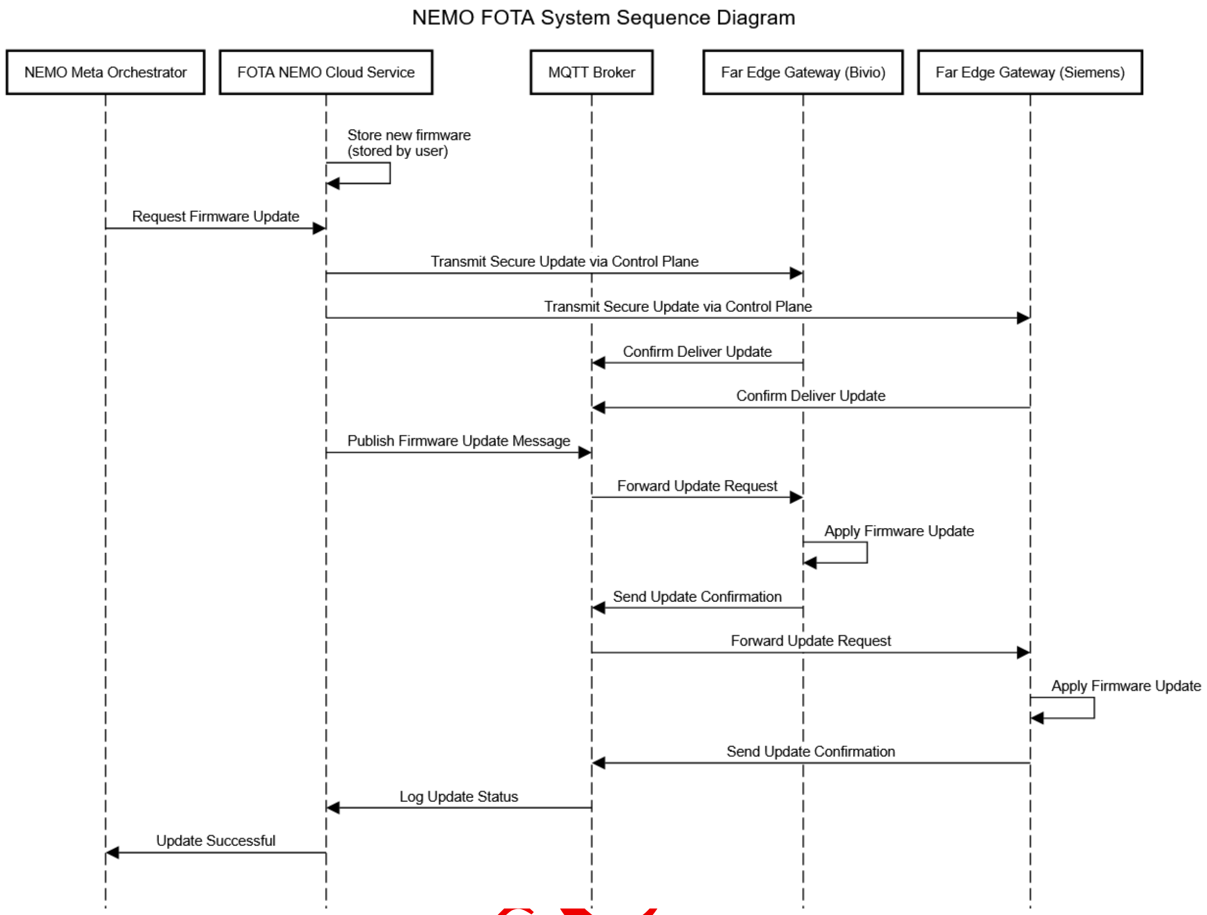


Figure 75: NEMO FOTA System Sequence Diagram.

The broker logs the update status and reports it to the FOTA system, which ultimately informs the NEMO meta-Orchestrator that the firmware update has been successfully completed. This structured process ensures secure and reliable firmware updates while maintaining clear communication between cloud services and far-edge devices.

6.1.4 FOTA PMU Cloud Service API documentation

This section is a result of the T3.1 task and provides the crucial information for the integration of FOTA management into NEMO framework. This API provides operations to interact with devices, including retrieving device information, fetching the last recorded data, updating firmware, and listing available devices.

Base url

/api/v1

GET /devices

Return a list of all available devices. For each available device it returns all data that is saved for each device. It also returns the current firmware version, and all data for all phasors.

GET /{device_id}/info

Returns a list of device's information for specific device. Return all the information that is available for specific device. It also returns data for all phasors that are connected to device.

GET /{device_id}/data

- device_id - integer

Return data for all phasors for given device. For each phasor, there is a field name, unit, angle, and magnitude. If device do not have available phasors the empty JSON is returned.

GET /firmware

Returns the list of available firmware files on the cloud service. The filenames are also the key IDs for updating firmware using the POST {device}/firmware command.

POST {device_id}/firmware

- PATH: device_id - integer
- BODY: { "filename": name of firmware file - string}

POST request for updating a specific device. In body of request filename is specified and based on that, correct firmware is flashed on device.

7 Measurement and Validation

This section takes a closer look at the main KPIs set for the NEMO project and the progress made to date; each KPI has assigned specific goals, whether it is boosting performance, improving security, streamlining integration, or enhancing functionality, finalizing the following breakdown shows what has been accomplished so far, and how each component contributes to validating NEMO's success.

7.1 Micro-Services Secure Execution Environment KPIs

Table 10: Micro-Service Secure Execution Environment KPIs

KPI #	Title
KPI4.1	Micro-services SEE increases from TRL-4 to TRL-5 by M24 (D3.2a) and to TRL-6 by M31 (D3.2b)
KPI4.2	Interface/Federate of at least 2 open-source containers' platforms and unikernels.
KPI4.3	Support at least 3 different native OS, e.g. Android, ROS and Linux

Following Table 10:

KPI 4.1 is about the TRL of the SEE interface. The SEE was successfully deployed and tested in the OneLab cluster, and the integration with the meta-Orchestrator asserted that this KPI could be accomplished.

KPI 4.2 demands the interaction of two open-source container platforms and Unikernels. The NEMO project has demonstrated the interaction of two Hermit unikernels on the Kubernetes Infrastructure of the OneLab Cluster, as well as on a local unikernel-specific Runtimes. We assert this KPI being successfully accomplished as well.

KPI 4.3 asserts the flexibility of the developed solution. Supporting HermitOS, as well as many other OS, such as Ubuntu, Alpine or Debian, this KPI can be considered accomplished as well.

7.2 PRESS, Safety & Policy enforcement framework KPIs

Table 11: PRESS, Safety & Policy enforcement framework KPIs

KPI #	Title
KPI5.2	Define > 20 SLO/KPIs (including CO2 footprint) for micro-services offloading decision by M12
KPI5.5	Reduce energy consumption >15%. In case of RES usage at the edge, CO2 footprint reduction >40%
KPI8.3	Micro-services policy & PRESS enforcement framework initial by M24(D3.2) and final by M31 (D3.3)

Following Table 11:

KPI 5.2 demands the definition of more than 20 SLO/KPIs for micro-services offloading decision making process. The NEMO meta-OS digests and considers more than 20 SLOs and KPIs in the form of either intents or KPIs that drive the orchestration of the NEMO-hosted workloads. These Intents or KPIs concern the NEMO-managed resource specifications, the NEMO-hosted workloads intents and the MOCA-managed metrics that concern the monetary aspects of a cluster and a service/application.

KPI 5.5 concerns the reduction of energy consumption by 15% and the CO2 footprint reduction by 40% for the case that RES usage is available. Both objectives are fulfilled through the optimal scheduling and management of the deployed applications and services (workloads) thought the NEMO meta-OS framework. More specifically, NEMO applies scaling on deployed workloads and on the corresponding pods only when it is necessary. This process is triggered once the target values that concern the workload performance capacity are reaching the boundaries set, thus guaranteeing the optimal energy management. Regarding the CO2 footprint reduction objective, the NEMO hosted resources (clusters and infrastructures) declare their power generation sources upon registration to the NEMO meta-OS framework. Then, the NEMO meta-OS is able to schedule (deploy and migrate) workloads to the resources that achieve the highest RES usage. This provides for significant RES reduction. Relevant validation activities will be performed in the context of NEMO Living Lab use cases.

KPI 8.3 The 1st release of the PPEF component was available by M24 and presented in detail in D3.2, whilst the final release of the PPEF component was available on M31 and described in D3.3. The final version component has been updated for performance, stability and enhanced its functionality through the monitoring of the GPU received metrics. It is deployed in all OneLab hosted NEMO environments and in NEMO pilots.

7.3 Cybersecurity & Digital Identity Attestation KPIs

The modules comprising the Cybersecurity & Digital Identity Attestation sub-system of NEMO contribute to the materialization of the following two KPIs, Table 12:

Table 12: Cybersecurity & DIA KPIs

KPI #	Title
KPI8.1	Cybersecure "by design" components (i.e. network zones, CF-DRL, SEE, CMDT, MOCA) ≥ 5.
KPI8.2	Supplementary cybersecurity methods & Digital Identity Attestation ≥ 6.

Security by Design incorporates a set of technical principles that aim to embed security controls and threat mitigation strategies directly into the architecture and codebase from the earliest design stages. In order for a component to be Cybersecure "by design" they should support one or more of the following features. A foundational concept is the Principle of Least Privilege (PoLp), which dictates that processes, services, and users should operate with only the permissions they need to function by reducing the attack surface and limiting the blast radius of a compromise. This is enforced through fine-grained access controls, privilege separation, and the use of secure tokens or scoped API keys. Defense in Depth (DiD) extends this approach by layering security mechanisms across multiple tiers such as input validation, access controls, encryption, monitoring, and anomaly detection so that failure in one layer does not lead to full system compromise. Systems are designed to fail securely, meaning exception handling and error states are coded to avoid exposing sensitive data, stack traces, or internal logic; defaults are set to deny access unless explicitly permitted. Secure defaults ensure that all deployments begin with hardened configurations like disabled debug modes, strong password policies, and TLS enabled by default minimizing risk from misconfiguration. During the design phase, threat modelling is conducted to identify potential attack vectors, using methodologies like STRIDE or DFDs (Data Flow Diagrams) to systematically analyze data paths and trust boundaries. Identified threats are mitigated with specific controls such as input sanitization, rate limiting, or authentication checks. Throughout the development lifecycle, continuous testing and validation are performed via automated static and dynamic analysis, dependency scanning (e.g., Snyk, OWASP Dependency-Check), fuzz testing, and regular penetration testing. Security findings feed back into the CI/CD pipeline, ensuring secure code is maintained across iterations. By implementing these technical practices consistently, systems achieve a resilient, security-first posture that can withstand real-world adversaries.

In that respect, in terms of **KPI8.1** the intercommunication module is cybersecure by design since it fully supports secure defaults and defence in depth through the relevant modes and layers of TLS, The Identity management system has been developed from scratch so as to be full in line with the defence in Depth approach, by layering several mechanisms for access controls, encryption, monitoring. The CNAPP - Cloud-Native Application Protection Platform developed is perfectly inline, by design, with continuous testing and validation testing throughout the development and operation phases. As a result, the modules developed as part of the Cybersecurity & Digital Identity Attestation sub-system have three distinct components that contribute to KPI8.1.

Moreover, a number of additional such components which support the security “by design “principle will be listed in the final deliverable of WP4.

In terms of **KPI8.2** Modern cybersecurity methods leverage a layered and adaptive approach, integrating preventive, detective, and responsive controls to protect systems against evolving threats. Core techniques include network segmentation, zero trust architecture, multi-factor authentication (MFA), behavioral analytics, and endpoint detection and response (EDR). Cryptographic protocols such as TLS 1.3, mutual TLS, and elliptic curve cryptography (ECC) are employed to ensure secure communication and data integrity, while tokenization, hardware security modules (HSMs), and secure enclaves are used to protect sensitive assets at rest and in use. Within this context, Digital Identity Attestation systems play a critical role in verifying the authenticity and integrity of user and device identities. These systems often rely on Public Key Infrastructure (PKI), biometric verification, verifiable credentials (VCs), and decentralized identifiers (DIDs) to establish trust in a cryptographically secure and privacy-preserving manner. Identity proofs may include signed assertions from trusted authorities, leveraging standards such as OAuth 2.0, OpenID Connect, FIDO2/WebAuthn, and W3C Verifiable Credentials, with attestation mechanisms built to detect device spoofing, tampering, or replay attacks.

Across those lines the developed Cybersecurity & Digital Identity Attestation sub-system of NEMO includes the full TLS cybersecurity components and supports an OAuth2.0 compatible authentication mechanism while the Digital Identity Management module is a full Digital Identity Attestation system complying with the OAuth2.0 standard. As a result, the modules developed as part of the Cybersecurity & Digital Identity Attestation sub-system support one full set and cybersecurity methods as well as two Identity Attestation systems thus contributing to KPI8.2 with three elements.

Moreover, three additional cybersecurity methods are employed in the CFDRL sub-system developed in WP2 for increasing the resistance to cyberattacks to distributed AI systems thus contributing with three more elements to KPI8.2 More elements of KPI8.2 will be listed in the final deliverable of WP4.

7.4 NEMO meta-Orchestrator KPIs

Table 13: MO KPIs

KPI #	Title
KPI5.1	Interface >3 local micro-services schedulers/ Orchestrators.
KPI5.3	Micro-services discovery 10k; Simulated repositories> 100.
KPI5.4	Low latency dynamic migration decision (<1 sec), zero-downtime service reschedules (Blue-Green) (<10 ms).

Following Table 13:

KPI5.1: The main solution that allows NEMO to orchestrate multiple microservices is Kubernetes (K8s), the meta-Orchestrator relies on OCM which supports the entire lifecycle of a K8s cluster, and it is used to provide multi-cluster orchestration across diverse computational environments. In the initial phase of the NEMO project the main K8s distribution tested where K8s, K3s, Kind and OpenShift nevertheless other distributions are also supported by this solution (Amazon EKS, Google GKE, Azure AKS, among others).

Through the SEE, NEMO is able to address secure and lightweight deployments on the Hermit operating system which is a rust-based, lightweight unikernel.

Specialized deployments on embedded systems through BEAM virtual machines are supported in NEMO thanks to FOTA component.

KPI:5.3: The programmatic entry path for NEMO platform is the Intent-API SDK. The service application K8s descriptors can be package as Helm charts that are processed by the Intent-API to check its syntax and break it down into custom resource definitions that our meta-Orchestrator is able to process across the different target clusters managed by the NEMO platform. There are thousands of Helm charts available, most of them in public repositories that include a large catalog of pre-packaged K8s applications. Furthermore, Helm charts can be based on docker public images hosted on public repositories like DockerHub which extend further the possibility to package our services based on container public images.

KPI5.4: The dynamic migration decision may vary depending on the constraints defined for a deployed service, these constraints are assessed by the PPEF and evaluated by the CFDRL module in order to communicate to the NEMO kernel which is the best placement or rescheduling action (scaling/migrate) for a concrete workload, taking into consideration the overall conditions of the managed clusters.

When it is requested the migration action is triggered by the IMC component in charge of backing up the resources and persistent volumes and restore them to a different target cluster with zero-downtime from users' perspective.

8 Conclusions

This deliverable D3.3 is the culmination of the work done in Work Package 3 (WP3) of the NEMO project, including all the technical effort, evaluations, and integrations made over the last months and transformed into a complete and mature version of the NEMO Kernel. The four core components (SEE, PPEF, Cybersecurity & Digital Identity Attestation, and NEMO meta-Orchestrator) have matured to a stable and working state.

As development progressed, efforts were established not only to implement advanced features but also to integrate each component into the NEMO ecosystem. The SEE component can manage lightweight unikernels and migration in Kubernetes nodes, allowing NEMO to achieve performance objectives in distributed edge-to-cloud systems, decreasing time and memory usage and making them ideal for edge environments.

The PPEF component has matured into key monitoring tools for services and now works with well-defined intents, promoting better resource utilization and compliance with service level targets, its tools can collect metrics, analyze them, and pipeline insights into other components and NEMO services, such as the meta-Orchestrator and the CFDRL, acting as the learning system.

The Cybersecurity and Digital Attestation component, working aligned with its modules, now provides an integral layer of trust, with secure access control and runtime security checks. In addition, this component incorporates Cloud-Native Application Protection Platform (CNAPP) principles to secure applications throughout the full software lifecycle.

Now, there is the meta-Orchestrator, which binds it all together, becoming a microservice for workload management, scaling decisions, and component communication. As such, it will use metrics from the other modules and apply intelligent control to keep the system operating optimally in a distributed environment.

Finally, the NEMO kernel has evolved and been tested with the help of the NEMO components, making a secure, extensible, and efficient system possible. The base laid is robust and sufficiently generic to accommodate use cases, pilots, and integrations yet to come outside of WP3 scope.

9 References

[1] NEMO D3.1 - Introducing NEMO Kernel. Lead Participant: RWTH. HORIZON - 101070118 - NEMO Deliverable Report, 2023.

[2] NEMO D3.2 - Nemo Kernel Initial Version. Lead Participant: ATOS. HORIZON - 101070118 - NEMO Deliverable Report, 2024.

[3] NEMO D2.3- Enhancing NEMO Underlying Technology. Lead Participant: TID. HORIZON - 101070118 - NEMO Deliverable Report, 2024.

[4] NEMO D4.2 Advanced NEMO platform & laboratory testing results. Initial version. Lead Participant: INTRA. HORIZON - 101070118 - NEMO Deliverable Report, 2024.

[5] NEMO D4.3 Advanced NEMO platform & laboratory testing results. Final version. Lead Participant: INTRA. HORIZON - 101070118 - NEMO Deliverable Report, 2025 (Not submitted)

[6] NEMO D5.3 NEMO Living Labs use cases evaluation results - Initial version. Lead Participant: ASM. HORIZON - 101070118 - NEMO Deliverable Report, 2025.

[7] NEMO D5.4 Living Labs use cases evaluation results. Final version. Lead Participant: ASM. HORIZON - 101070118 - NEMO Deliverable Report, 2025 (Not submitted)

10 Annexes

10.1 Guidelines for TDX and Confidential Containers Technology

First, the server should have enough hardware capabilities to enable TDX. To do so, it has been used this guide 6, the beginning starts with the following commands as can be observed in Figure 76, and Figure 78:

```
git clone -b main https://github.com/canonical/tdx.git
cd tdx
sudo ./setup-tdx-host.sh
```

Figure 76: Enable Intel TDX in Host OS

After that, reboot the machine.

10.1.1 Enable the TDX in the BIOS

To enable the BIOS in a proper way it is necessary to follow the next steps, as is represented in Figure 77:

Required BIOS Settings for Intel TDX:

- Memory Settings:
    - Disable Node Interleaving
- Processor Settings:
    - Enable x2APIC Mode
    - Disable CPU Physical Address Limit
- System Security:
    - Set Memory Encryption to Multiple Keys.
    - Disable Global Memory Integrity.
    - Enable Intel Trusted Domain Extension (TDX).
    - Set TME-MT/TDX Key Split to a non-zero value (such as, 1)
    - Enable TDX Secure Arbitration Mode Loader (SEAM).
    - Enable Intel(R) SGX.

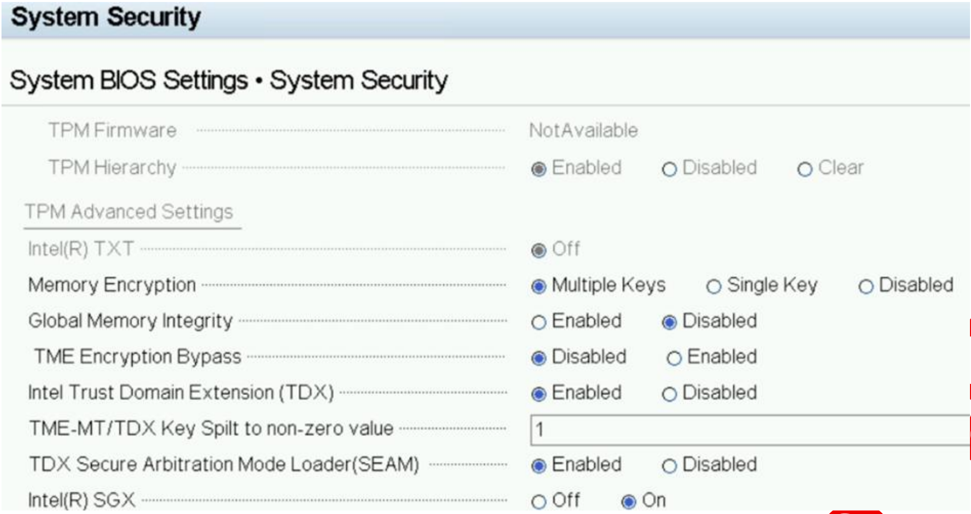


Figure 77: System BIOS settings.

```
sudo dmesg | grep -i tdx
```

Figure 78: Verify Intel TDX is Enabled on Host OS

If you have "virt/tdx: module initialized" as the output of the message means that TDX has initialized properly.

TDX in the server has been enabled; to give it a try deploying a TD and enable the remote attestation to follow the steps 6, 7 and 8 of this guide [2].

Install Confidential Containers

Deploy the operator by running the following command (we are using the latest version, which is v0.12.0) like in Figure 79:

```
kubectl apply -k github.com/confidential-containers/operator/
config/release?ref=v0.12.0
```

Figure 79: Operator deployment.

Wait until the pod has the STATUS "Running" like in Figure 80:

```
kubectl get pods -n confidential-containers-system --watch
```

Figure 80: wait for "Running" status

Now, proceed with deploying the CC Runtime, responsible for creating the necessary runtimes for the deployment, Figure 81:

```
kubectl apply -k github.com/confidential-containers/operator/config/
samples/ccruntime/default?ref=v0.12.0
```

Figure 81: CC Runtime deployment

Wait until the pod has the STATUS "Running" as like in Figure 82:

```
kubectl get pods -n confidential-containers-system --watch
```

Figure 82: wait for "Running status"

To verify that everything has been installed correctly like Figure 83 below:

```
kubectl get runtimeclass
```

Figure 83: Kubectl get runtimeclass

The output should be Table 14:

Table 14: Kata container installation.

Name	Handler	Age
kata	kata-qemu	37h
kata-clh	kata-clh	37h
kata-qemu	kata-qemu	37h
kata-qemu-sev	kata-qemu-coco-dev	37h
kata-qemu-sev	kata-qemu-sev	37h
kata-qemu-snp	kata-qemu-snp	37h
kata-qemu-tdx	kata-qemu-tdx	37h

Deployment of the Pod in CoCo by Encrypting and Signing the Image

```
# Clone KBS git repository
git clone https://github.com/confidential-containers/trustee.git
cd trustee/kbs
export KBS_DIR_PATH=$(pwd)

# Generate a user auth key pair
openssl genpkey -algorithm ed25519 > config/private.key
openssl pkey -in config/private.key -pubout -out config/public.pub

cd ..

# Start KBS cluster
docker-compose up -d
```

Figure 84: CoCo deployment

Encrypting the Image

To encrypt the image, we use skopeo. To install it, follow these instructions [3]. You must have at least version 1.16.0 of skopeo[43]. For this example, the image busybox:latest has been used, but any image can be used, as can be observed in Figure 85.

[43] https://github.com/containers/skopeo/blob/main/install.md

```
# edit ocicrypt.conf
tee > ocicrypt.conf <<EOF
{
    "key-providers": {
        "attestation-agent": {
            "grpc": "127.0.0.1:50000"
        }
    }
}
EOF

# encrypt the image
OCICRYPT_KEYPROVIDER_CONFIG=ocicrypt.conf skopeo copy --insecure-policy --
encryption-key provider:attestation-agent docker://library/busybox
oci:busybox:encrypted
```

Figure 85: Image encryption.

With this last command, several things happen inside the cluster:

- The CoCo Keyprovider generates a random key and a key identifier. Then, it encrypts the image using this key.
- The CoCo Keyprovider registers the key with the key identifier in the KBS.

Now, upload the image, Figure 86:

```
skopeo copy oci:busybox:encrypted [SCHEME]://[REGISTRY_URL]:encrypted
```

Figure 86: Upload image.

In our case, Figure 87:

```
cosign sign --key cosign.key docker.io/jorgealmansa/busybox:encrypted
```

Figure 87: Image Signing.

Next, edit an image pull validation policy file.
The file is called security-policy.json:

```
{
    "default": [{"type": "reject"}],
    "transports": {
        "docker": {
            "[REGISTRY_URL]": [
                {
                    "type": "sigstoreSigned",
                    "keyPath": "kbs:///default/cosign-key/1"
                }
            ]
        }
    }
}
```


Figure 88: Security policy

You need to replace [REGISTRY_URL] with docker.io/jorgealmansa/busybox:encrypted in our case.

Register security-policy.json in the KBS storage:

```
mkdir -p $KBS_DIR_PATH/data/kbs-storage/default/security-policy
cp        security-policy.json        $KBS_DIR_PATH/data/kbs-storage/default/
security-policy/test
```

Figure 89: Register security policy in KBS storage.

Deploying an Encrypted Image Using CoCo on CC HW

This is an example YAML file for deploying encrypted images:

```
cat << EOT | tee encrypted-image-test-busybox.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: encrypted-image-test-busybox
  name: encrypted-image-test-busybox
  annotations:
    io.containerd.cri.runtime-handler: [RUNTIME_CLASS]
spec:
  containers:
  - image: [REGISTRY_URL]:encrypted
    name: busybox
  dnsPolicy: ClusterFirst
  runtimeClassName: [RUNTIME_CLASS]
EOT
```

Figure 90: Deploying encrypted images.

In our case, we replace [RUNTIME_CLASS] with kata-qemu-tdx and [REGISTRY_URL] with docker.io/jorgealmansa/busybox.

Finally, the IP of the KBS service must be configured in the file */opt/kata/share/defaults/kata-containers/configuration-qemu-tdx.toml*.

To do this, perform a *docker network inspect* of the KBS cluster to see the IPs of each container. Then, modify the *kernel_params* line so that it contains *agent.aa_kbc_params=cc_kbc::<KBS_URI>*, for
"agent.aa_kbc_params=cc_kbc::http://172.19.0.1:8080" (for example).

If you encounter the error *tee_qv_get_collateral failed: 0xe019*, it is due to a network issue, meaning that your AS cannot connect to the local PCCS.

There are two ways to resolve this:

- If you do not have the PCCS service installed, use the following line in */config/sgx_default_qcnl.conf*:

```
{"collateral_service":          "https://api.trustedservices.intel.com/sgx/
certification/v4/"}
```

Figure 91: Resolving PCCS fail.

- If PCCS is installed (*sudo systemctl status pccs*), you should use your machine's IP in the file */config/sgx_default_qcnl.conf*, since the AS container must connect to that IP, Figure 92:

```
{
  // *** ATTENTION : This file is in JSON format so the keys are case sensitive. Don't
change them.

  //PCCS server address
  "pccs_url": "https://<IP-SERVER>:8081/sgx/certification/v4/"

  // To accept insecure HTTPS certificate, set this option to false
  ,"use_secure_cert": false

  // You can use the Intel PCS or another PCCS to get quote verification collateral.  Retrieval of PCK
  // Certificates will always use the PCCS described in pccs_url.  When collateral_service is not defined, both
  // PCK Certs and verification collateral will be retrieved using pccs_url
  //,"collateral_service": "https://api.trustedservices.intel.com/sgx/certification/v4/"

  // If you use a PCCS service to get the quote verification collateral, you can specify which PCCS API version is to be used.
  // The legacy 3.0 API will return CRLs in HEX encoded DER format and the sgx_ql_qve_collateral_t.version will be set to 3.0, while
  // the new 3.1 API will return raw DER format and the sgx_ql_qve_collateral_t.version will be set to 3.1. The pccs_api_version
  // setting is ignored if collateral_service is set to the Intel PCS. In this case, the pccs_api_version is forced to be 3.1
  // internally.  Currently, only values of 3.0 and 3.1 are valid.  Note, if you set this to 3.1, the PCCS use to retrieve
  // verification collateral must support the new 3.1 APIs.
  //,"pccs_api_version": "3.1"

  // Maximum retry times for QCNL. If RETRY is not defined or set to 0, no retry will be performed.
  // It will first wait one second and then for all forthcoming retries it will double the waiting time.
  // By using retry_delay you disable this exponential backoff algorithm
  ,"retry_times": 6

  // Sleep this amount of seconds before each retry when a transfer has failed with a transient error
  ,"retry_delay": 10

  // If local_pck_url is defined, the QCNL will try to retrieve PCK cert chain from local_pck_url first,
  // and failover to pccs_url as in legacy mode.
```

```
  //,"local_pck_url": "http://localhost:8081/sgx/certification/v4/"

  // If local_pck_url is not defined, set pck_cache_expire_hours to a none-zero value will enable local cache.
  // The PCK certificates will be cached in memory and then to the disk drive.
  // ===== Important: Once the local cache files are created, currently there is no other way to clean them other
  //                   than to delete them manually, or wait for them to expire after "pck_cache_expire_hours" hours.
  //                   To delete the cache files manually, go to these foders:
  //                            Linux : $AZDCAP_CACHE, $XDG_CACHE_HOME, $HOME, $TMPDIR, /tmp/
  //                        Windows : $AZDCAP_CACHE, $LOCALAPPDATA\..\..\LocalLow
  //                         If there is a folder called .dcap-qcnl, delete it. Restart the service after all cache
  //                           folders were deleted. The same method applies to "verify_collateral_cache_expire_hours"
  ,"pck_cache_expire_hours": 168

  // To set cache expire time for quote verification collateral in hours
  // See the above comment for pck_cache_expire_hours for more information on the local cache.
  ,"verify_collateral_cache_expire_hours": 168

  // You can add custom request headers and parameters to the get certificate API.
  // But the default PCCS implementation just ignores them.
  //,"custom_request_options" : {
  //  "get_cert" : {
  //    "headers": {
  //      "head1": "value1"
  //    },
  //    "params": {
  //      "param1": "value1",
  //      "param2": "value2"
  //    }
  //  }
  //}
}
```

Figure 92: PCCS config file.