NEMO

Next Generation Meta Operating System

D4.2 Advanced NEMO platform & laboratory testing results. Initial version

Document Identification			
Status	Final	Due Date	30/11/2024
Version	1.0	Submission Date	17/12/2024

Related WP	WP4	Document Reference	D4.2
Related Deliverable(s)	D1.1, D1.2, D1.3, D2.2, D3.2, D4.1	Dissemination Level (*)	PU
Lead Participant	INTRA	Lead Author	Dimitrios Skias (INTRA)
Contributors	SYN, INTRA, AEGIS, SPACE, ATOS, MAG, ENG, ESOFT, SU	Reviewers	Panagiotis Karkazis (MAG)
			Ignacio Prusiel (ATOS)

Keywords:

Integration, Validation, API, SDK, Lifecycle Management, Migration Controller, Automation

(*) 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
Enric Pere Pages Montanera	ATOS
Rubén Ramiro	ATOS
Ignacio Prusiel	ATOS
Matija Cankar	COM
Dimitrios Skias	INTRA
Panagiotis Karkazis	MAG
Astik Samal	MAG
Nikos Drosos	SPACE
Emmanouil Bakiris	SPACE
Antonis Gonos	ESOFT
Theodore Zahariadis	SYN
Terpsi Velivassaki	SYN
Spyros Vantolas	AEGIS
Hassane Rahich	SU

Document History			
Version	Date	Change editors	Changes
0.1	12/09/2024	INTRA	ToC
0.2	03/10/2024	INTRA	Updates in ToC and initial input
0.3	18/10/2024	INTRA	Updates in section 1,2 and 4
0.4	25/10/2024	SYN, ATOS, AEGIS, ESOFT	Updates in section 3 and 4
0.5	08/11/2024	INTRA, MAG	Updates in section 1 and 2
0.6	22/11/2024	SYN, ATOS, AEGIS, ESOFT	Updates in section 3
0.7	29/11/2024	INTRA, ATOS, AEGIS, SYN, COM	Updates in section 4, conclusions and introduction of Annex A & B
0.8	6/12/2024	INTRA	Document consolidation; Peer-review ready version
0.9	13/12/2024	INTRA, MAG, ATOS	Document consolidation; Peer-review comments addressed
0.91	17/12/2024	INTRA	Final version ready
1.0	17/12/2024	ATOS	Format review and submission to EC

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	D. Skias (INTRA)	17/12/2024
Quality manager	R. Valle Soriano (ATOS)	17/12/2024
Project Coordinator	E. Pages (ATOS)	17/12/2024

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
AAA	Authentication, Authorization, and Accounting
AI	Artificial Intelligence
API	Application Programming Interface
CD	Continuous Delivery
CFDRL	Cybersecure Federated Deep Reinforcement Learning
CLI	Command Line Interface
CMDT	Cybersecure Microservices' Digital Twin
CI	Continuous Integration
CLI	Command-line Interface
CMDT	Cybersecure Microservices' Digital Twin
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRD	Custom Resource Definition
DApps	Distributed Applications
DLT	Distributed Ledger Technology
Dx.y	Deliverable number y belonging to WP x
E2E	End-to-End
EC	European Commission
FL	Federated Learning
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
IBMC	Intent-based Migration Controller
IdM	Identity Management
IDS	Intrusion Detection System
IPFS	Interplanetary File System
IoT	Internet-of-Things
IT	Information Technology
K8s	Kubernetes
LAN	Local Area Network
LCM	Life-Cycle Manager
meta-OS	Meta-Operating System
ML	Machine Learning
mNCC	Meta Network Cluster Controller
MO	Meta-Orchestrator
MOCA	Monetization and Consensus-based Accountability
MQTT	Message Queuing Telemetry Transport
NAC	NEMO Access Control
OS	Operating System
PPEF	PRESS & Policy Enforcement Framework

RAM	Random Access Memory
RBAC	Role-Based Access Control
RL	Reinforcement Learning
SDK	Software Development Kit
SEE	Secure Execution Environment
TRL	Technology Readiness Level
V&V	Validation & Verification
WAL	Write-Ahead Logging
WP	Work Package
YAML	Yet Another Markup Language

Executive Summary

The document presents insights into the first integrated version of the NEMO meta-OS consisting of the core components, the interfaces and the integration activities to this point. The work also involves the creation of integration scenarios that guided the integration tests conducted in a laboratory setting, utilizing the supporting CI/CD environment and tools and verified the system-level technical capacity of the platform. Moreover, the deliverable provides a detailed presentation of the technical developments conducted within WP4, detailing the NEMO meta-OS Service Management Layer's technical advancements and updates, including their associated interactions within NEMO meta-OS. The final version of the NEMO meta-OS integrated platform is expected to be presented in D4.3 [1], "Advanced NEMO Platform & Laboratory Testing Results. Final Version," which will be produced in the second quarter of 2025.

1 Introduction

The NEMO meta-OS framework aims to support optimal operation of hyper-distributed applications implemented as microservices in a highly distributed and diverse environment of cloud-edge and IoT technologies. Therefore, the associated integration and subsequently the verification and validation of such activities are not trivial and require a well-designed methodology and execution plan.

In D4.1 [2], a detailed description of the zero-ops CI/CD environment and the associated integration guidelines were discussed. This document aims to shed light on the scenario-driven integration and validation activities that aim to realise the first integrated version of the NEMO meta-OS as dictated by the Validation & Verification methodology that is presented. The main objective of this approach is on one hand to assure that NEMO specifications on interfaces and data models are coherent and followed by the stemming technical developments and on the on the other to facilitate and accelerated the necessary integration activities.

1.1 Purpose of the document

The purpose of this document is twofold. First, it presents the technical advancements that fall into the NEMO Service Management Layer and second to present in detail the integration activities that are driven by the NEMO Validation & Verification (V&V) methodology and document the associated results which led to the production of the first integrated NEMO meta-OS framework.

1.2 Relation to other project work

The integration and testing strategy act as the driver of the development process. Thus, this document is strongly connected with all the technical WPs (WP2, WP3 and WP4). Furthermore, the work presented in this document strongly relates to WP1 activities, as it considers the technical specifications arising from the requirements' elicitation process and the architectural specifications. In addition, the platform integrated view and current prototype implementations will be applied and tailored to each of the NEMO trials within WP5. Last, but not least, the document reports technical options and prototype functionalities, which are meant to be used and extended by third parties joining the project through the Open Calls.

1.3 Structure of the document

The remainder of this report is organized as follows.

Section 2 provides information on the CI/CD environment of the NEMO meta-OS and on the OneLab facilities that provide for the integration activities of the first integrated version of the NEMO meta-OS. In addition, it presents the high-level architecture view of the first integrated version of the NEMO meta-OS highlighting the key integration activities for each functional layer and describes the components that are fully or partially integrated.

Section 3 describes the overview, the architecture, the initial results and the interactions with other components for the modules that are comprising the Service Management Layer of the NEMO meta-OS platform, namely the intent-based Migration Controller (IBMC), the Plugin & Application Lifecycle Manager (LCM), the Monetization and Consensus based Accountability (MOCA) and the Intent-based SDK/API.

Section 4 sheds light into the integration activities that are conducted and materialized the first integrated version of the NEMO meta-OS, following the scenario-driven V&V methodology.

Section 5 provides conclusions and insights in view of the final version of the NEMO meta-OS.

Finally, Annex A & B provide a detailed description of the Intent-based API and MOCA interfaces and data models.

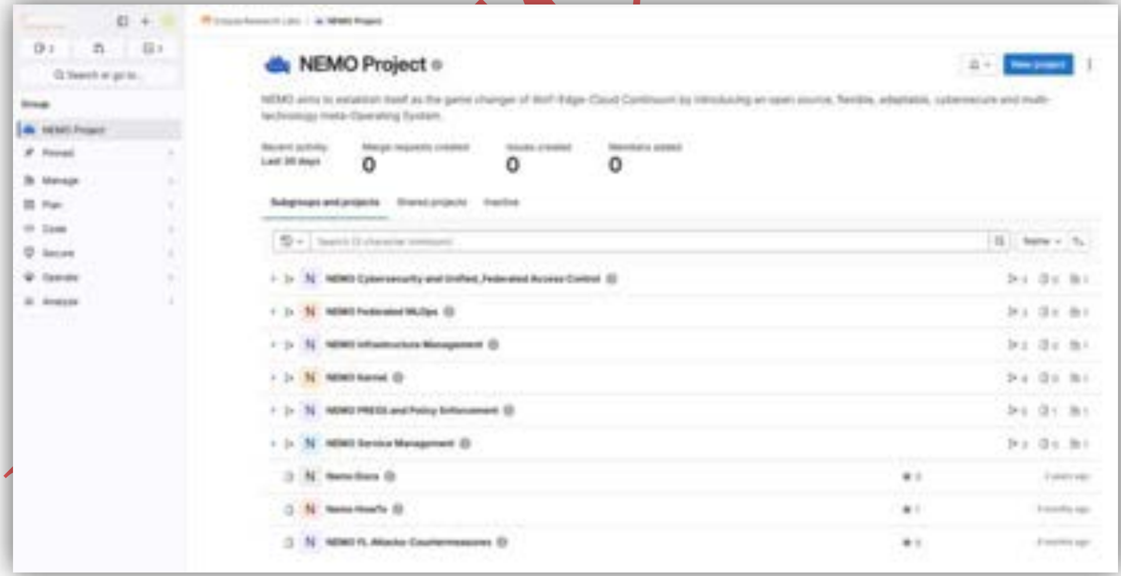
2 NEMO Integration, Validation & Verification approach and tools

The first integrated NEMO meta-OS version that is described in this document capitalizes on the NEMO CI/CD environment and tools. The associated CI/CD pipeline facilitates the agile integration and validation approach that the NEMO adopts and is applied throughout the cloud and edge infrastructure that is available and orchestrated by NEMO meta-OS. The NEMO meta-OS underlying infrastructure resides in OneLab facilities. The following sections shed light both on the source code repository configuration in Eclipse Gitlab and on the OneLab cloud and edge infrastructure that is provided to NEMO meta-OS. This environment is essential for conducting and subsequently demonstrating the integration and validation activities that resulted in the first integrated version of NEMO meta-OS.

2.1 NEMO CI/CD Environment & Tools

2.1.1 Open Source repository

For the NEMO project, the GitLab CI/CD framework has been set up and organized in an Eclipse Research Labs hosted instance of GitLab. The official GitLab group of NEMO is titled "NEMO Project" and is accessible publicly at https://gitlab.eclipse.org/eclipse-research-labs/nemo-project. The group hosts the source code that is related to each thematic entity-specific development as dictated by the NEMO meta-OS architecture. Each thematic entity is organized as a subgroup of the NEMO GitLab group, Figure 1.



NEMO Project

NEMO aims to establish itself as the game changer of IoT-Edge-Cloud Continuum by introducing an open source, flexible, adaptable, cybersecure and multi-technology meta-Operating System.

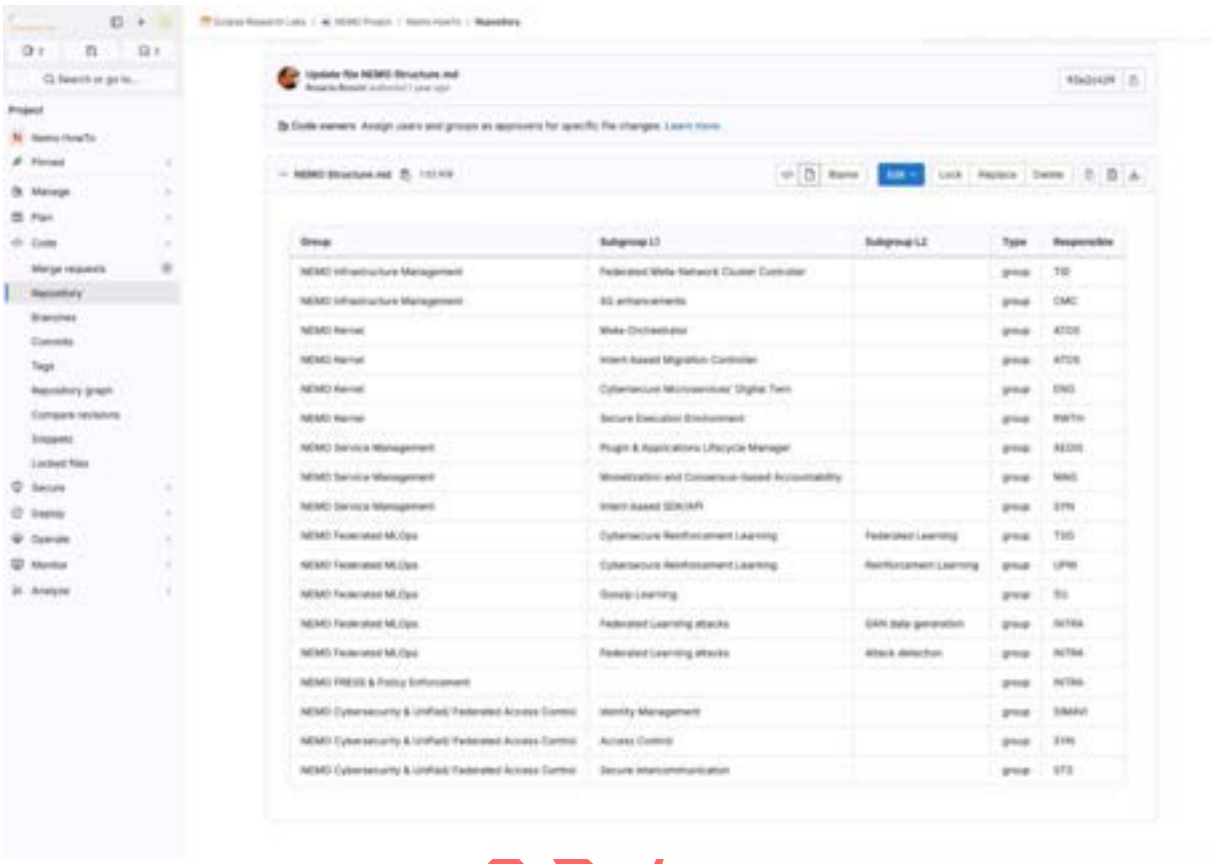
Recent activity Last 30 days	Merge requests created	Issues created	Members added
	0	0	0

Subgroups and projects | Shared projects | Inactive

- NEMO Cybersecurity and Unified_Federated Access Control
- NEMO Federated MLOps
- NEMO Infrastructure Management
- NEMO Kernel
- NEMO PRESS and Policy Enforcement
- NEMO Service Management
- Nemo-Docs
- Nemo-HowTo
- NEMO FL Attacks Countermeasures

Figure 1: NEMO code repository in Eclipse Research labs

Within each subgroup, the development activities are organized based on the implemented outcomes of the relevant tasks. Moreover, for each subgroup an owner is assigned as illustrated in the Figure 2 below.


Update file NEMO Structure.md

Code owners Assign users and groups as approvers for specific file changes. Learn more.

NEMO Structure.md

Group	Subgroup L1	Subgroup L2	Type	Responsible
NEMO Infrastructure Management	Federated Meta-Network Cluster Controller		group	TID
NEMO Infrastructure Management	5G enhancements		group	CMC
NEMO Kernel	Meta-Orchestrator		group	ATOS
NEMO Kernel	Intent-based Migration Controller		group	ATOS
NEMO Kernel	Cybersecure Microservices' Digital Twin		group	ENG
NEMO Kernel	Secure Execution Environment		group	RWTH
NEMO Service Management	Plugin & Applications Lifecycle Manager		group	AEGIS
NEMO Service Management	Monetization and Consensus-based Accountability		group	MAG
NEMO Service Management	Intent-based SDK/API		group	SYN
NEMO Federated MLOps	Cybersecure Reinforcement Learning	Federated Learning	group	TSG
NEMO Federated MLOps	Cybersecure Reinforcement Learning	Reinforcement Learning	group	UPM
NEMO Federated MLOps	Gossip Learning		group	SU
NEMO Federated MLOps	Federated Learning attacks	GAN data generation	group	INTRA
NEMO Federated MLOps	Federated Learning attacks	Attack detection	group	INTRA
NEMO PRESS & Policy Enforcement			group	INTRA
NEMO Cybersecurity & Unified/Federated Access Control	Identity Management		group	SIMAVI
NEMO Cybersecurity & Unified/Federated Access Control	Access Control		group	SYN
NEMO Cybersecurity & Unified/Federated Access Control	Secure Intercommunication		group	STS

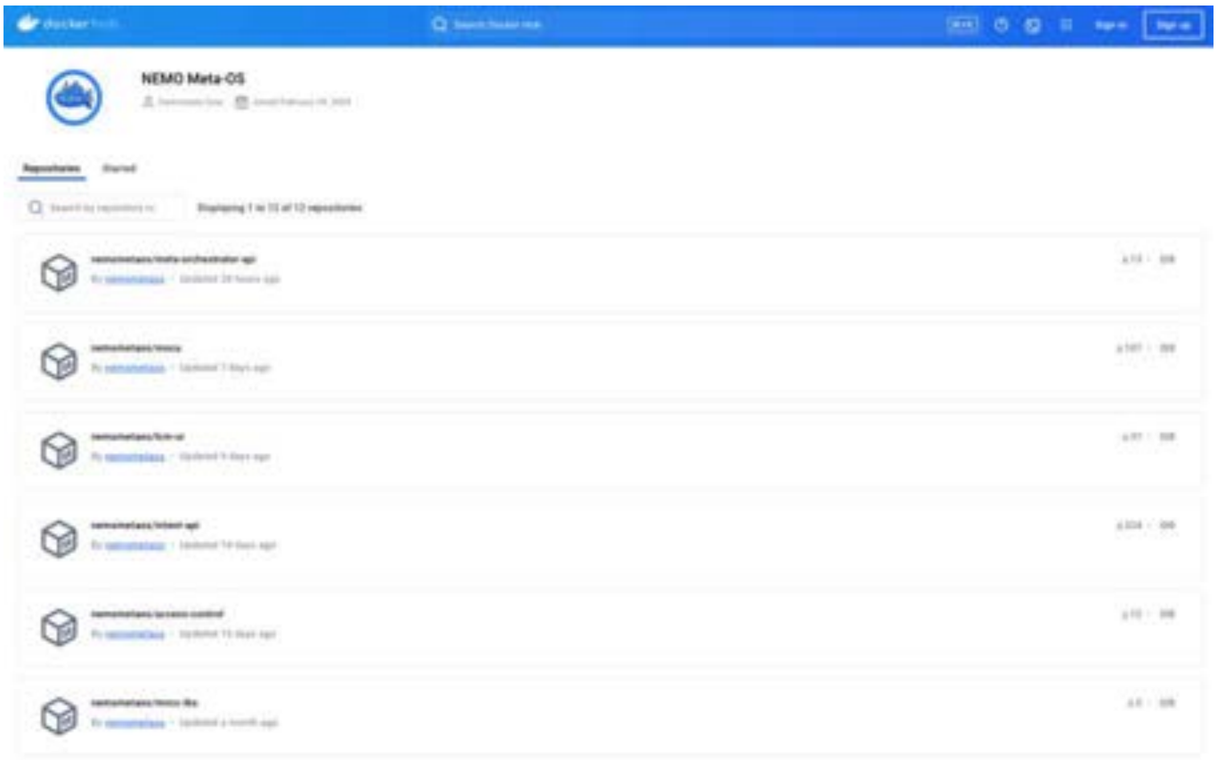
Figure 2: NEMO Structure and Ownership

2.1.2 NEMO Automated Deployment and Configuration

The NEMO CI/CD environment capitalizes on Kubernetes[1] manifests for the deployment of the NEMO components in the NEMO meta-OS infrastructure. The documentation that concerns the NEMO CI/CD integration steps that are necessary for the component deployment process is described in a readme file within the NEMO repository.

The NEMO components' container images are uploaded to the docker.io instance under the account of NEMO, that is following the naming convention *"nemometaos/Xcomponent_nameX"*. The docker[2] image repository (https://hub.docker.com/u/nemometaos) is presented in Figure 3.

[1] https://kubernetes.io/
[2] https://www.docker.com/



NEMO Meta-OS

Repositories Starred

Displaying 1 to 12 of 12 repositories

nemometaos/meta-orchestrator-api

nemometaos/mocu

nemometaos/fcm-ui

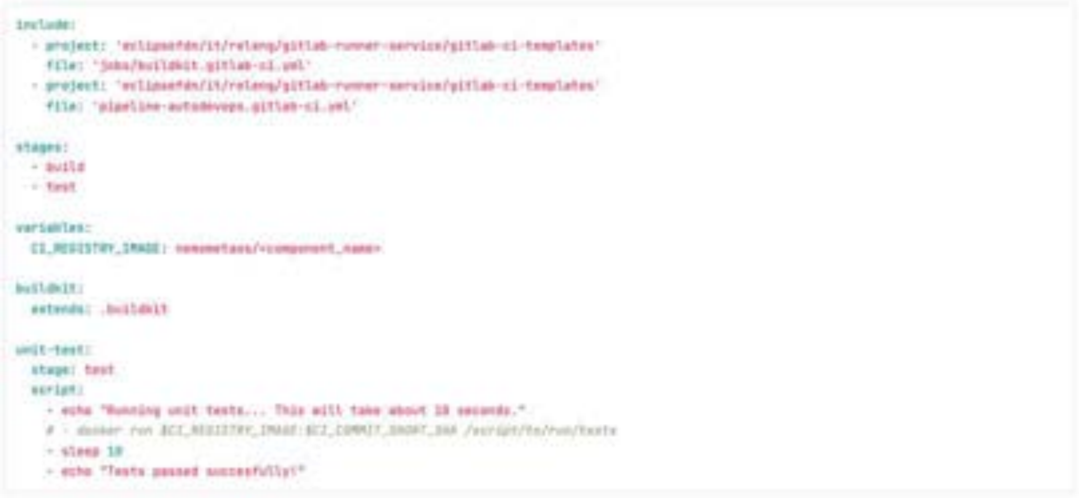
nemometaos/intent-api

nemometaos/access-control

nemometaos/mocu-lbs

Figure 3: NEMO meta-OS docker registry

Once a valid *dockerfile*[3] exists, a *".gitlab-ci.yml"* file must be created in the project root directory. Then once the *dockerfile* exists for the NEMO developed component the *".gitlab-ci.yml"* file must be created in the project root directory. Figure 4 below presents the relevant configuration file that the NEMO partners must adapt to their technical solution according to the instructions that are provided.



```
include:
  - project: 'eclipsefdn/it/releng/gitlab-runner-service/gitlab-ci-templates'
    file: 'jobs/buildkit.gitlab-ci.yml'
  - project: 'eclipsefdn/it/releng/gitlab-runner-service/gitlab-ci-templates'
    file: 'pipeline-autodevops.gitlab-ci.yml'

stages:
  - build
  - test

variables:
  CI_REGISTRY_IMAGE: nemometaos/<component_name>

buildkit:
  extends: .buildkit

unit-test:
  stage: test
  script:
    - echo "Running unit tests... This will take about 10 seconds."
    # - docker run $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA /script/to/run/tests
    - sleep 10
    - echo "Tests passed successfully!"
```

Figure 4: NEMO gitlab-ci.yml configuration

[3] https://docs.docker.com/reference/dockerfile/

In the above example the NEMO partners have to substitute the <*component_name*> with the name of their component. This configuration will do the following:

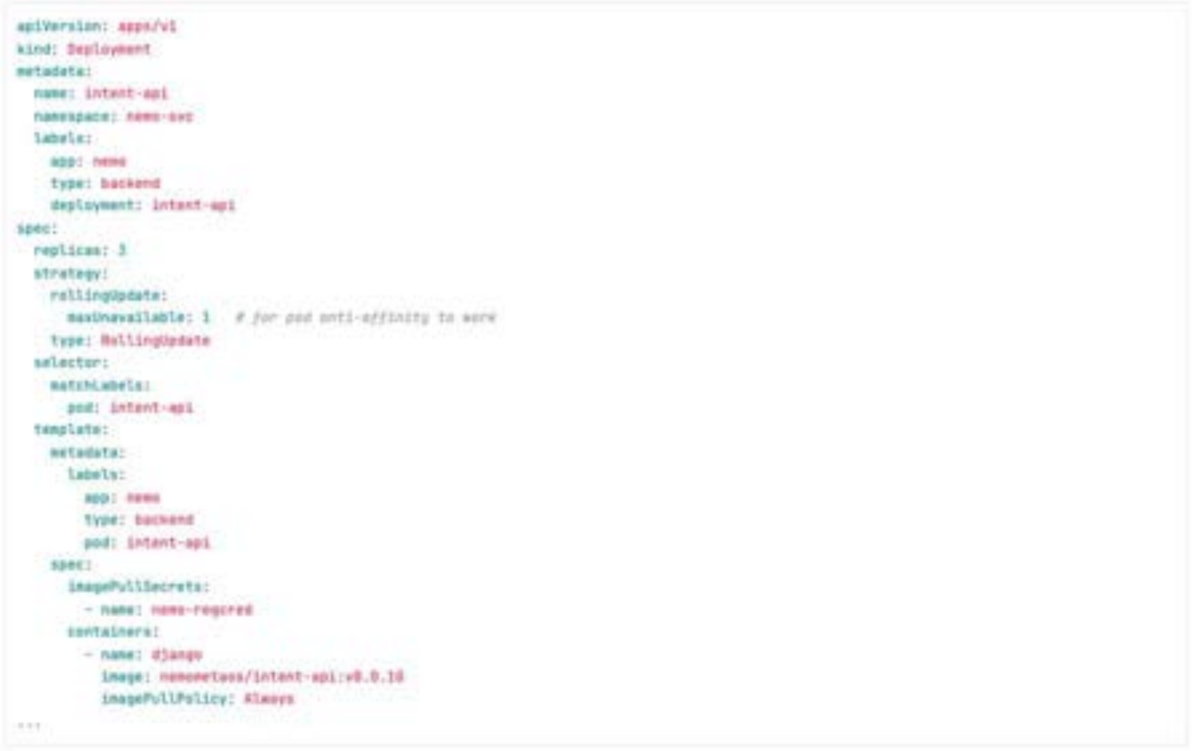
Upload the docker image *nemometaos/<component_name>:latest* tag every time a commit happens to the main branch

Upload the docker *image nemometaos/<component_name>:<tag_name>* tag every time a new tag is created.

For example, the creation of tag v0.1.1 on the *nemometaos/intent-api* uploaded to the docker registry is *nemometaos/intent-api:v0.1.1* image. The version of the component (tag) must be always ascending integers.

In order to deploy the NEMO component to Kubernetes orchestrated environment the component owner must provide all the necessary kubernetes configuration files (manifests) and test that they can be deployed and work in the OneLab cluster by using the provided credentials to access the cluster.

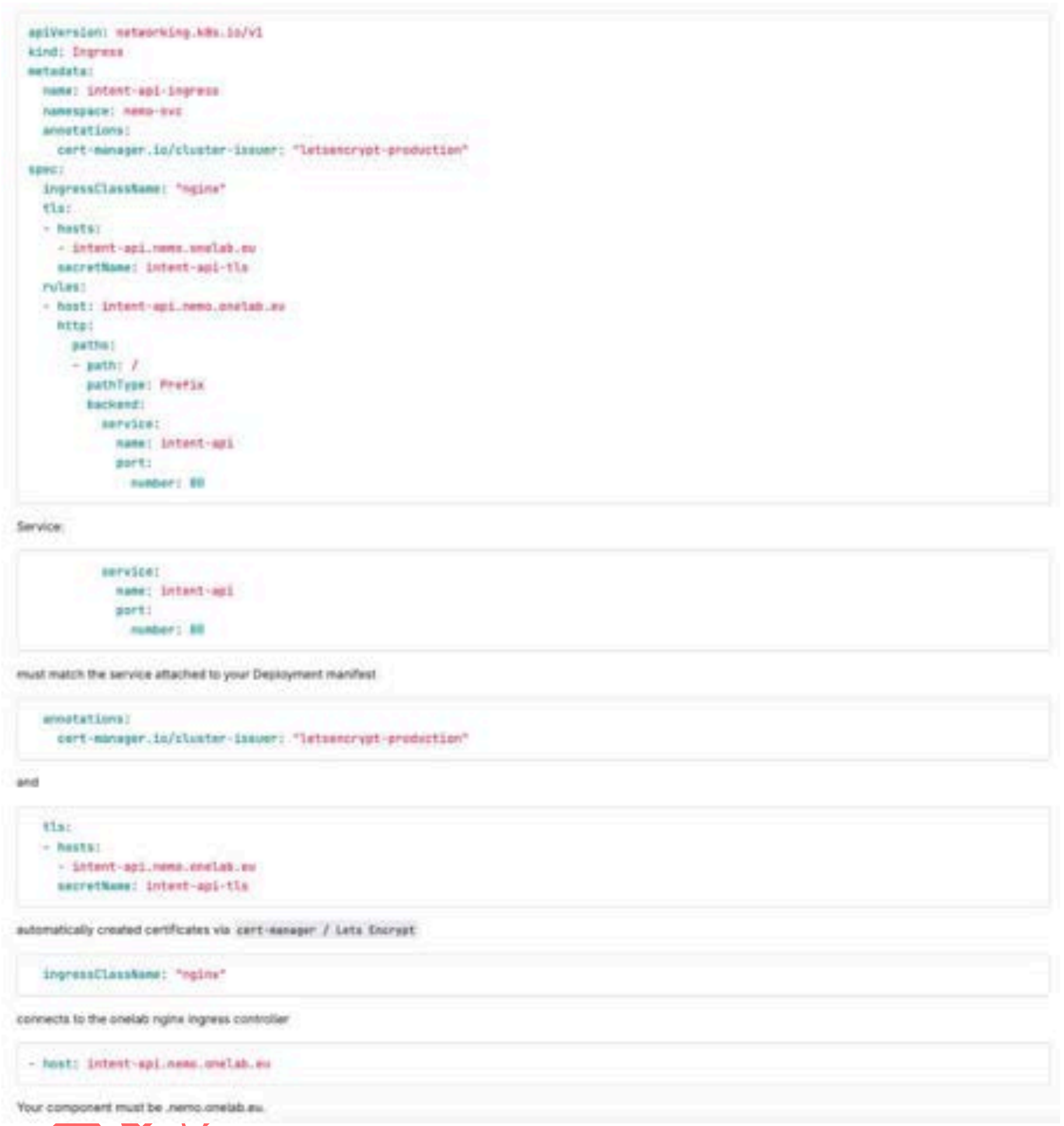
In order to pull images from the *nemometaos* account, every namespace in Kubernetes has the *nemo-regcred* secret that must be used as *imagePullSecrets* in the component's *Deployment* manifest as indicated in the following example.



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: intent-api
  namespace: nemo-svc
  labels:
    app: nemo
    type: backend
    deployment: intent-api
spec:
  replicas: 3
  strategy:
    rollingUpdate:
      maxUnavailable: 1    # for pod anti-affinity to work
    type: RollingUpdate
  selector:
    matchLabels:
      pod: intent-api
  template:
    metadata:
      labels:
        app: nemo
        type: backend
        pod: intent-api
    spec:
      imagePullSecrets:
        - name: nemo-regcred
      containers:
        - name: django
          image: nemometaos/intent-api:v0.0.10
          imagePullPolicy: Always
...
```

Figure 5: Kubernetes deployment manifest example (intent-based API)

Moreover, the NEMO meta-OS provided integration guide described the *Ingress* configuration file that concerns the communication of the outside world with the deployed component.



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: intent-api-ingress
  namespace: nemo-svc
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-production"
spec:
  ingressClassName: "nginx"
  tls:
  - hosts:
    - intent-api.nemo.onelab.eu
    secretName: intent-api-tls
  rules:
  - host: intent-api.nemo.onelab.eu
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: intent-api
            port:
              number: 80
```

Service:

```
          service:
            name: intent-api
            port:
              number: 80
```

must match the service attached to your Deployment manifest

```
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-production"
```

and

```
  tls:
  - hosts:
    - intent-api.nemo.onelab.eu
    secretName: intent-api-tls
```

automatically created certificates via `cert-manager / Lets Encrypt`

```
  ingressClassName: "nginx"
```

connects to the onelab nginx ingress controller

```
- host: intent-api.nemo.onelab.eu
```

Your component must be .nemo.onelab.eu

Figure 6: Kubernetes ingress manifest example (intent-based API)

Once the steps mentioned above has been completed then the developed component is deployed successfully in the Kubernetes orchestrated environment in OneLab. In NEMO, the Continuous Deployment part of the pipeline is configured through the FLUX CD[4] which is a CNCF[5] adopted open-source tool that enables GitOps for managing the configuration of a Kubernetes cluster. In a GitOps pipeline, the desired state of the cluster is stored in a Git repository, and FLUX CD ensures that the actual cluster state matches the desired state defined in the repository.

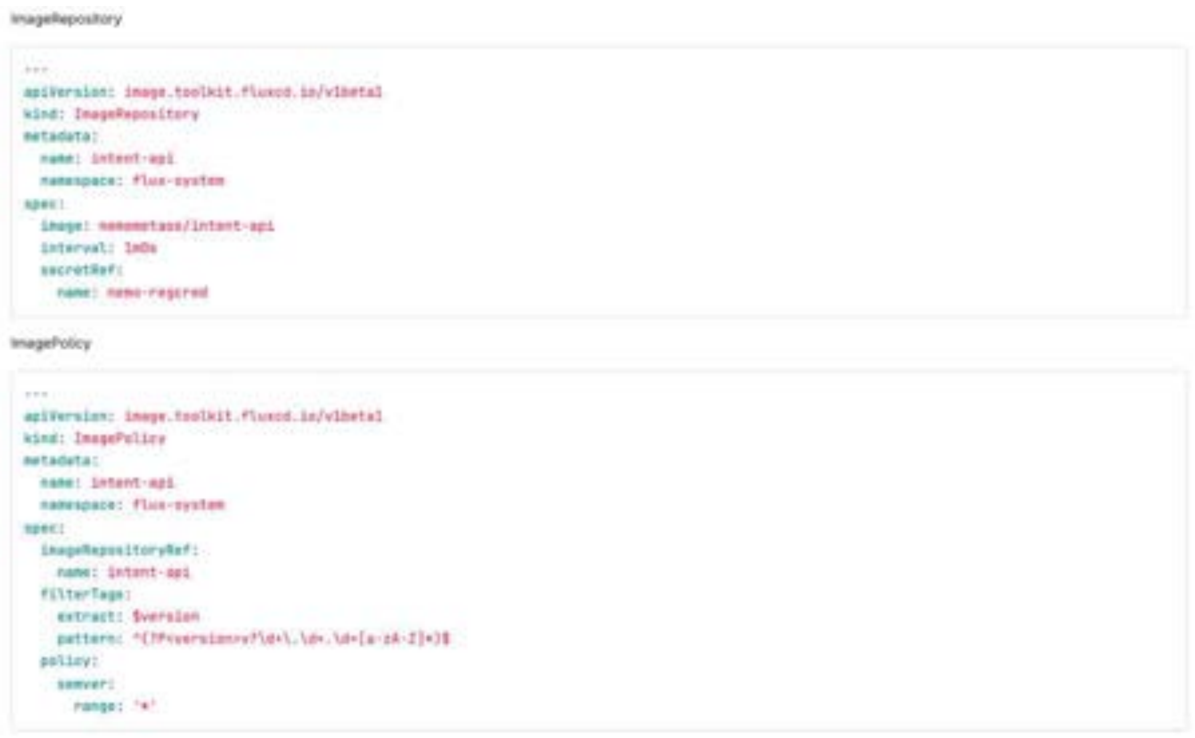
[4] https://fluxcd.io/
[5] https://cncf.io/

Once the manifests are created and the component is verified that is working, the manifests must be transferred to the FLUX CD repository. The repository is structured in folders as follows:

<cluster_name> / <kubernetes_namespace> / <component_name> / <component_subcomponents>

Each component must be deployed into the respective folder that matched their Kubernetes assigned namespace. Sub dependencies (e.g. postgres[6], redis[7] etc) can be also setup as Helm charts[8].

To automate the process an *ImageRepository* & ImagePolicy Custom Resource Definition (CRD) must be committed alongside the component manifests as indicated by the following examples.



ImageRepository

```
---
apiVersion: image.toolkit.fluxcd.io/v1beta1
kind: ImageRepository
metadata:
  name: intent-api
  namespace: flux-system
spec:
  image: nemometaos/intent-api
  interval: 1m0s
  secretRef:
    name: nemo-regcred
```

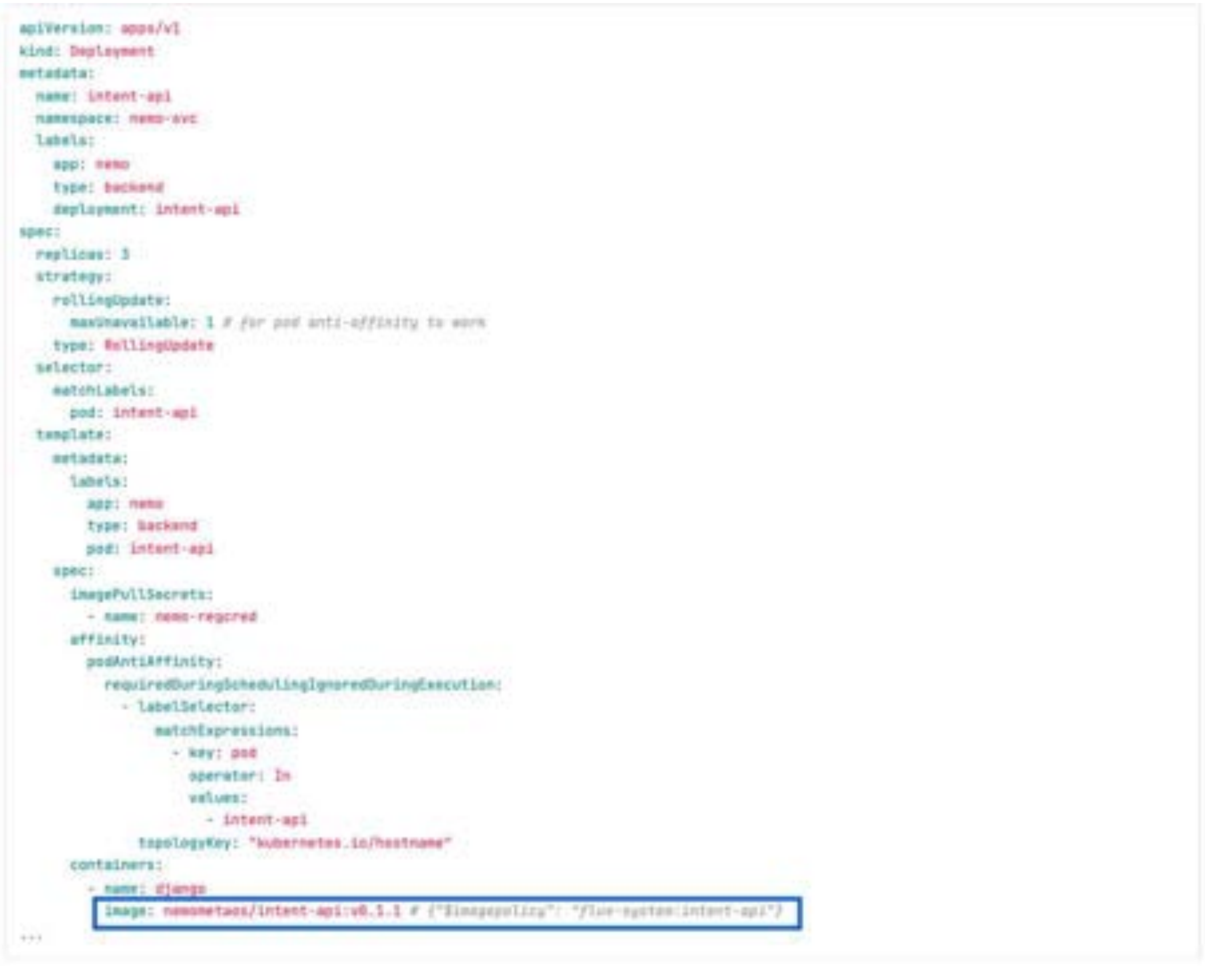
ImagePolicy

```
---
apiVersion: image.toolkit.fluxcd.io/v1beta1
kind: ImagePolicy
metadata:
  name: intent-api
  namespace: flux-system
spec:
  imageRepositoryRef:
    name: intent-api
  filterTags:
    extract: $version
    pattern: ^(?P<version>v?\d+\.\d+.\d+[a-zA-Z]*)$
  policy:
    semver:
      range: '*'
```

Figure 7: Image repository and policy configuration files

The proper names must be set and must reside inside the *flux-system* namespace, and the appropriate image repository must be set too. After that, the *Deployment manifest* of the component must upsert the following annotation to the line that defines the newly created docker image as indicated in the rectangular box in Figure 8 below.

[6] https://www.postgresql.org/
[7] https://redis.io/
[8] https://helm/sh/



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: intent-api
  namespace: nemo-svc
  labels:
    app: nemo
    type: backend
    deployment: intent-api
spec:
  replicas: 3
  strategy:
    rollingUpdate:
      maxUnavailable: 1 # for pod anti-affinity to work
    type: RollingUpdate
  selector:
    matchLabels:
      pod: intent-api
  template:
    metadata:
      labels:
        app: nemo
        type: backend
        pod: intent-api
    spec:
      imagePullSecrets:
        - name: nemo-regcred
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: pod
                    operator: In
                    values:
                      - intent-api
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: django
          image: nemometaos/intent-api:v0.1.1 # {"$imagepolicy": "flux-system:intent-api"}
...
```

Figure 8: Example of the Deployment manifest

After that, any ***ascending, new tag version***, that is created from the component repository, will be pushed to the docker hub repository and set inside the deployment manifest (version bump) as commit to the repository by FLUX CD.

2.2 Cloud/Edge/IoT Integration and Validation Infrastructure

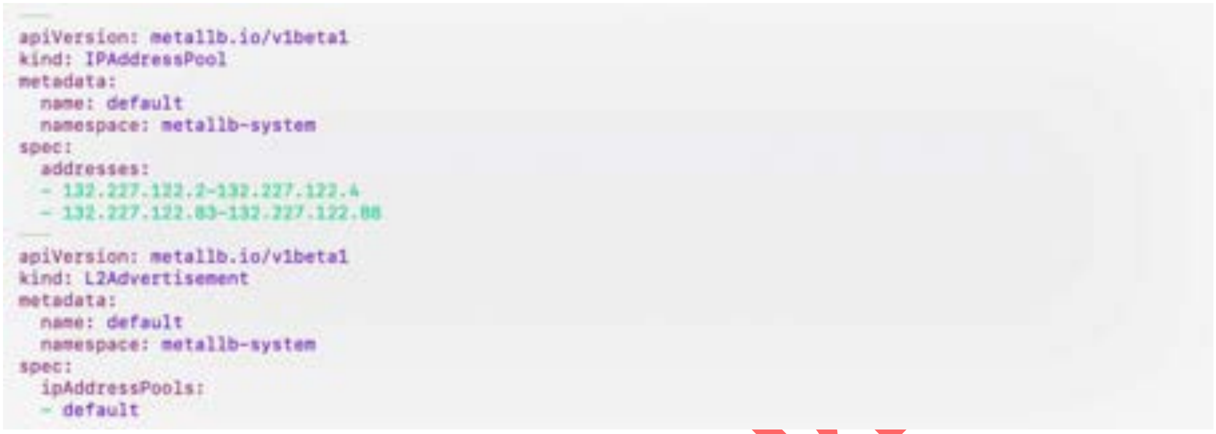
2.2.1 OneLab Clusters for NEMO

Four distinct Kubernetes clusters have been established to fulfil specific operational requirements. These include one primary cluster designated for development workloads, two supporting clusters including a lightweight cluster deployed on Raspberry Pis[9] and finally the production cluster optimized for handling tasks requiring Graphics Processing Unit (GPU) resources.

Each cluster consists of a series of nodes structured to ensure efficient operation. The master node is responsible for core functionalities such as application scheduling, scaling, and overarching cluster

[9] https://www.raspberrypi.com/

management. Worker nodes are dedicated to executing tasks assigned by the master node, which include deploying containers and hosting applications. In a production environment, multiple worker nodes are typically utilized to provide redundancy and enhance service availability, thereby ensuring robust and uninterrupted operations. Additionally, each cluster is configured with a load balancer, which distributes incoming network traffic across the nodes to ensure optimal resource utilization, fault tolerance, and high availability of the services.

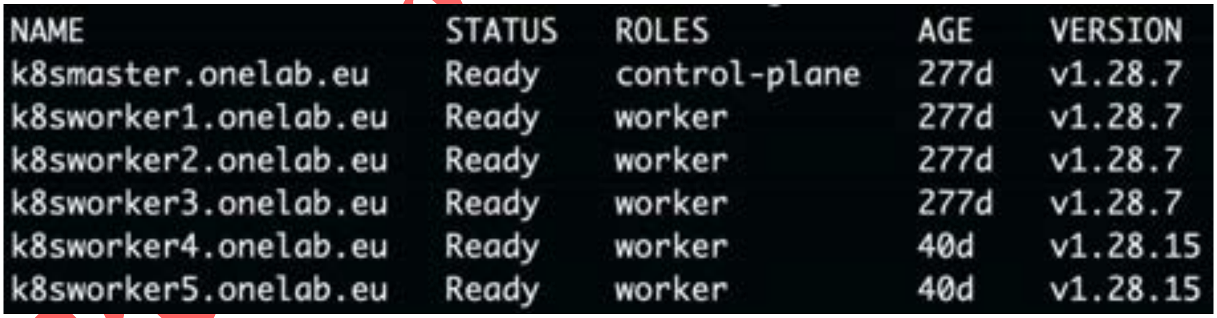


```
---
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: default
  namespace: metallb-system
spec:
  addresses:
  - 132.227.122.2-132.227.122.4
  - 132.227.122.83-132.227.122.88
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: default
  namespace: metallb-system
spec:
  ipAddressPools:
  - default
```

Figure 9: Example of MetalLB[10] configuration

2.2.1.1 NEMO Dev cluster

The development cluster consists of one control-plane node (k8smaster.onelab.eu) and five worker nodes (k8sworker1-5.onelab.eu), all in a *Ready* status. The Kubernetes specified versions ranging from v1.28.7 to v1.28.15.



```
NAME                   STATUS   ROLES           AGE    VERSION
k8smaster.onelab.eu    Ready    control-plane   277d   v1.28.7
k8sworker1.onelab.eu   Ready    worker          277d   v1.28.7
k8sworker2.onelab.eu   Ready    worker          277d   v1.28.7
k8sworker3.onelab.eu   Ready    worker          277d   v1.28.7
k8sworker4.onelab.eu   Ready    worker          40d    v1.28.15
k8sworker5.onelab.eu   Ready    worker          40d    v1.28.15
```

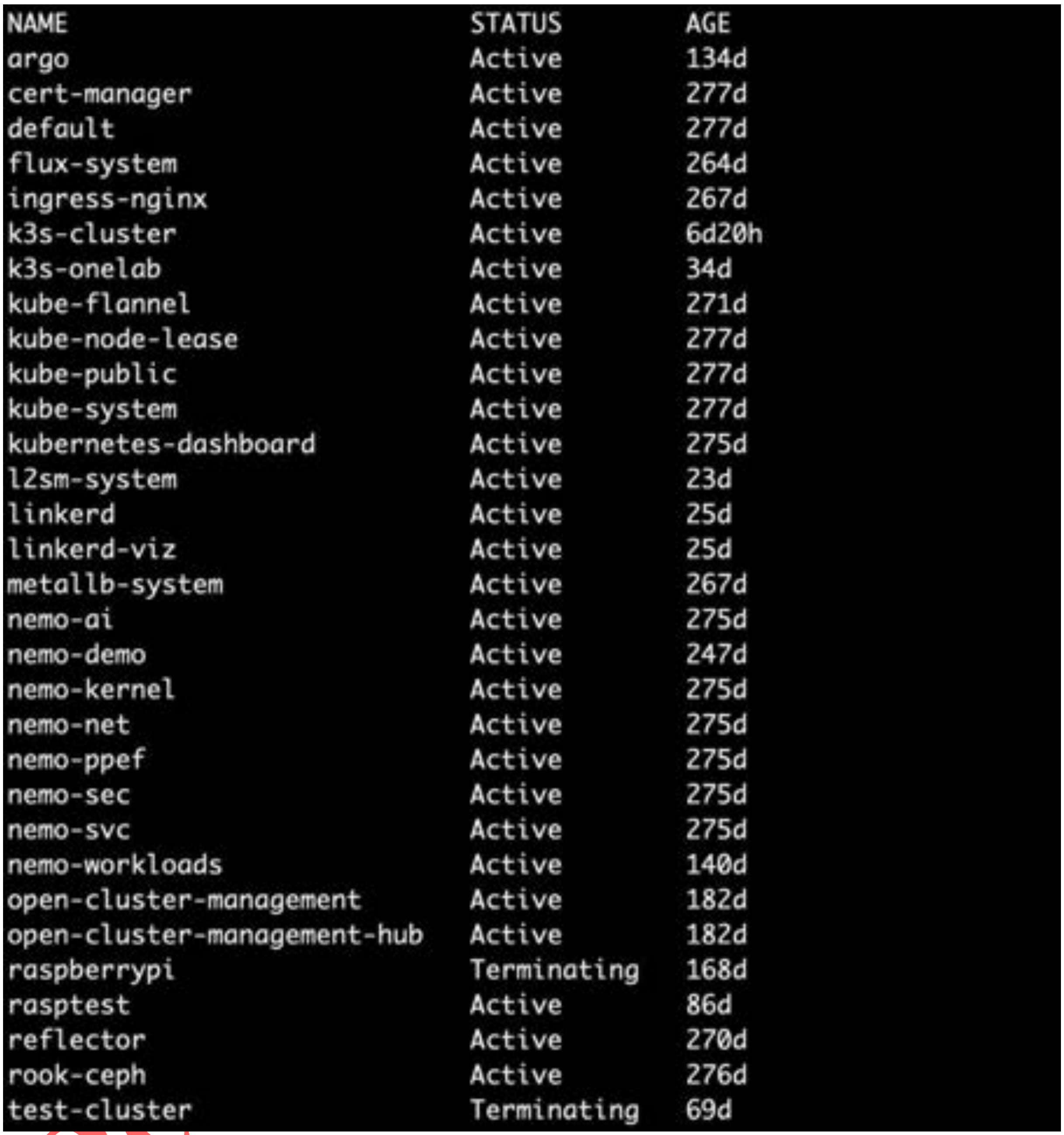
Figure 10: Dev cluster nodes

Node name	Node Type	Specifications	Public IP
k8smaster.onelab.eu	Master	**CPU**: 8 CPU Cores **RAM:** 16GB	132.227.122.23

[10] https://metallb.universe.tf/

Node name	Node Type	Specifications	Public IP
		Storage: 140GB Ephemeral **OS-Image**: Ubuntu 22.04.4 LTS **Kernel Version**: 5.15.0-116-generic **Container-runtime**: containerd://1.6.28	
k8sworker1.onelab.eu	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS-Image**: Ubuntu 22.04.4 LTS **Kernel Version**: 5.15.0-116-generic **Container-runtime**: containerd://1.6.28	132.227.122.66
k8sworker2.onelab.eu	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS-Image**: Ubuntu 22.04.4 LTS **Kernel Version**: 5.15.0-116-generic **Container-runtime**: containerd://1.6.28	132.227.122.24
k8sworker3.onelab.eu	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS-Image**: Ubuntu 22.04.4 LTS **Kernel Version**: 5.15.0-116-generic **Container-runtime**: containerd://1.6.28	132.227.122.59
k8sworker4.onelab.eu	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS-image**: Ubuntu 22.04.4 LTS **Kernel Version**: 5.15.0-116-generic **Container-runtime**: containerd://1.6.28	132.227.122.41
k8sworker5.onelab.eu	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB RAM **Storage**: 120GB Ephemeral + 150GB Ceph **OS-image**: Ubuntu 22.04.4 LTS **Kernel Version**: 5.15.0-116-generic **Container-runtime**: containerd://1.6.28	132.227.122.47

Table 1: NEMO dev cluster (K8S)

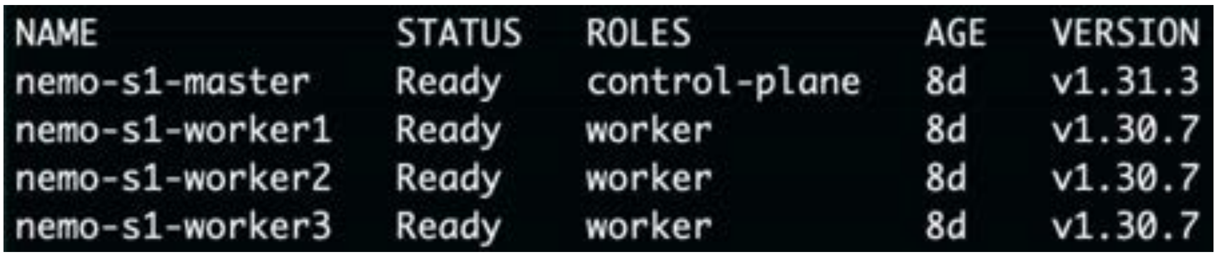


```
NAME                            STATUS        AGE
argo                            Active        134d
cert-manager                    Active        277d
default                         Active        277d
flux-system                     Active        264d
ingress-nginx                   Active        267d
k3s-cluster                     Active        6d20h
k3s-onelab                      Active        34d
kube-flannel                    Active        271d
kube-node-lease                 Active        277d
kube-public                     Active        277d
kube-system                     Active        277d
kubernetes-dashboard            Active        275d
l2sm-system                     Active        23d
linkerd                         Active        25d
linkerd-viz                     Active        25d
metallb-system                  Active        267d
nemo-ai                         Active        275d
nemo-demo                       Active        247d
nemo-kernel                     Active        275d
nemo-net                        Active        275d
nemo-ppef                       Active        275d
nemo-sec                        Active        275d
nemo-svc                        Active        275d
nemo-workloads                  Active        140d
open-cluster-management         Active        182d
open-cluster-management-hub     Active        182d
raspberrypi                     Terminating   168d
rasptest                        Active        86d
reflector                       Active        270d
rook-ceph                       Active        276d
test-cluster                    Terminating   69d
```

Figure 11: Dev cluster namespaces

2.2.1.2 Staging 1 cluster

The staging cluster (Staging 1) comprises one control-plane node (nemo-s1-master) and three worker nodes (nemo-s1-worker1, nemo-s1-worker2, and nemo-s1-worker3), all reporting a *Ready* status, running Kubernetes versions v1.31.3 (control-plane) and v1.30.7 (workers).

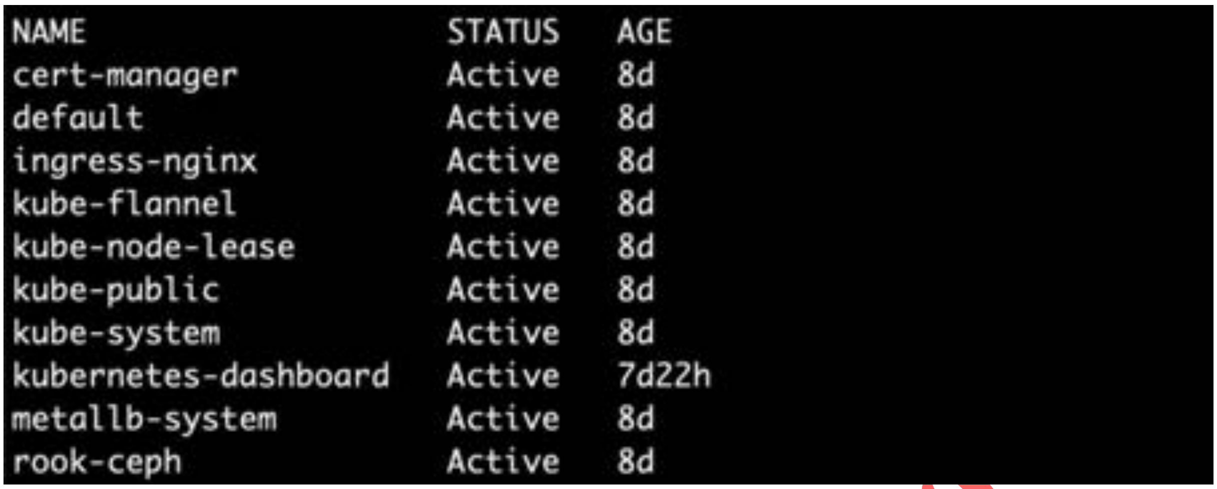


```
NAME               STATUS   ROLES           AGE   VERSION
nemo-s1-master     Ready    control-plane   8d    v1.31.3
nemo-s1-worker1    Ready    worker          8d    v1.30.7
nemo-s1-worker2    Ready    worker          8d    v1.30.7
nemo-s1-worker3    Ready    worker          8d    v1.30.7
```

Figure 12: Staging 1 cluster nodes

Node Name	Node Type	Specifications	Public IP
nemo-s1-master	Master	**CPU**: 8 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS Image**: Ubuntu 22.04.3 LTS **Kernel Version**: 5.15.0-78-generic **Container-runtime**: containerd://1.7.12	132.227.122.104
nemo-s1-worker1	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS Image**: Ubuntu 22.04.3 LTS **Kernel Version**: 5.15.0-78-generic **Container-runtime**: containerd://1.7.12	132.227.122.105
nemo-s1-worker2	Worker and Storage	**CPU**: 16 CPU Cores **RAM**: 16GB **Storage**: 120GB Ephemeral + 150GB Ceph **OS Image**: Ubuntu 22.04.3 LTS **Kernel Version**: 5.15.0-78-generic **Container-runtime**: containerd://1.7.12	132.227.122.106
nemo-s1-worker3	Worker and Storage	**CPU:** 16 CPU Cores **RAM:** 16GB **Storage:** 120GB Ephemeral + 150GB Ceph **OS Image:** Ubuntu 22.04.3 LTS **Kernel Version:** 5.15.0-78-generic **Container-runtime:** containerd://1.7.12	132.227.122.107

Table 2: Staging 1 cluster (K8S)

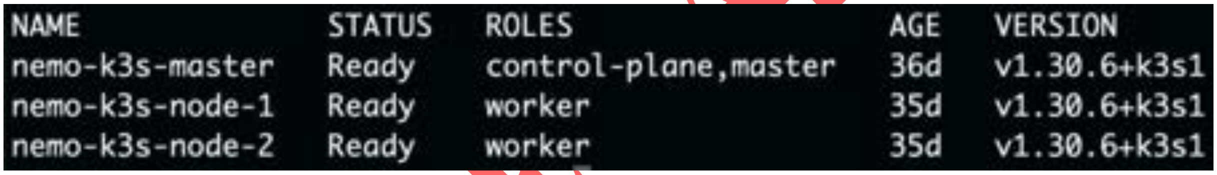


```
NAME                     STATUS   AGE
cert-manager             Active   8d
default                  Active   8d
ingress-nginx            Active   8d
kube-flannel             Active   8d
kube-node-lease          Active   8d
kube-public              Active   8d
kube-system              Active   8d
kubernetes-dashboard     Active   7d22h
metallb-system           Active   8d
rook-ceph                Active   8d
```

Figure 13: Staging 1 cluster namespaces

2.2.1.3 Staging 2 cluster (K3S)

The Staging 2), deployed on Raspberry Pis 4, consists of one control-plane node (nemo-k3s-master) and two worker nodes (nemo-k3s-node-1 and nemo-k3s-node-2), all in a *Ready* state and running Kubernetes version v1.30.6+k3s1.



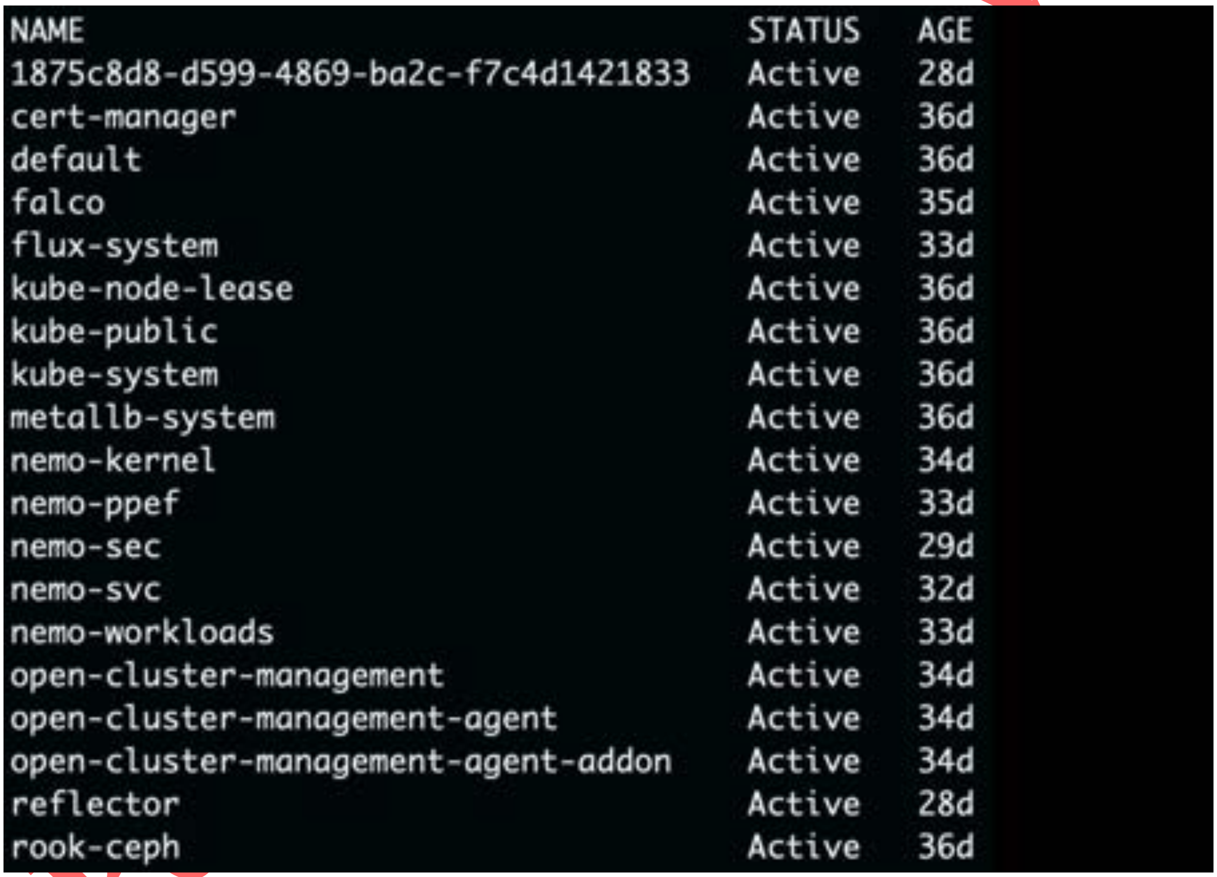
```
NAME              STATUS   ROLES                  AGE   VERSION
nemo-k3s-master   Ready    control-plane,master   36d   v1.30.6+k3s1
nemo-k3s-node-1   Ready    worker                 35d   v1.30.6+k3s1
nemo-k3s-node-2   Ready    worker                 35d   v1.30.6+k3s1
```

Figure 14: Staging 2 cluster nodes

Node name	Node Type	Specifications	Public IP
nemo-k3s-master	Master	**CPU**: 4 CPU Cores **RAM**: 8GB **Storage**: 64GB External SSD **OS Image**: Ubuntu 24.10 **Kernel Version**: 6.11.0-1004-raspi **Container-runtime**: containerd://1.7.22-k3s1	132.227.122.99
nemo-k3s-node-2	Worker and Storage	**CPU**: 4 CPU Cores **RAM**: 8GB **Storage**: 1TB External SSD **OS Image**: Ubuntu 24.10 **Kernel Version**: 6.11.0-1004-raspi **Container-runtime**: containerd://1.7.22-k3s1	132.227.122.88

Node name	Node Type	Specifications	Public IP
nemo-k3s-node-3	Worker and Storage	**CPU**: 4 CPU Cores **RAM**: 8GB **Storage**: 64GB External SSD **OS Image**: Ubuntu 24.10 **Kernel Version**: 6.11.0-1004-raspi **Container-runtime**: containerd://1.7.22-k3s1	132.227.122.91

Table 3: Staging 2 Cluster (K3S)



```
NAME                                   STATUS   AGE
1875c8d8-d599-4869-ba2c-f7c4d1421833   Active   28d
cert-manager                           Active   36d
default                                Active   36d
falco                                  Active   35d
flux-system                            Active   33d
kube-node-lease                        Active   36d
kube-public                            Active   36d
kube-system                            Active   36d
metallb-system                         Active   36d
nemo-kernel                            Active   34d
nemo-ppef                              Active   33d
nemo-sec                               Active   29d
nemo-svc                               Active   32d
nemo-workloads                         Active   33d
open-cluster-management                Active   34d
open-cluster-management-agent          Active   34d
open-cluster-management-agent-addon    Active   34d
reflector                              Active   28d
rook-ceph                              Active   36d
```

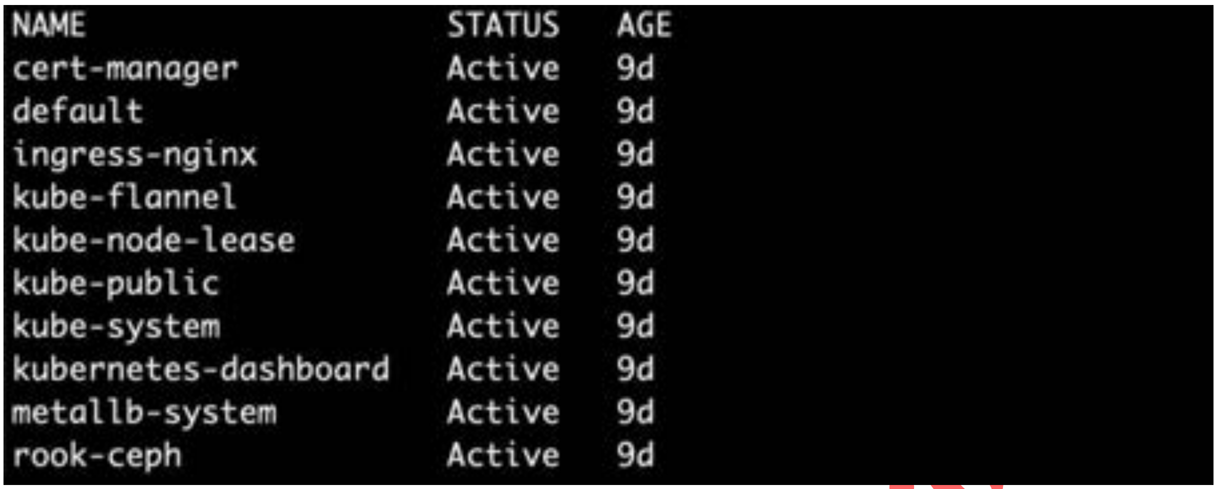
Figure 15: Staging 2 cluster namespaces

2.2.1.4 Production cluster (K8S)

The production cluster includes one control-plane node (nemo-prod-master), three worker nodes (nemo-prod-worker1, nemo-prod-worker2, and nemo-prod-worker3), and one GPU-enabled worker node (nemo-prod-gpu-worker), all in a Ready state and running Kubernetes version v1.30.7.

Node Name	Node Type	Specifications	Public IP
nemo-prod-master	Master	**CPU:** 4 CPU Cores **RAM:** 8GB **Storage:** 80GB Ephemeral **OS Image:** Ubuntu 22.04.3 LTS **Kernel Version:** 5.15.0-78-generic **Container-runtime:** containerd://1.7.12	132.227.122.42
nemo-prod-worker1	Worker and Storage	**CPU:** 8 CPU Cores **RAM:** 16GB **Storage:** 250GB Ephemeral + 150GB Ceph **OS Image:** Ubuntu 22.04.3 LTS **Kernel Version:** 5.15.0-78-generic **Container-runtime:** containerd://1.7.12	132.227.122.43
nemo-prod-worker2	Worker and Storage	**CPU:** 8 CPU Cores **RAM:** 16GB **Storage:** 120GB Ephemeral + 150GB Ceph **OS Image:** Ubuntu 22.04.4 LTS **Kernel Version:** 5.15.0-78-generic **Container-runtime:** containerd://1.7.12	132.227.122.113
nemo-prod-worker3	Worker and Storage	**CPU:** 8 CPU Cores **RAM:** 16GB **Storage:** 120GB Ephemeral + 150GB Ceph **OS Image:** Ubuntu 22.04.2 LTS **Kernel Version:** 5.15.0-78-generic **Container-runtime:** containerd://1.7.12	132.227.122.114
nemo-prod-gpu-worker	Worker and Storage	**CPU:** 4 CPU Cores **RAM:** 8GB **Storage:** 120GB Ephemeral + 150GB Ceph **OS Image:** Ubuntu 22.04.4 LTS **Kernel Version:** 5.15.0-78-generic **Container-runtime:** containerd://1.7.12	132.227.122.115

Table 4: Production Cluster (k8S)



```
NAME                   STATUS   AGE
cert-manager           Active   9d
default                Active   9d
ingress-nginx          Active   9d
kube-flannel           Active   9d
kube-node-lease        Active   9d
kube-public            Active   9d
kube-system            Active   9d
kubernetes-dashboard   Active   9d
metallb-system         Active   9d
rook-ceph              Active   9d
```

Figure 16: Production cluster namespaces

2.3 Integration & V&V Methodology & Plan

NEMO will follow an agile and incremental approach of iteration cycles, grouped in 3 Phases, as depicted in Figure 17.

Phase 1: Baseline *(M1-M18).* Provides the initial NEMO Proof of Concept. Phase 1 starts with system, specification of the meta-OS Architecture and decomposition (WP1), design analysis, prototyping (WP2-WP4), integration, testing and validation of all key meta-OS components (WP4). The outcome will be *NEMO Ver. A* and initial Living Labs validation and the selection of the new consortium members and new components from Open Call #1 to be implemented with Phase 2.

Phase 2: Advance *(M19-M30).* All NEMO components are further developed (WP2-WP4), while NEMO is expanded with new functionality added from the new consortium members accepted via Open Call #1. Stronger integration with 5G networks and MANO systems will be realized and validated in Living Labs). The outcome will be *NEMO Ver. B* and Living Labs validation, along with new AIoT applications and services from Open Call #2.

Phase 3: Mature *(M30-M36).* Focus on validation and optimization, and more realistic field conditions testing and verification, not only from NEMO consortium but also from 3rd parties selected via Open Call #2, increasing system TRL and preparing NEMO Ver. 1.0, validated in Living Labs. This phase also strengthens activities related to engagement of open-source communities and relevant initiatives, ensuring accessibility, sustainability and availability in open-source platforms.

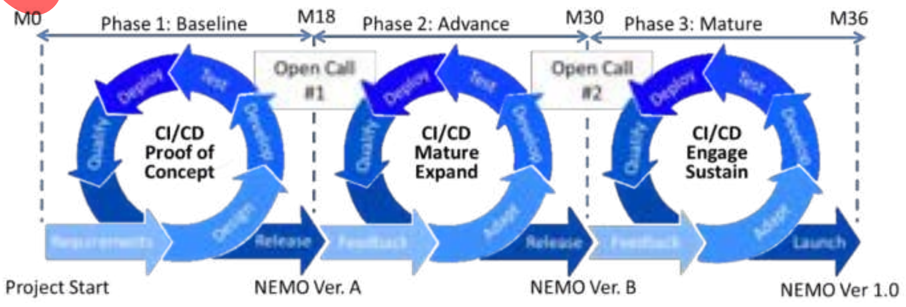


Figure 17: NEMO project phases and main meta-OS version releases

It should be underlined that each phase will follow an agile and incremental ***Continuous Integration/ Continuous Deployment/ Continuous Piloting (CI/CD/CP)*** approach, as explained in the previous subsections. The proposed approach allows responding to developments in the state of the art and emerging technology trends, as well as to continuously improve the results based on experimentation in the field.

2.3.1.1 NEMO Scenario-driven integration and verification

This section elaborates on the NEMO scenario-driven approach that is adopted by the project as the foundation of the integration activities that resulted into the first[t] integrated NEMO platform.

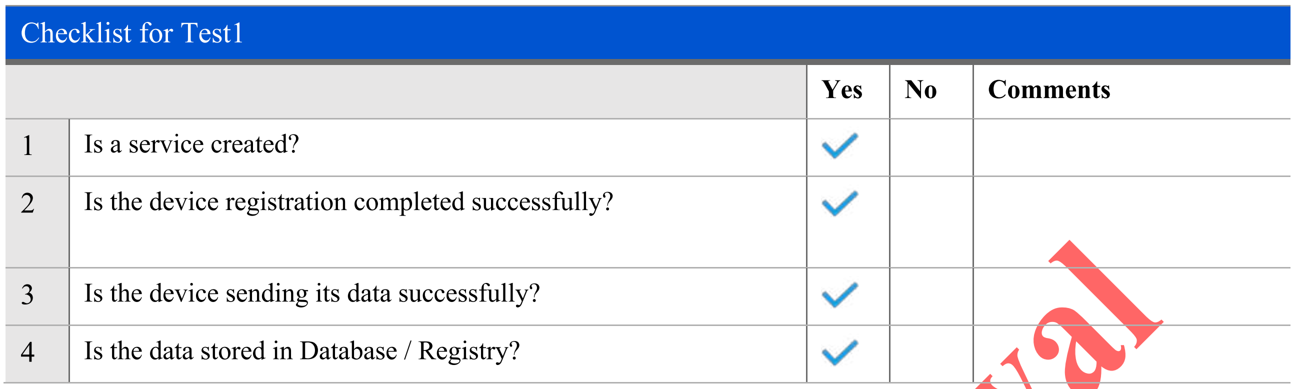
The integration tests conducted are scenario-driven and each scenario covers a part of the integration workflow that are defined and described in section 4. The specified tests might be distinguished in bilateral, that is between component A and B, and/or system level cross-cutting ones.

The integration tests that are conducted follow the below presented structure. First, the scenario is defined. For this, the Scenario template presented below is used to specify the particular test. More specifically, the Scenario template incorporates details that pertain to the *objective* of the tests, the participating *components*, the *requirements* that are addressed, the *features* that are tested, the *steps* that are needed to be verified and finally the *test setup* which provides details on the integration setting that facilitates the test.

Then, the results are presented in detail as dictated by the steps that are defined in the scenario. Finally, the Checklist template is applied describing the successful or unsuccessful are reported and fall into the specific integration scenario. The stemming results for all the defined workflows are presented in section 5, following the abovementioned structure Scenario, *Outcome/Results, Checkpoints.*

Test 1:	
Objective	
Components	
Requirements alignment	
Features to be tested	
Test setup	
Steps	1. 2. 3.

Figure 18: Integration testing – Scenario template



Checklist for Test1				
		Yes	No	Comments
1	Is a service created?	✓		
2	Is the device registration completed successfully?	✓		
3	Is the device sending its data successfully?	✓		
4	Is the data stored in Database / Registry?	✓		

Figure 19: NEMO Integration testing - Checklist template (Example)

2.4 NEMO OneLab infrastructure deployments

The latest stable releases of the developed NEMO meta-OS components are deployed in the OneLab cluster as indicated in the following figures that depict the NEMO deployments in each namespace.

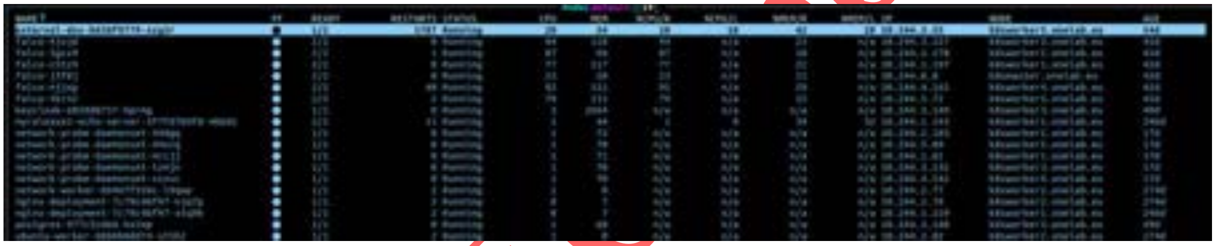


Figure 20: Default namespace

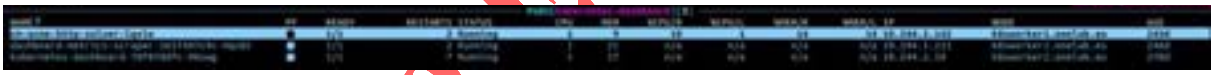


Figure 21: Kubernetes-dashboard namespace

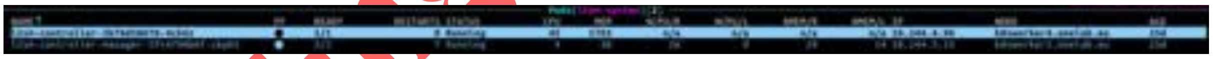


Figure 22: l2SM namespace

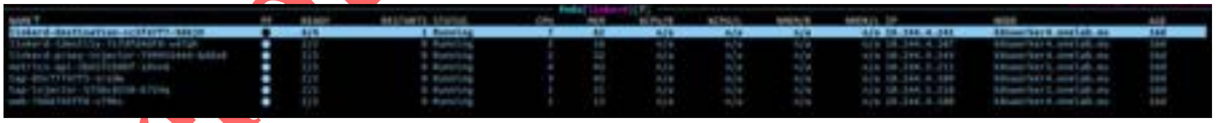


Figure 23: LinkerD namespace

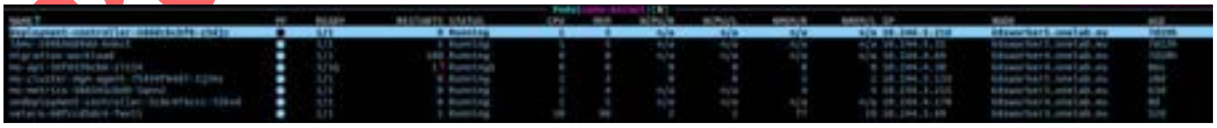


Figure 24: NEMO Kernel namespace

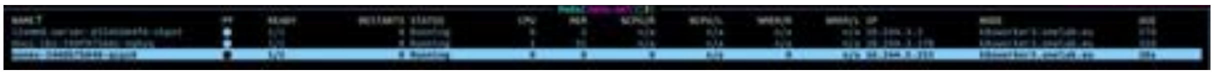


Figure 25: NEMO-net namespace

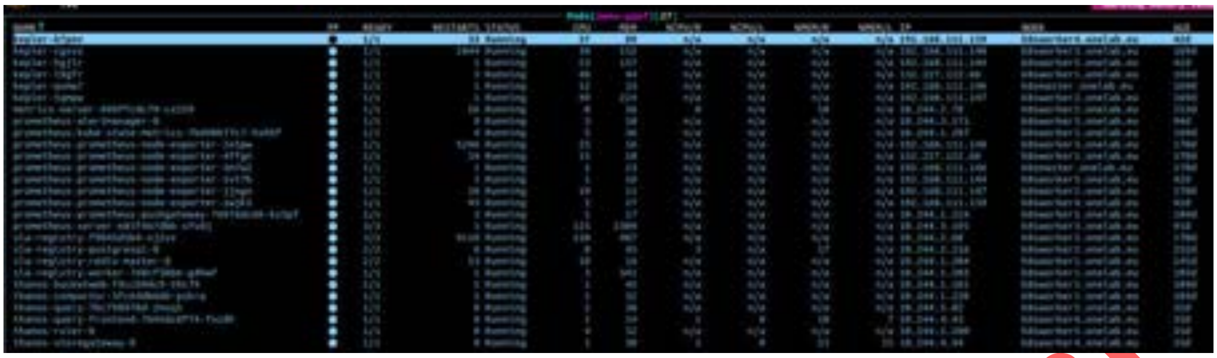

[illegible]

Figure 26: NEMO-PPEF namespace

[illegible]

Figure 27: NEMO-sec namespace

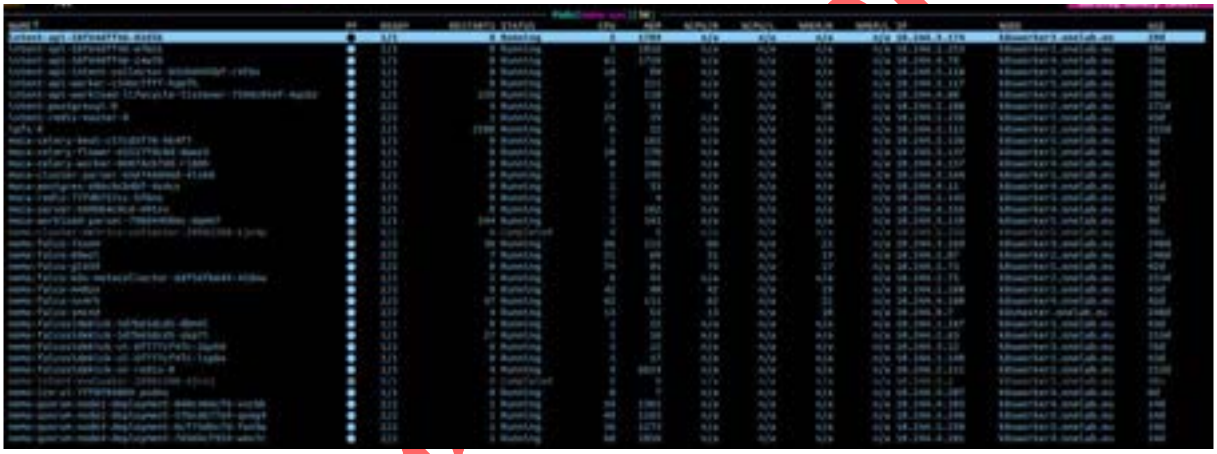

[illegible]

Figure 28: NEMO-svc namespace

The deployed workloads reside in a workloads' specific namespace as indicated in the figure below.

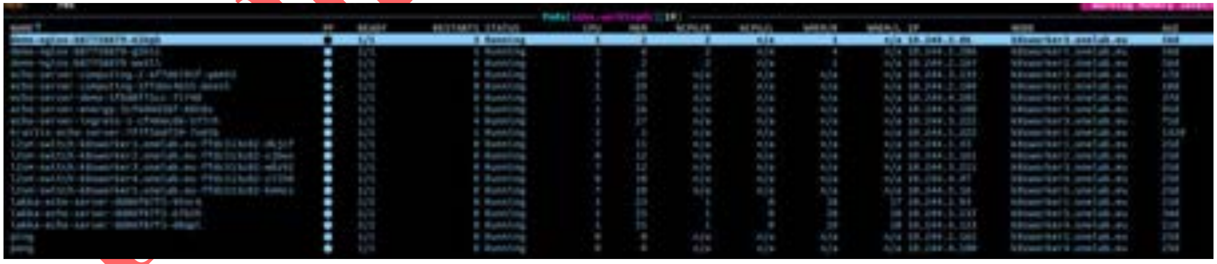

[illegible]

Figure 29: NEMO-workloads namespace

2.5 NEMO Integrated Platform (Ver. 1)

The NEMO meta-OS platform concerns a composition of a big set of technical tools residing in every functional layer of its architecture spanning from the infrastructure layer to the service management layer of the platform.

Figure 30 below illustrates the functional view of the NEMO meta-OS architecture which was introduced in D1.2 [3]. The functional view is segregated in three horizontal layers namely the

I*nfrastructure Management layer,* the *NEMO Kernel* and the *NEMO Service Management layer,* including three NEMO cross-cutting functions (verticals) namely, the PPEF, the CFDRL and the Cybersecurity & Federated Access Control.

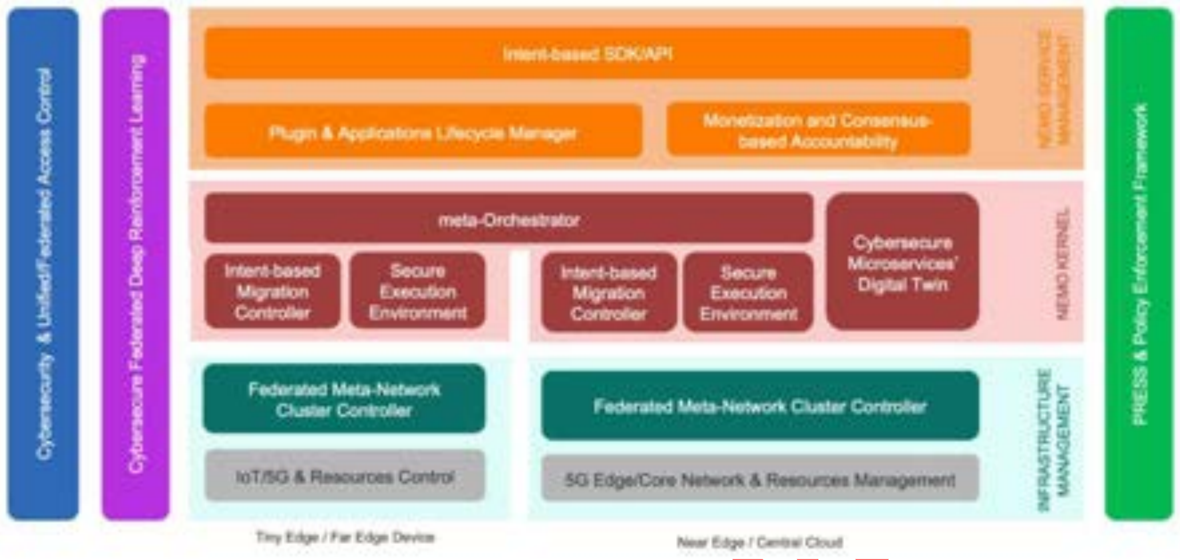

Cybersecurity & Unified/Federated Access Control
Cybersecure Federated Deep Reinforcement Learning
Intent-based SDK/API
Plugin & Applications Lifecycle Manager
Monetization and Consensus-based Accountability
NEMO SERVICE MANAGEMENT
meta-Orchestrator
Intent-based Migration Controller
Secure Execution Environment
Intent-based Migration Controller
Secure Execution Environment
Cybersecure Microservices' Digital Twin
NEMO KERNEL
Federated Meta-Network Cluster Controller
Federated Meta-Network Cluster Controller
IoT/5G & Resources Control
5G Edge/Core Network & Resources Management
INFRASTRUCTURE MANAGEMENT
PRESS & Policy Enforcement Framework
Tiny Edge / Far Edge Device
Near Edge / Central Cloud

Figure 30: The NEMO high-level architecture

The first integrated NEMO meta-OS platform materialized through four scenario-driven integration activities that aimed glue together the first stable releases of the NEMO technical ecosystem. Figure 31 depicts the overview of the first integrated NEMO platform. The square rectangles denote the integrated components.

The continuous lines indicate that the respective components are fully integrated meaning that the exposed interfaces, the respective data models and the provided functionality that is included in their latest stable release has been tested in the NEMO OneLab infrastructure and the communication between the participating modules has been successfully verified.

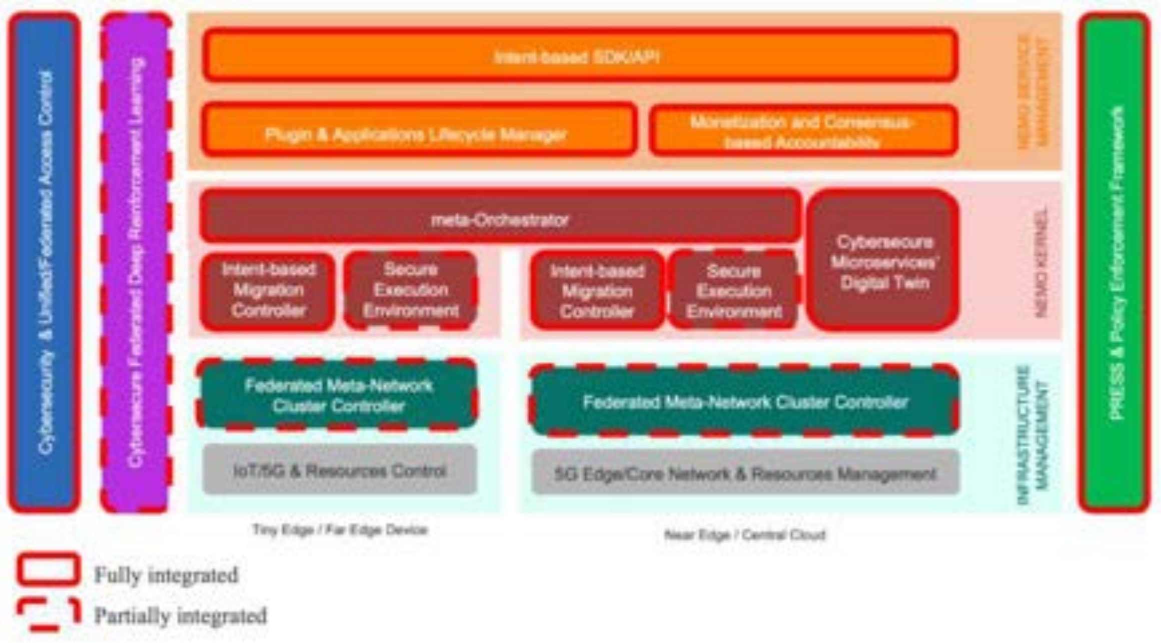

Cybersecurity & Unified/Federated Access Control
Cybersecure Federated Deep Reinforcement Learning
Intent-based SDK/API
Plugin & Applications Lifecycle Manager
Monetization and Consensus-based Accountability
NEMO SERVICE MANAGEMENT
meta-Orchestrator
Intent-based Migration Controller
Secure Execution Environment
Intent-based Migration Controller
Secure Execution Environment
Cybersecure Microservices' Digital Twin
NEMO KERNEL
Federated Meta-Network Cluster Controller
Federated Meta-Network Cluster Controller
IoT/5G & Resources Control
5G Edge/Core Network & Resources Management
INFRASTRUCTURE MANAGEMENT
PRESS & Policy Enforcement Framework
Tiny Edge / Far Edge Device
Near Edge / Central Cloud
Fully integrated
Partially integrated

Figure 31: The 1st integrated version of NEMO meta-OS (high-level architecture view)

One the other hand the dashed-line square rectangles indicate that the integration has been partially achieved. This means that for the particular components the integration within NEMO meta-OS has been achieved either through integration tests that were conducted in the context of partners premises or indicates that the integration was limited due to implementation activities that are currently ongoing.

The results of the integration tests that were executed in view of the first release of the NEMO meta-OS are presented in detail in section 4 of the deliverable. The main goal of the current release is to demonstrate a good level of system-level coherence, verifying its readiness for the final integration activities that will be performed to produce the final version of the NEMO meta-OS.

2.5.1 Meta-OS functionality in NEMO v1

This section aims to provide an overview of the conducted integration activities and provided functionality of the current development state of the NEMO meta-OS components for each functional layer of the NEMO meta-OS architecture.

2.5.1.1 NEMO Infrastructure Management

The mNCC is the NEMO components that provides an abstraction layer of the underlying infrastructure network level. The mNCC delivers network orchestration overseeing network connectivity management guaranteeing multi-cluster and multi-domain connectivity. The incorporated connectivity adaptors that provide for data translation between different network protocols. More specifically, the *L2S-M* enables dynamic creation and management of isolated virtual networks within meta-OS operator clusters, the *5G adaptor*, supports deterministic communications through TSN bridges with 5G LAN solutions and the SDN-based connectivity adaptor, supports network management. Finally, mNCC component facilitates the monitoring of the communications between pods deployed in each cluster's nodes. The collected data are communicated through RabbitMQ.

Regarding integration activities, mNCC offered functionality is currently being finalized and initial integration activities with on premise deployments have been achieved and documented. The latest development updates of the component are presented in D2.3 [4] that will be submitted on M28.

2.5.1.2 NEMO Kernel

The NEMO components associated with the Kernel layer are the MO, the CMDT, the IBMC and the SEE as illustrated in the NEMO meta-OS architecture view figures above. The core functionalities offered from the NEMO Kernel components have been presented, demonstrated and documented in the present document through scenario driven integration activities (section 4). More specifically, the MO facilitates the workload deployment and migration processes. For the latter the IBMC supports the workload migration process ensuring efficient resource use, improved scalability, and continuous service availability during migrations. The CMDT provides enhanced workload monitoring related measurements to the platform, which are being consumed through the RabbitMQ, by the NEMO meta-OS components. Finally, the SEE (Kubernetes cluster) which is a solution for creating secure execution environments for critical and dynamic services, ensuring robust, secure, and efficient operations, is available and integrated within NEMO meta-OS ecosystem. The NEMO user (workload provider) can formulate a request through the Service Management layer asking for their services to be deployed in SEE. This is indicated by the proper configuration of workloads' intents.

2.5.1.3 NEMO Service Management

The NEMO Service Management layer components' functional updates are presented in detail in section 3 of the present document. More specifically, section 3 provides insights on components' architecture, provided functionality offered by the respective modules, their communication interfaces and associated data models, initial results and plans in view of the final version of the platform.

The provided functionality from the NEMO Service Management layer components' namely, IBMC, MOCA, LCM and Intent-based API can be considered almost completed. The final releases of these components will be documented in D4.3 [1], due on M33, which will also describe the final integrated version of the NEMO meta-OS.

2.5.1.4 NEMO Cross-cutting Functions

With reference to the crosscutting functions, AI is vertically present in the metaOS architecture. The CFDRL component provides capacity of learning decision-making models capitalizing on the data collected by the NEMO meta-OS monitoring tool procedures offered by PPEF, CMDT and mNCC. The integration between the monitoring tools and CFDRL has been achieved allowing NEMO workload and cluster-level measurements to be digested by the CFDRL through RabbitMQ. The AI-assisted decision making that concern the workload optimal deployment and migration processes will be conducted in view of the final version of the NEMO meta-OS, as it requires the verified integration of the participating components for these workflows which has been achieved in the framework of this version (first) and is documented in section 4. The learning procedure in CFDRL combines two complementary learning paradigms: Federated Learning (FL) and Reinforcement Learning (RL). In addition, CFDRL to address privacy preservation challenges introduced FREDY (Federated Resilience Enhanced with Differential Privacy) [5] which integrates Flower with Private Aggregation of Teacher Ensembles (PATE) [6] to bolster privacy features. For the first release of the NEMO meta-OS the CFDRL component is considered as partially integrated.

Regarding Security in the NEMO meta-OS is built on the concept of ZeroTrust. NEMO Access Control (NAC) allows the implementation of a comprehensive approach to applying flexible, easily configurable, granular privileged access to NEMO resources by either internal components or, beyond the perimeter, to external entities. NAC provides a common secure API gateway for all the requests that are targeting NEMO meta-OS and offers access control based on a set of modular criteria, which may include identity management, catering for Authentication, Authorization, and Accounting (AAA). The NAC integration results in the context of the first release of the NEMO meta-OS are presented in section 4.

The NEMO meta-OS communication layer is based on RabbitMQ, a message broker enabling communication and synchronization among distributed systems and applications. It acts as an intermediary, facilitating secure message exchange while offering essential capabilities like message routing, queuing, and transformation.

Finally, the PPEF component facilitates service and resource monitoring for the NEMO meta-OS at both workload and cluster level. The PPEF concerns the metrics collection from the deployed monitoring tools, the evaluation between the collected measurements and the intents' expectation targets and the communication of this information within NEMO meta-OS (through RabbitMQ). The integration of the PPEF within NEMO meta-OS has been achieved and presented in section 4.4.

3 NEMO Service Management Layer updates

This section reports the latest specifications and design options for the NEMO components of the Service Management layer in the NEMO architecture. These components provide a middleware between core NEMO functionality and workloads, but also end users. They support ZeroOps principles and expose interfaces to external entities (services or users). Moreover, the supported services include Lifecycle Management and DLT-based accountability of workload or infrastructure usage and collectively contribute to NEMO openness and adoption by third parties, referring to application or infrastructure owners, as well as developing entities.

3.1 Intent-based Migration Controller

3.1.1 Overview

The Intent-based Migration Controller (IBMC) serves as a key component within the NEMO ecosystem, it is designed to facilitate the migration of workloads across the IoT, Edge, and Cloud Continuum. By employing intent-based networking concepts, the IBMC ensures efficient resource use, improved scalability, and continuous service availability during migrations. This approach enables the IBMC to interpret and execute high-level migration intents, which supports the flexibility needed to manage the complexities of the meta-OS environment effectively.

3.1.2 Architecture

Figure 32 illustrates a simplified high-level architecture of the components underneath the Development View of the IBMC.

ibmc-controller: Is in charge of handling the communications with other components. This communication is performed by managing different RabbitMQ[11] queues and reading/delivering the correct messages needed for each migration step.

Velero[12]: Each Velero operation is defined as a custom resource using a Kubernetes Custom Resource Definition (CRD) and is stored in *etcd*. Velero also includes controllers responsible for processing these custom resources to handle backups, restores, and related tasks. This allows to backup or restore every resource in a cluster, with the capacity of selectively filter by resource type, namespace and/or label.

S3 Storage[13]: Dedicated to store the backups created by Velero, it's a key element for the migration process. The S3 bucket is located in the main NEMO cluster and every other cluster has access to it to allow the possibility of retrieving backups from one cluster to another.

[11] http://www.rabbitmq.com/
[12] https://velero.io/
[13] https://aws.amazon.com/s3/

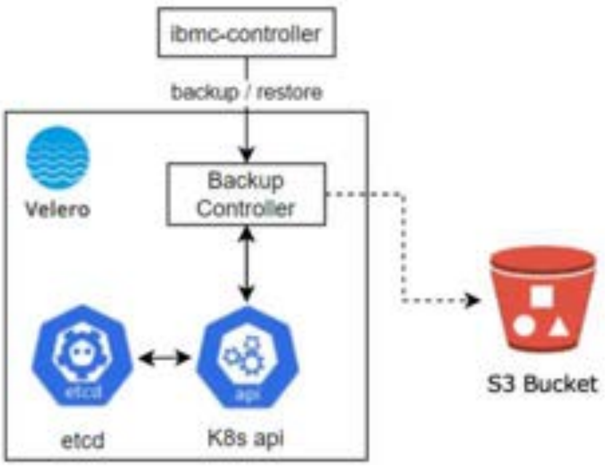


Figure 32: IBMC Simplified Architecture

Figure 33 shows a more complete architecture of the IBMC. In this figure, the migration process of a workload between two different clusters is represented, displaying the communication sequence since the migration is triggered.

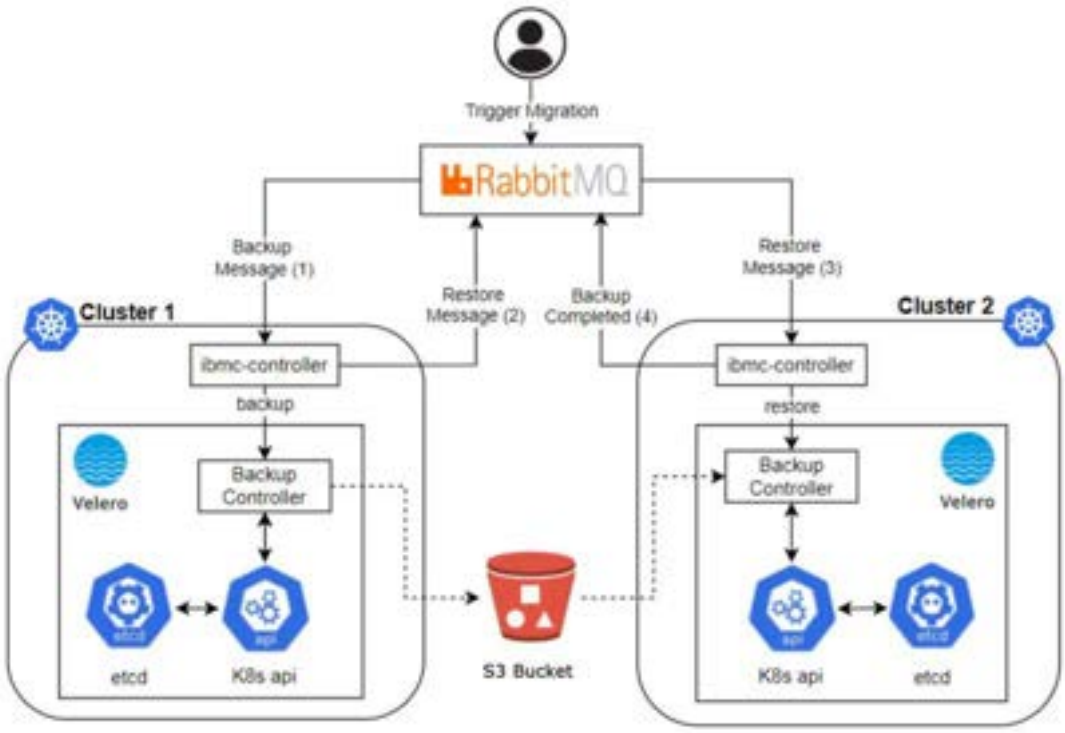


Figure 33: IBMC Complete Architecture

3.1.3 Interaction with other NEMO components

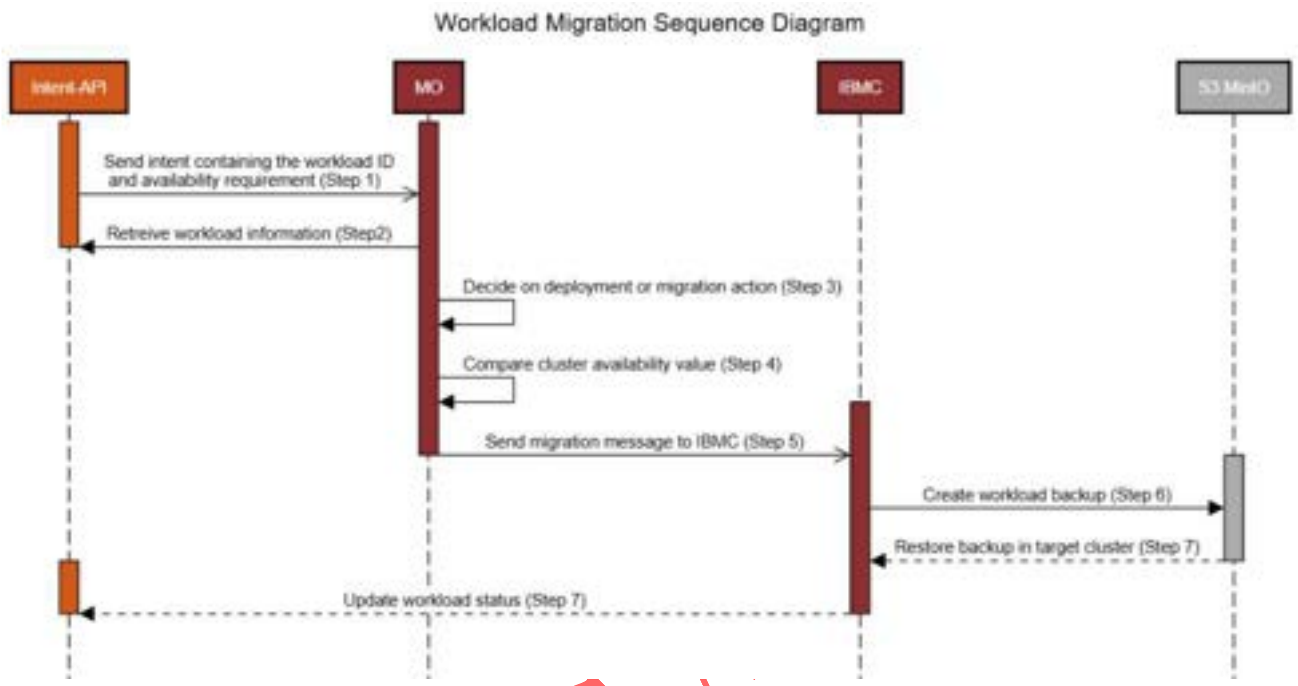

Workload Migration Sequence Diagram
Intent-API
MO
IBMC
S3 MinIO
Send intent containing the workload ID and availability requirement (Step 1)
Retreive workload information (Step2)
Decide on deployment or migration action (Step 3)
Compare cluster availability value (Step 4)
Send migration message to IBMC (Step 5)
Create workload backup (Step 6)
Restore backup in target cluster (Step 7)
Update workload status (Step 7)

Figure 34: Migration Sequence Diagram

1. The Intent-API publishes a workload intent with an availability requirement.
2. The MO retrieves workload status from the Intent-API (deployed or not deployed).
3. If the workload is already deployed in any cluster, then a migration action is triggered.
4. The cluster availability where the workload is currently deployed is compared to the one from the intent. If X<Y, the workload migration is triggered.
5. The MO sends a message via RabbitMQ queue to the IBMC containing the workload ID and a target cluster that meets the availability requirement.
6. The IBMC creates a backup of the workload associated resources and uploads it to the S3 MinIO[14] instance located in the OneLab main cluster.
7. The IBMC downloads the resources and restores them in the target cluster. After this, the workload is removed from the source cluster.
8. The IBMC sends a message to the Intent-API updating the workload status, specifying the cluster where it has been deployed.

[14] https://min.io/

3.1.3.1 Meta-Orchestrator (MO)

The meta-Orchestrator (MO) plays a central role in the IBMC's workflow. When the MO receives a migration intent, it makes a placement decision based on cluster availability. This decision is sent to the IBMC and contains the target cluster for the migration.

3.1.3.2 Intent-Based API

The Intent-Based API stores all the information related to the workloads deployed. It is responsible of creating and sending the intents that trigger the migration of a workload.

When a migration is successfully completed, a message from the IBMC is sent to the Intent-Based API to update the workload status, including the new cluster where it has been deployed.

3.1.4 Initial results

The sections below provide a summary of the results generated through the utilization of the component.

3.1.4.1 Standalone results

An initial test of the standalone IBMC component was conducted in the OneLab environment. The experiment's setup included both OneLab clusters (the main cluster and the K3s cluster) with Velero pre-installed and configured. Both clusters had access to the MinIO[15] instance deployed in the main cluster. Additionally, a workload was already deployed in the main cluster as part of the preconditions.

To simulate a migration scenario, a migration trigger was manually sent. This triggered the migration process, moving the workload from the main cluster to the K3s cluster. Upon completion, the migration was successful, with the workload fully deployed in the K3s cluster and removed from the main cluster.

3.1.4.2 Integration results

An end-to-end test was conducted involving the Intent-Based API, the Meta-Orchestrator and IBMC. The initial conditions were the same of the previous experiment, with the same objective of migrating a workload from the main cluster to the K3s cluster.

In this experiment, the process starts with the Intent-Based API posting an intent to RabbitMQ, which is read by the Meta-Orchestrator. The Meta-Orchestrator interprets the intent and verifies whether the workload ID specified in the intent is already deployed in any cluster. To obtain this information, the Meta-Orchestrator sends a request back to the Intent-Based API in order to retrieve the workload's deployment status.

If the workload is already deployed in a cluster, the Meta-Orchestrator compares the availability value of that cluster to the one specified in the intent. If the availability value of the current cluster is lower, the Meta-Orchestrator initiates a migration by posting a message to the RabbitMQ queue corresponding to the cluster where the workload is currently deployed. This message contains the information detailing the workload to be migrated (workload ID), the cluster where it is deployed and a new target cluster meeting the availability requirements.

When the IBMC controller receives this message, the migration process proceeds as in the initial experiment, resulting in the workload being successfully deployed in the K3s cluster and removed from the main cluster.

[15] https://min.io/

3.1.5 Conclusion and roadmap

The Intent-Based Migration Controller (IBMC) within the meta-OS framework represents a major leap forward for the NEMO platform, enabling smooth workload migration across the IoT-to-Edge-to-Cloud continuum while sustaining a dynamic balance within the meta-OS environment.

Looking forward, the IBMC roadmap emphasizes ongoing refinement and adaptation to meet emerging needs and technological advancements within the meta-OS landscape. Planned enhancements and future directions are outlined to ensure the IBMC continues to lead in migration capabilities as the meta-Operating System ecosystem evolves.

3.2 Plugin & Applications Lifecycle Manager

Overview The Plugin & Applications Lifecycle Manager (LCM) is a versatile mechanism designed for unified, just-in-time management of plugins and applications throughout the NEMO ecosystem. Serving as the bridge between NEMO users and the ecosystem, the LCM enables seamless deployment of workloads, such as services, applications, and plugins, within the NEMO environment. It also supports over-the-air updates and bug fixes, ensuring the system remains up to date.

While workloads are running on the NEMO meta-OS, an event-driven mechanism monitors critical performance-related events. Additionally, a security controller oversees security events, notifies users of detected anomalies, and implements mitigation measures against identified cyber threats. The LCM's user interface will integrate with other NEMO components, including the Intent-based API, PPEF, MOCA and CMDT, to provide a comprehensive and cohesive user experience. The interfaces offered include user profiles, workload management and monitoring, security monitoring, and historical analysis tailored to the user's role.

3.2.1 Architecture

The LCM comprises of a set of subcomponents namely the LCM CD, LCM Controller, Security Controller, Event-based Response, LCM Repository and LCM Dashboard.

The LCM high-level architecture of NEMO meta-OS is depicted in the development view diagram in Figure 35.

LCM CD is based on ARGO CD framework to manage NEMO workloads provided by NEMO partners or NEMO Open Call participants and deploys workloads in S3 bucket container.

LCM Controller is a control mechanism that facilitates communication between LCM submodules and the NEMO ecosystem, offering endpoints for sending and receiving information.

Security Controller handles runtime security monitoring of NEMO workloads, notifying both users and relevant NEMO components of detected events.

Event-based Response module is designed to implement automated actions in response to events initiated by user input or detected by other NEMO components.

LCM Repository is used to store data related to workload lifecycle, security incidents, detected events and other workload related information to provide historical analysis and runtime statuses.

LCM Dashboard serves as the gateway between end-users and the NEMO meta-OS ecosystem, granting privileged users access to manage their workloads and monitor both performance and security.

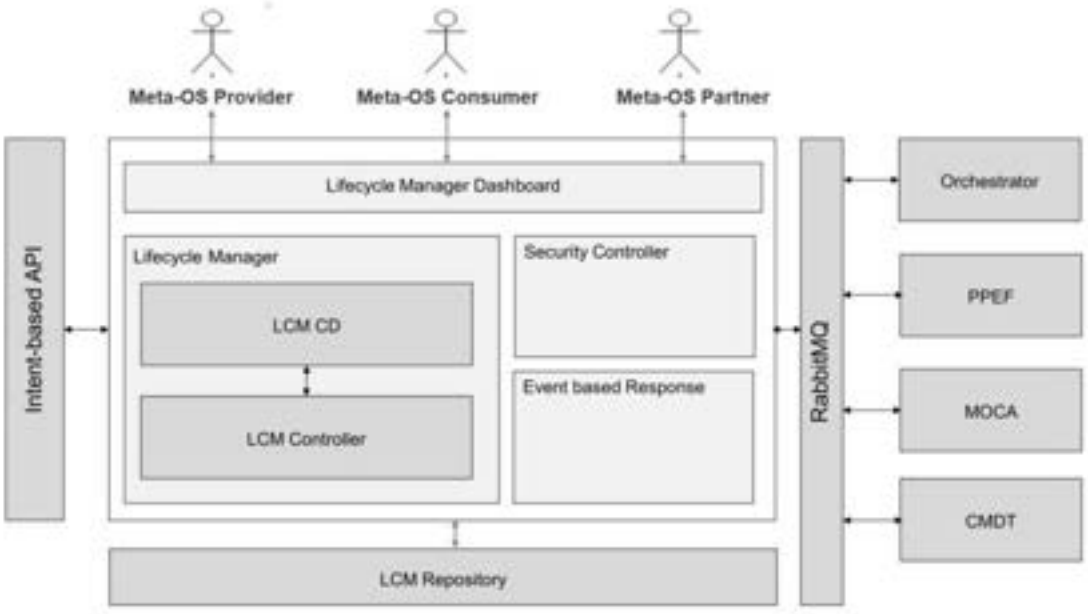


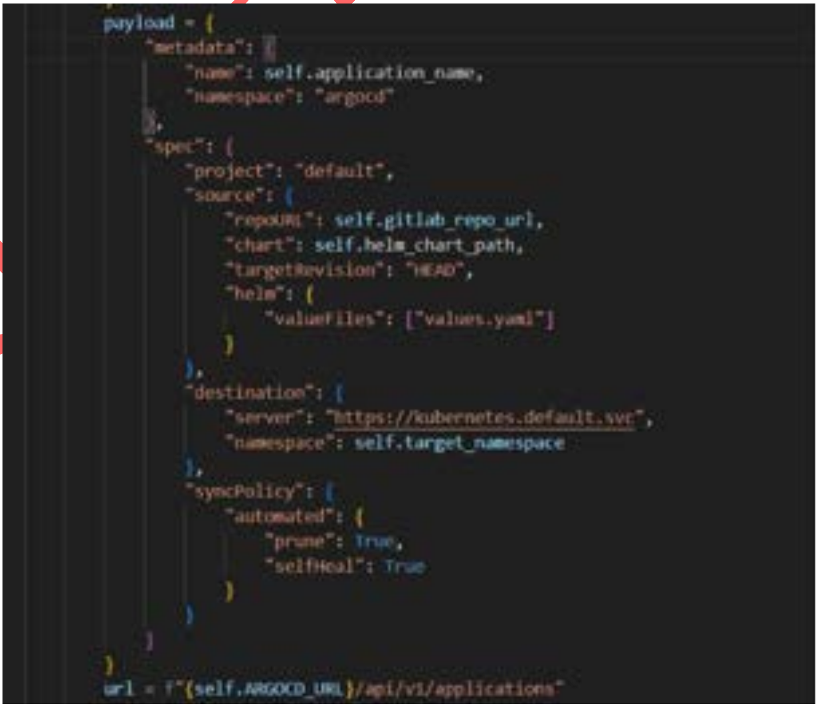
Figure 35: LCM Architecture

3.2.2 Lifecycle Manager

3.2.2.1 LCM CD

LCM CD, which corresponds to LCM Continuous Deployment, automates workloads deployment by ensuring that the state of applications in a Kubernetes cluster matches the configurations stored in Git repositories. Its key strength lies in the declarative approach to application definition, enabling users to define Kubernetes manifests in a version-controlled format.

Figure 36 provides a description of the payload transmitted while a plugin is being deployed.



```
payload = {
    "metadata": {
        "name": self.application_name,
        "namespace": "argocd"
    },
    "spec": {
        "project": "default",
        "source": {
            "repoURL": self.gitlab_repo_url,
            "chart": self.helm_chart_path,
            "targetRevision": "HEAD",
            "helm": {
                "valueFiles": ["values.yaml"]
            }
        },
        "destination": {
            "server": "https://kubernetes.default.svc",
            "namespace": self.target_namespace
        },
        "syncPolicy": {
            "automated": {
                "prune": True,
                "selfHeal": True
            }
        }
    }
}
url = f"{self.ARGOCD_URL}/api/v1/applications"
```

Figure 36: Plugin Deployment

LCM CD is based on ArgoCD tool and provides features like automated synchronization, rollback options, and support for multi-environment deployments. With its user-friendly web interface and seamless Kubernetes integration, LCM CD simplifies the deployment lifecycle, enhancing collaboration, traceability, and overall efficiency.

3.2.2.2 LCM Controller

The LCM Controller contains the logic of the LCM component and interacts with the internal subcomponents as well as the other NEMO components to retrieve information in real-time and feed the user interface while stores meaningful information in the LCM storage for keeping the historical status and performance for further analysis. The LCM controller includes functions to subscribe and consume data from relevant RabbitMQ channels, API endpoints to communicate with other components and functions to store and retrieve data from the storage repository. As the LCM Controller interacts with the other NEMO components more details on functions and APIs used will be reported in Section 4.2.3.

3.2.2.3 LCM Repository

LCM Repository uses Elasticsearch[16] for storing, searching, and analysing data provided by various NEMO components like Intent-based API, PPEF, MOCA and CMDT. Elasticsearch provides fast search responses and comes with extensive REST APIs for storing and searching the data. Stored data include the status and lifecycle of the workloads, security events detected, workload performance and resource usage.

3.2.2.4 Security Controller

The Security Controller caters for security monitoring at runtime regarding NEMO workloads and alerts both the user and relevant NEMO components for detected events. This component aims to complement the security validation checks made before deployment of workloads into NEMO clusters, such as scanning processes in the Continuous Integration workflow or in the images registries, as these validation checks take place prior to the containers' deployment and even block some deployments as a result of failing the security assessment. The Security Controller aims to identify security incidents which take place at containers' runtime and may refer either to events at the system call level or to vulnerabilities arising from software dependencies, known vulnerabilities and insufficient security configurations.

Falco[17] framework was selected as the foundation for the development of the plugin that is available through Security Controller. Falco is an open-source, CNCF adopted, runtime security platform that allows you to detect and respond to suspicious behavior within containers and applications. Falco is deployed in OneLab premises, as illustrated in section 2.4.

Falco continuously monitors the deployed containers and generates security auditing events that are digested by the Security Controller and are handled by the LCM. The Security Controller is responsible for the filtration of security events and subsequently their mapping with deployed workloads that correspond to a NEMO user.

3.2.2.5 LCM Visualization

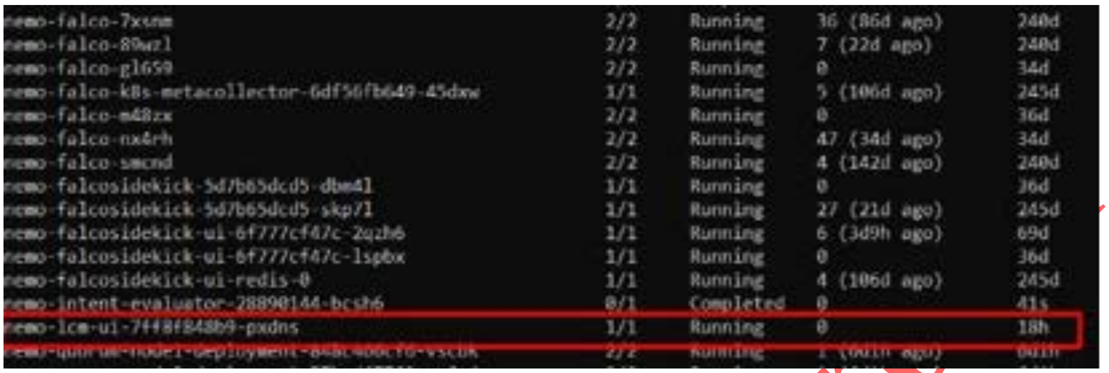
LCM visualization is the main interface of NEMO project providing the necessary interfaces for each NEMO user role to manage workloads and resources in NEMO meta-OS. The LCM visualizations aim to provide interfaces for seamless user experience with NEMO ecosystem available to experts and less experienced users. The target is to provide the relevant information for workloads and resources lifecycle, usage, and security in a compact format at different levels of detail (workload, resources, user,

[16] https://elastic.co/
[17] https://www.falcoframework.com/

system). Indicative screenshots are presented in section 3.2.4 containing the initial results of the LCM component based on the implementation and integration progress at the current phase.

3.2.3 Interaction with other NEMO components

The LCM component is deployed in the NEMO Onelab infrastructure, as shown in Figure 37.



```
nemo-falco-7xsnm                                      2/2   Running     36 (86d ago)    240d
nemo-falco-89wzl                                      2/2   Running     7 (22d ago)     240d
nemo-falco-gl659                                      2/2   Running     0               34d
nemo-falco-k8s-metacollector-6df56fb649-45dxw         1/1   Running     5 (106d ago)    245d
nemo-falco-m48zx                                      2/2   Running     0               36d
nemo-falco-nx4rh                                      2/2   Running     47 (34d ago)    34d
nemo-falco-smcnd                                      2/2   Running     4 (142d ago)    240d
nemo-falcosidekick-5d7b65dcd5-dbm41                   1/1   Running     0               36d
nemo-falcosidekick-5d7b65dcd5-skp71                   1/1   Running     27 (21d ago)    245d
nemo-falcosidekick-ui-6f777cf47c-2qzh6                1/1   Running     6 (3d9h ago)    69d
nemo-falcosidekick-ui-6f777cf47c-1spbx                1/1   Running     0               36d
nemo-falcosidekick-ui-redis-0                         1/1   Running     4 (106d ago)    245d
nemo-intent-evaluator-28890144-bcsh6                  0/1   Completed   0               41s
nemo-lcm-ui-7ff8f848b9-pxdns                          1/1   Running     0               18h
nemo-quorum-node1-deployment-848c406cfd-vscbk         2/2   Running     1 (60m ago)     60m
```

Figure 37: Onelab deployment LCM and Security Controller

The LCM interacts with Intent-based API and MOCA components through relevant API endpoints and consumes data from PPEF, CMDT and Security Controller through RabbitMQ exchanges. This section describes in more detail the functions and data models used to interact with other NEMO components. Figure 38 depicts the data model of the Intent-based API endpoints which are used for the management of the workloads.

```
GET /workload/
         {
                name: string,
                 version: string,
         }
                  -
POST /workload/
         {
                name: string,
                 version: string,
                schema:{},
                type: string (chart),
                intents: Array [],
         }

POST /workload/upload/
         {
                  file: file (tgz helm chart),
                  name: string,
                  version: string
         }
POST /workload/{id}/template/
         {
                release_ name: string,
                namespace: string,
                values_override:{},
                include_crds:boolean,
                is_upgrade:boolean,
                no_hooks:boolean,
                ingress_enabled:boolean,
                intents:[
                  {
                           intent_type,
                           service_start_time,
                           service_end_time,
                           targets:[
                                    {
```

```
                                                target_name,
                                                target_condition,
                                                target_value_range
                                        }
                                ]
                        }
                ]
        }

GET /workload/instance/
        {
                instance_id: string
        }
```

Figure 38: Intent-based API workload management

Figure 39 presents the data model of the Intent-based API endpoints which are used for the management of the intents.

```
GET /intent/
        {
                intent_id:string
        }
                -
POST /intent/template/
        {
                instance_id: string,
                intent_type: string,
                service_start_time: string,
                service_end_time: string,
                targets:[
                  {
                          target_name: string,
                          target_condition: string,
                          target_value_range: string
                  }
                ]
         }
```

Figure 39: Intent-based API intents management

Figure 40 presents the data model of the MOCA API for the resources provisioning.

```
GET /moca/cluster/retrieve
        {
        }
                -
POST /moca/cluster/register
        {
                  cluster_name: string,
                  cpus: number,
                  memory: number,
                  storage: number,
                  availability: string,
                  green_energy: string,
                  cost: string,
                  cpu_base_rate: number,
                  memory_base_rate: number
        }
```

Figure 40: MOCA Resource provisioning

Additionally, the LCM subscribes to RabbitMQ exchanges to retrieve real-time data from PPEF, Security Controller and CMDT.

In summary, the retrieved information includes cluster usage metrics from PPEF (CPU, RAM, and storage, etc.), security events identified by the Security Controller (both system-wide and per workload), and data from CMDT, which currently encompasses the number of workload replicas and network related information including workload's response times. Finally, data consumed from RabbitMQ or API endpoints are stored in LCM internal repository for historical overview and detailed analysis.

Figure 41 depicts functions to retrieve data to/from LCM Repository.

```
GET /cluster/{id}/data/
         {
                  timestamp_from: string,
                  timestamp_to: string,
                  cluster_id: string
         }
GET /workload_instance/{id}/data
         {
                  timestamp_from: string,
                  timestamp_to: string,
                  workload_id: string
         }
GET /workload_security_events/{id}/data
         {
                  timestamp_from: string,
                  timestamp_to: string,
                  workload_id: string
         }
GET /workload_CMDT/{id}/data
         {
                  timestamp_from: string,
                  timestamp_to: string,
                  workload_id: string
         }
```

Figure 41: Searching LCM Repository

3.2.4 Initial results

The LCM provides interfaces for different services such as *Plugin Monitoring*, *Workload Monitoring*, *Intent Management* and *Resource Provisioning*. Figure 42 illustrates the homepage of LCM dashboard.

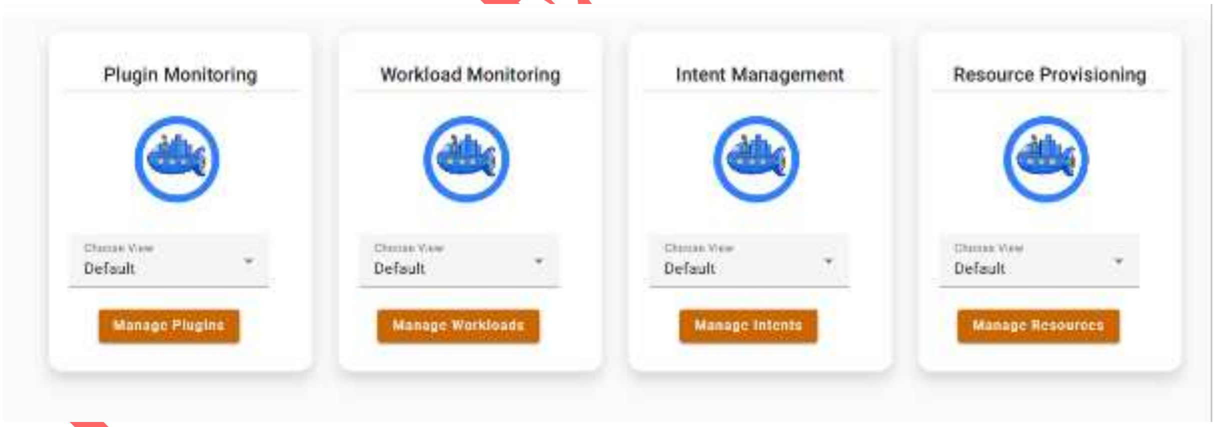


Figure 42: LCM Dashboard Homepage

Plugin Monitoring offers a CI/CD process and lifecycle monitoring for NEMO plugins and applications. Currently, the user is able to deploy a plugin and manage already deployed plugins while monitoring basic lifecycle parameters like versioning and activity are also available.

The *Workload Monitoring* section includes functionality to manage workloads in their whole lifecycle, from registering to Intent-based API to deployment and running several instances according to the user role and credentials. More details on the LCM UI available views are presented in Section 4 which concerns the NEMO scenario-driven integration and verification results.

3.2.5 Conclusion and roadmap

The achievements of LCM component, considering developments that are deployed in OneLab environment, fully integrated in NEMO meta-OS include:

- Workload management and monitoring
- Resources provisioning and monitoring
- Plugin deployment and lifecycle management
- Security/vulnerability scanning and monitoring

Towards the NEMO final version release, our goal is to deliver a comprehensive tool for managing and monitoring workload and resource lifecycle. This tool will consolidate information from various NEMO components into a streamlined format, ensuring a seamless experience for both expert and non-expert users within the NEMO ecosystem.

3.3 Monetization and Consensus-based Accountability

3.3.1 Overview

The Monetization and Consensus-based Accountability (MOCA) component enables the fair and secure monetization of the NEMO platform among the different users participating (consumers or providers). MOCA creates a distributed, tamper-resistant blockchain based ledger between different operators and verticals to track provenance and enforce secure negotiation and transaction of resources such as through smart contracts. The MOCA system provides the users with *"credits"* - the accountability unit of the platform. Their purpose is to reward the users who contribute to the platform by either providing infrastructure or deploying their services and allow them to accelerate precommercial exploitation of multi-tenant AIoT-Edge-Cloud continuum. The usage of DLT-based smart contracts for computing the accounting tasks allows for transparency and immutability in the transactions.

In retrospect, MOCA encompasses the following features:

- It uses blockchain technologies (more specifically Quorum) to perform the accounting actions and to transmit the results. This allows for immutability and traceability for all actions.
- It is integrated with NEMO's Authentication platform; therefore, the users have role-based capabilities.
- It gives periodic but also real-time reports of the users' information, like their bill details and the state of their resources (clusters, workloads).
- The accounting process takes into consideration the amount of the offered resources, the region demands and the infrastructure type to properly calculate the costs and rewards of each user.

3.3.2 Architecture

MOCA comprises of the following components:

- An **Event Server** that allows other components and users to retrieve information on the details of the registered resources (clusters and workloads) and the accounting events.
- The **Decentralized Applications (DApps),** which contain the accounting logic and store information like the IPFS links to the cluster configuration files and the NEMO resources' information.
- The **Smart Contracts component** (private Quorum blockchain), where DApps are deployed, and the transactions and calculations take place.
- An **IPFS** network, where the cluster configuration files are stored. Like Quorum, IPFS offers immutability to the data and detection of malicious attacks.

Since the full description of the sub-components' functionality is provided in deliverable D4.1, the basic workflow will be presented, briefly. Figure 43 shows the sub-components that are part of the MOCA component. The *Event Server* is the main communication interface with the rest of the MOCA components which is responsible for handling (a) the requests for the cluster registration, (b) the communication with the *IPFS* for storing the cluster configuration files and (c) the exposure of the functionality of the *DApps* through REST API endpoints. The MOCA in periodic basis computes the resource usage of the deployed workloads and the usability status of the registered clusters. These periodic tasks communicate with the DApps and update appropriately the users' information held by the *Event Server*. A closer inspection on the calculation details is delivered in Section 3.3.4 and a full example of the workflow in Section 4.

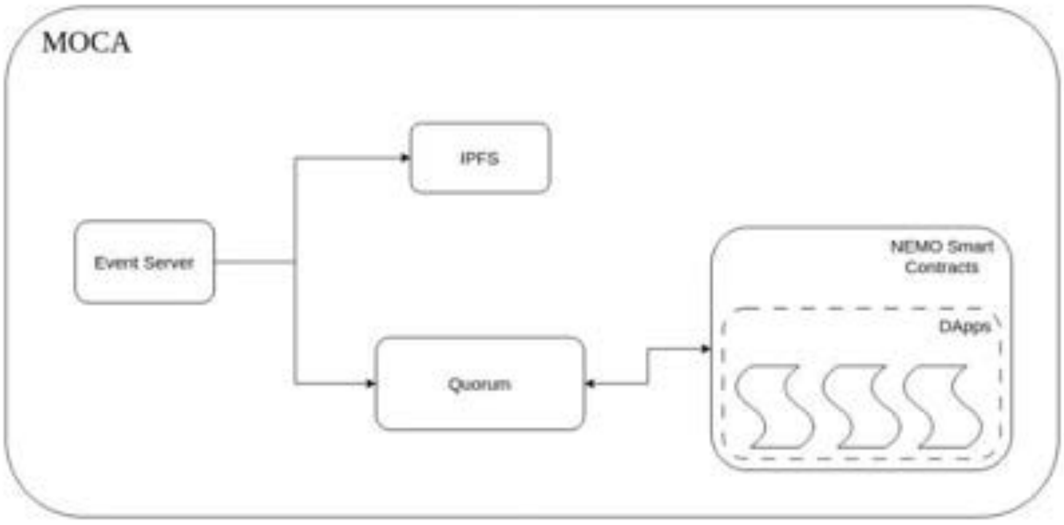
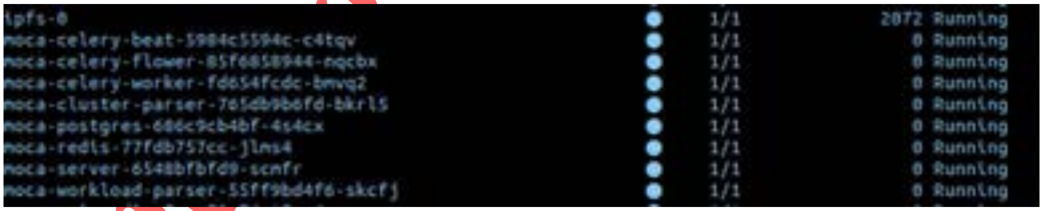


Figure 43: MOCA diagram

3.3.3 Interaction with other NEMO components

The MOCA component is deployed in the NEMO OneLab premises, as shown in Figure 44.



```
ipfs-0                                        ●   1/1        2072 Running
moca-celery-beat-5984c5594c-c4tqv             ●   1/1           0 Running
moca-celery-flower-85f6858944-nqcbx           ●   1/1           0 Running
moca-celery-worker-fd654fcdc-bmvq2            ●   1/1           0 Running
moca-cluster-parser-765db9b6fd-bkrl5          ●   1/1           0 Running
moca-postgres-686c9cb4bf-4s4cx                ●   1/1           0 Running
moca-redis-77fdb757cc-jlms4                   ●   1/1           0 Running
moca-server-6548bfbfd9-scnfr                  ●   1/1           0 Running
moca-workload-parser-55ff9bd4f6-skcfj         ●   1/1           0 Running
```

Figure 44: The MOCA deployment in the Onelab cluster

Figure 45 demonstrates the interactions of MOCA with the rest of the NEMO components. More specifically, MOCA integrates directly with the NEMO Intent API and NEMO RabbitMQ. Other NEMO components (LCM, PPEF, CMDT) can go through the Intent API, to access the MOCA Event Server endpoints. The RabbitMQ integration establishes the connection between MOCA and the NEMO Meta-Orchestrator. During the cluster registration, the two components exchange though the RabbitMQ the appropriate information to complete the action.

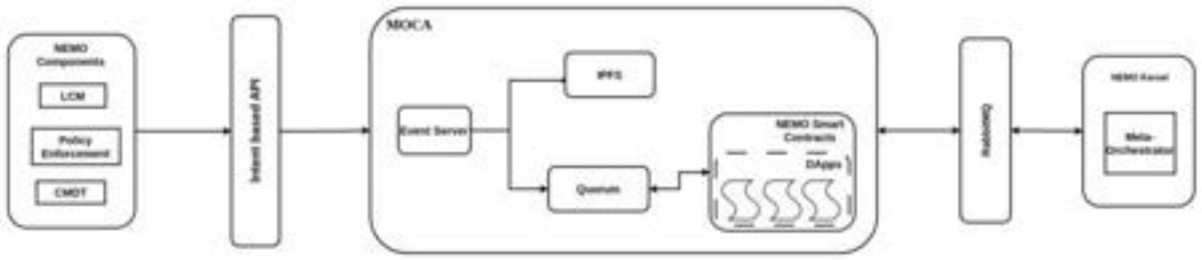
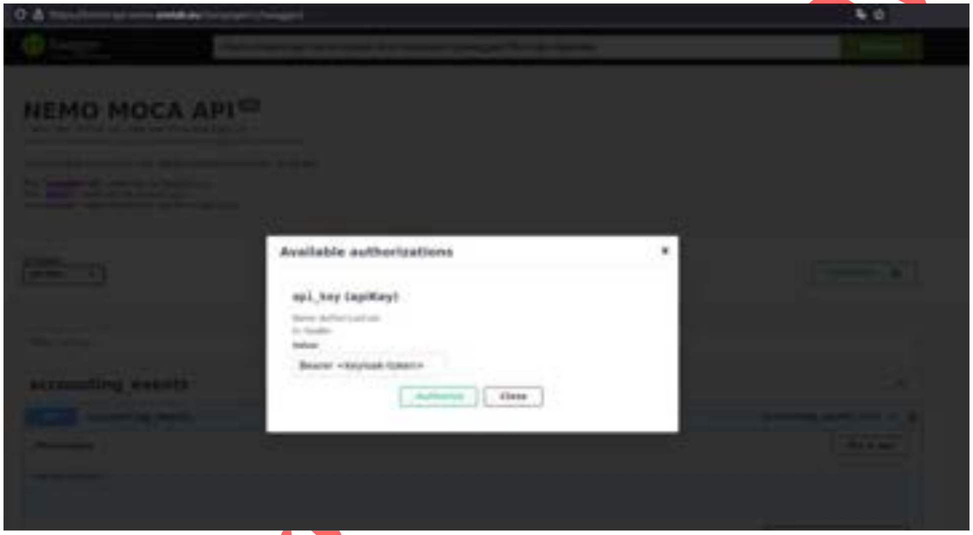


Figure 45: MOCA integration diagram

The following sections give a more thorough look on the MOCA API and the capabilities it offers.

Note: All the endpoints require to be authorized with the use of an authorization header, which sends a token provided from the *NEMO Access Control plugin*, as shown in the demonstration of Figure 46.


NEMO MOCA API

Available authorizations

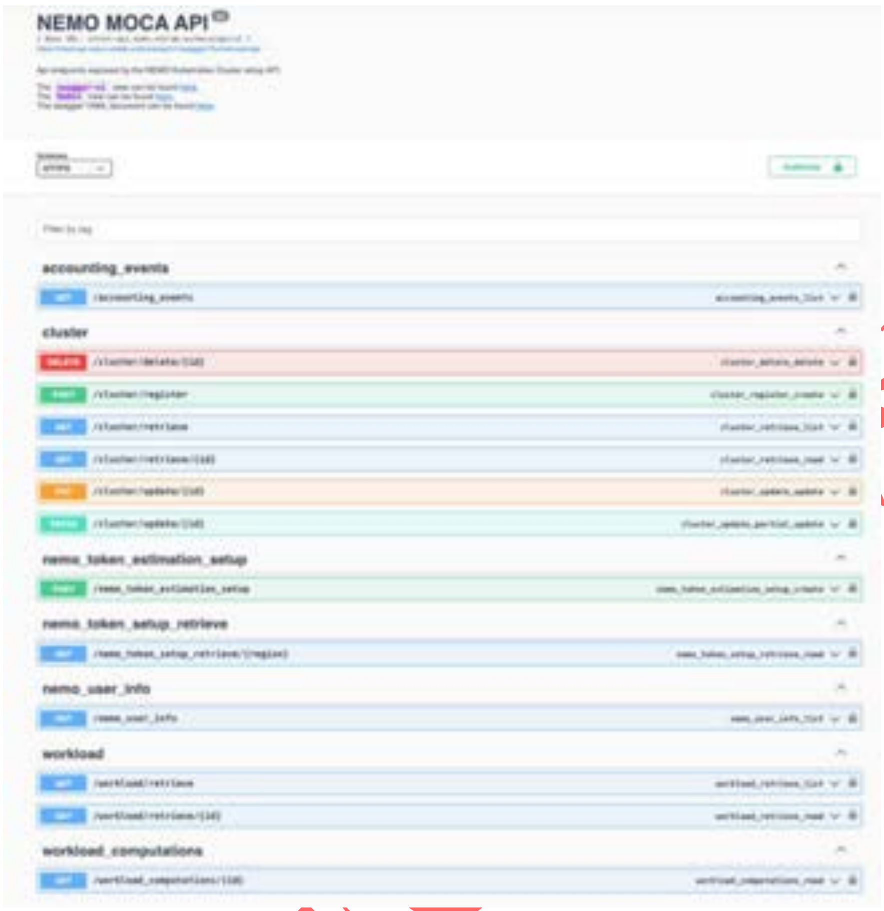
api_key (apiKey)

Bearer <keycloak-token>

Figure 46: MOCA API authorization example

3.3.3.1 MOCA API

This section presents the MOCA API that is exposed and is available in openAPI format (https://intent-api.nemo.onelab.eu/moca/api/v1/swagger/). Figure 47, illustrates the relevant contents. The complete functionality offered by the MOCA API (endpoints and data models) is described in detail in ANNEX A.



NEMO MOCA API

[illegible]

Filter by tag

accounting_events

GET	/accounting_events	accounting_events_list

cluster

DELETE	/cluster/delete/{id}	cluster_delete_delete
POST	/cluster/register	cluster_register_create
GET	/cluster/retrieve	cluster_retrieve_list
GET	/cluster/retrieve/{id}	cluster_retrieve_read
PUT	/cluster/update/{id}	cluster_update_update
PATCH	/cluster/update/{id}	cluster_update_partial_update

nemo_token_estimation_setup

POST	/nemo_token_estimation_setup	nemo_token_estimation_setup_create

nemo_token_setup_retrieve

GET	/nemo_token_setup_retrieve/{region}	nemo_token_setup_retrieve_read

nemo_user_info

GET	/nemo_user_info	nemo_user_info_list

workload

GET	/workload/retrieve	workload_retrieve_list
GET	/workload/retrieve/{id}	workload_retrieve_read

workload_computations

GET	/workload_computations/{id}	workload_computations_read

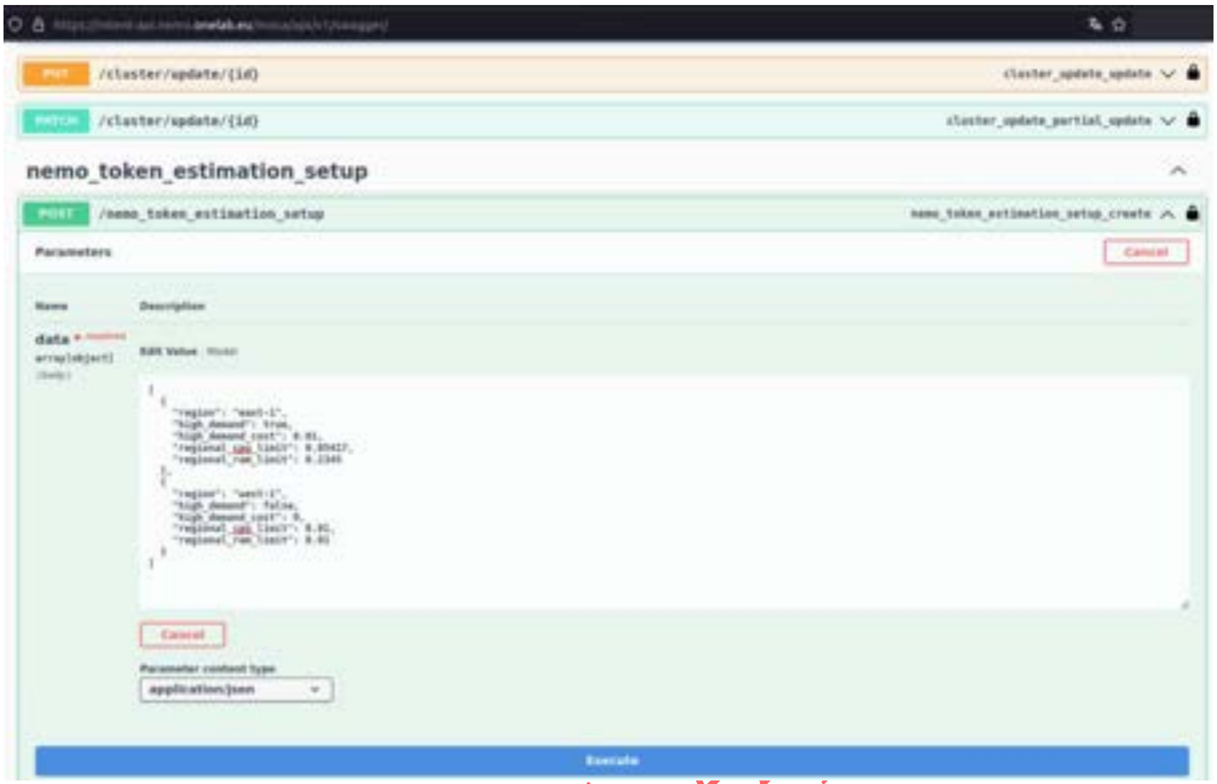
Figure 47: MOCA API

3.3.4 Initial result

MOCA uses various smart contracts to perform the accounting of the NEMO platform. In first version four contracts are supported namely the (a) *NemoTokenEstimation,* (b) *NemoFunds,* (c) *InfrastructureOwnerModel* and (d) *ServiceProviderModel.* In the following sections, a more thorough analysis of their functionality will be presented to better understand the calculation mechanism.

3.3.4.1 Regional Costs

The contact *NemoTokenEstimation* is responsible for supplying the costs for all the registered clusters to NEMO. A cluster can be categorized by whether it is in high/low demand, and its usability status in terms of available resources (CPU, RAM, Network bandwidth, etc.). The contract stores that information and allows for the retrieval of the details via the MOCA Event Server with the use of the */nemo_token_estimation_setup* endpoint (Figure 48).



```
PUT    /cluster/update/{id}                                   cluster_update_update
PATCH  /cluster/update/{id}                                   cluster_update_partial_update

nemo_token_estimation_setup

POST   /nemo_token_estimation_setup                           nemo_token_estimation_setup_create

Parameters                                                                          Cancel

Name              Description
data * required
array[object]     Edit Value | Model
(body)
[
  {
    "region": "east-1",
    "high_demand": true,
    "high_demand_cost": 0.01,
    "regional_cpu_limit": 0.85417,
    "regional_ram_limit": 0.2346
  },
  {
    "region": "west-1",
    "high_demand": false,
    "high_demand_cost": 0,
    "regional_cpu_limit": 0.01,
    "regional_ram_limit": 0.01
  }
]

Cancel

Parameter content type
application/json

Execute
```

Figure 48: Setup region information through Event Server

An example of a successful cluster registering information is shown in Figure 49.

```
[illegible]                                                                          Debug
status                 [illegible]
transaction hash       [illegible]
block hash             [illegible]
block number           [illegible]
from                   [illegible]
to                     [illegible]
gas                    [illegible] gas
transaction cost       [illegible] gas
execution cost         [illegible] gas
input                  [illegible]
output                 0x
decoded input          {
                           "string[] _region": [
                               "east-1",
                               "west-1"
                           ],
                           "bool[] _highDemand": [
                               true,
                               false
                           ],
                           "uint256[] _highDemandCost": [
                               [illegible]
                               "0"
                           ],
                           "uint256[] _regionalCpuLimit": [
                               [illegible]
                           ],
                           "uint256[] _regionalMemoryLimit": [
                               [illegible]
                           ]
```

Figure 49: Logs of inserting cluster information

Table 5 shows the details of the contract.

NemoTokenEstimationSetupContract.sol

```
pragma solidity ^0.8.0;

contract NemoTokenEstimationSetupContract {
    address private _owner;

    struct RegionInfo {
        bool isSet;
        bool highDemand;
        uint256 highDemandCost;
        uint256 regionalCpuCosts;
        uint256 regionalMemoryCosts;
    }

    mapping(string => RegionInfo) public regionalInfoMapping;
    modifier onlyOwner() {
        require(msg.sender == _owner, "Caller is not owner!");
        _;
    }
    modifier validateRegionInfo(
        uint256 _regionLength,
        uint256 _highDemandLength,
        uint256 _highDemandCost,
        uint256 _regionalCpuCostsLength,
        uint256 _regionalMemoryCostsLength
    ) {
        require(
            _regionLength == _highDemandLength &&
                _regionLength == _highDemandCost &&
                _regionLength == _regionalCpuCostsLength &&
                _regionLength == _regionalMemoryCostsLength,
            "Input array lengths must match"
        );
        _;
    }
    constructor() {
        _owner = msg.sender;
    }
    function initializeNemoTokenEstimationInfo(
        string[] memory _region,
        bool[] memory _highDemand,
        uint256[] memory _highDemandCost,
        uint256[] memory _regionalCpuCosts,
        uint256[] memory _regionalMemoryCosts
    )
        public
        onlyOwner
        validateRegionInfo(
            _region.length,
            _highDemand.length,
            _highDemandCost.length,
            _regionalCpuCosts.length,
            _regionalMemoryCosts.length
        )
    {
        for (uint256 i = 0; i < _region.length; i++) {
            regionalInfoMapping[_region[i]] = RegionInfo({
                isSet: true,
                highDemand: _highDemand[i],
                highDemandCost: _highDemandCost[i],
                regionalCpuCosts: _regionalCpuCosts[i],
                regionalMemoryCosts: _regionalMemoryCosts[i]
            });
        }
    }
    function isRegionSet(string memory _region) public view returns (bool) {
        RegionInfo memory info = regionalInfoMapping[_region];
        return info.isSet;
    }
    function getRegionInfo(
        string memory _region
    ) public view returns (bool, uint256, uint256, uint256) {
        RegionInfo storage info = regionalInfoMapping[_region];
        return (
            info.highDemand,
            info.highDemandCost,
            info.regionalCpuCosts,
            info.regionalMemoryCosts
        );
    }
}
```

Table 5: The NemoTokenEstimation smart contract details

3.3.4.2 Handling transactions of clusters and workflows

The *NemoFunds* contract is responsible for keeping track of registered clusters and workflows, storing and emitting the transactions taking place and tracking the tokens available for every entity. When the usage of a workload is computed, the *NemoFunds* contract makes sure to appropriately change the balance of the actors involved (clusters, workloads, NEMO platform). Then, the changes become known to the Event Server though the use of events. Table 6 shows the details of the contract.

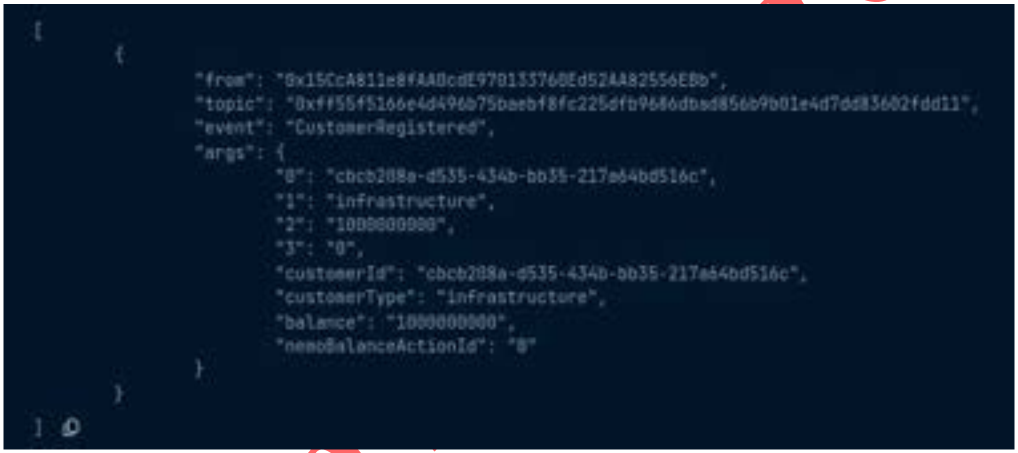
NemoFunds.sol

```
pragma solidity ^0.8.0;

contract NemoFunds {
    address public owner;
    uint256 public nemoTotalBalance;
    uint256 public nemoActionsCounter;
    uint256 public nemoRate;

    enum TransactionType {
        deposit,
        withdrawal
    }
    struct NemoBalanceInfo {
        string customerId;
        uint256 customerTokens;
        TransactionType transactionType;
    }

    mapping(uint256 => NemoBalanceInfo) public nemoBalanceActions;
    mapping(string => uint256) public customersBalance;
    mapping(string => bool) public registeredCustomers;

    event CustomerRegistered(
        string customerId,
        string customerType,
        uint256 balance,
        uint256 nemoBalanceActionId
    );
    event DepositTokens(
        string customerId,
        uint256 tokens,
        uint256 balance,
        uint256 nemoBalanceActionId
    );
    event WithdrawTokens(
        string customerId,
        uint256 tokens,
        uint256 balance,
        uint256 nemoBalanceActionId
    );

    modifier onlyOwner() {
        require(msg.sender == owner, "Caller is not the owner");
        _;
    }

    constructor() {
        nemoTotalBalance = 10000000000;
        nemoActionsCounter = 0;
        nemoRate = 20000;
    }

    function isCustomerRegistered(
        string memory customerId
    ) public view returns (bool) {
        return registeredCustomers[customerId];
    }

    function registerCustomer(
        string memory customerId,
        string memory _identifier
    ) public {
        require(
            !isCustomerRegistered(customerId),
            "The customer is already registered!"
        );

        if (
            keccak256(abi.encode(_identifier)) ==
            keccak256(abi.encode("ServiceProvider"))
        ) {
            registerServiceProvider(customerId);
        } else if (
            keccak256(abi.encode(_identifier)) ==
            keccak256(abi.encode("InfrastructureOwner"))
        ) {
            registerInfrastructureOwner(customerId);
        }
    }

    function registerServiceProvider(string memory customerId) private {
        registeredCustomers[customerId] = true;
        customersBalance[customerId] = 500000000;
        nemoBalanceActions[nemoActionsCounter] = NemoBalanceInfo({
            customerId: customerId,
            customerTokens: customersBalance[customerId],
            transactionType: TransactionType.deposit
        });

        emit CustomerRegistered(
            customerId,
            "service",
            customersBalance[customerId],
            nemoActionsCounter
        );

        nemoActionsCounter++;
    }

    function registerInfrastructureOwner(string memory customerId) private {
        registeredCustomers[customerId] = true;
        customersBalance[customerId] = 1000000000;
        nemoBalanceActions[nemoActionsCounter] = NemoBalanceInfo({
            customerId: customerId,
            customerTokens: customersBalance[customerId],
            transactionType: TransactionType.deposit
        });

        emit CustomerRegistered(
            customerId,
            "infrastructure",
            customersBalance[customerId],
            nemoActionsCounter
        );

        nemoActionsCounter++;
    }

    function depositTokens(string memory customerId, uint256 tokens) public {
        require(nemoTotalBalance > tokens, "NEMO is out of funds!!!");

        nemoTotalBalance -= tokens;

        customersBalance[customerId] += tokens;

        nemoBalanceActions[nemoActionsCounter] = NemoBalanceInfo({
            customerId: customerId,
            customerTokens: customersBalance[customerId],
            transactionType: TransactionType.deposit
        });

        emit DepositTokens(
            customerId,
            tokens,
            customersBalance[customerId],
            nemoActionsCounter
        );

        nemoActionsCounter++;
    }

    function withdrawTokens(string memory customerId, uint256 tokens) public {
        require(
            customersBalance[customerId] > tokens,
            "Customer is out of funds!!!"
        );

        customersBalance[customerId] -= tokens;

        nemoTotalBalance += tokens;

        nemoBalanceActions[nemoActionsCounter] = NemoBalanceInfo({
            customerId: customerId,
            customerTokens: customersBalance[customerId],
            transactionType: TransactionType.deposit
        });
        emit WithdrawTokens(
            customerId,
            tokens,
            customersBalance[customerId],
            nemoActionsCounter
        );

        nemoActionsCounter++;
    }

    function nemoPayment(string memory customerId) public {
        require(
            customersBalance[customerId] > nemoRate,
            "Customer is out of funds!!!"
        );

        uint256 nemoPaymentFee = (customersBalance[customerId] * nemoRate) /
            10 ** 8;

        customersBalance[customerId] -= nemoPaymentFee;

        nemoTotalBalance += nemoPaymentFee;

        nemoBalanceActions[nemoActionsCounter] = NemoBalanceInfo({
            customerId: customerId,
            customerTokens: customersBalance[customerId],
            transactionType: TransactionType.withdrawal
        });

        emit WithdrawTokens(
            customerId,
            nemoPaymentFee,
            customersBalance[customerId],
            nemoActionsCounter
        );

        nemoActionsCounter++;
    }

    function getNemoBalance() public view returns (uint256) {
        return nemoTotalBalance;
    }

    function makeTransaction(
        string memory serviceId,
        string memory clusterId,
        uint256 _tokens
    ) public {
        withdrawTokens(serviceId, _tokens);
        depositTokens(clusterId, _tokens);
        nemoPayment(clusterId);
    }
}
```

Table 6: The NemoFunds smart contract details

3.3.4.3 Cluster provision

The contract *InfrastructureOwnerModel* is responsible for registering the clusters joining NEMO. It communicates with the *NemoFunds* contract to complete the registration. The process provides the cluster with 10 tokens as an initialization sum. An example of the result of the registration of the cluster from the blockchain's side can be seen in Figure 50, where the transaction logs show the successful registration of the cluster. The logs show that an event was emitted (the *CustomerRegistered* event), to inform the Event Server of the action. It should be noted here that all the arithmetic values presented are normalized (multiplied with a constant variable 10^8), since Solidity[18], the programming language for the smart contacts, cannot handle float values. Therefore, all the values are multiplied with a big enough constant to avoid issues with any float numbers.



```
[
        {
                "from": "0x15CcA811e8fAA0cdE978133760Ed524A82556EBb",
                "topic": "0xff55f5166e4d496b75baebf8fc225dfb9686dbad856b9b01e4d7dd83602fdd11",
                "event": "CustomerRegistered",
                "args": {
                        "0": "cbcb208a-d535-434b-bb35-217a64bd516c",
                        "1": "infrastructure",
                        "2": "1000000000",
                        "3": "0",
                        "customerId": "cbcb208a-d535-434b-bb35-217a64bd516c",
                        "customerType": "infrastructure",
                        "balance": "1000000000",
                        "nemoBalanceActionId": "0"
                }
        }
]
```

Figure 50: Example of the transaction logs of the cluster registration

A more thorough example of how the cluster registration is performed will be presented in Section 4 of the deliverable. Table 7 shows the details of the contract.

```
InfrastructureOwnerModel.sol
pragma solidity ^0.8.0;
import "./NemoFunds.sol";

contract InfrastructureOwnerModel {
    NemoFunds public nemoFunds;

    struct InfrastructureInfo {
        string cluster_name;
        uint256 totalCpu;
        uint256 totalMemory;
        uint256 totalDisk;
        string ipfsLink;
```

[18] https://soliditylang.org/

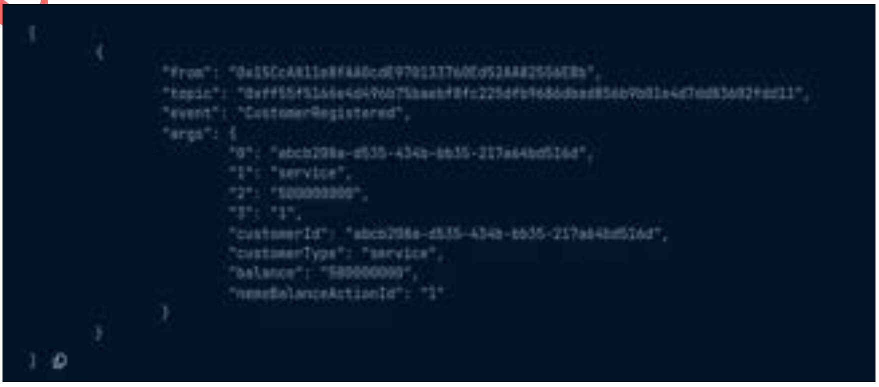
```
        string availability;
        string green_energy;
        string cost;
        uint256 cpu_base_rate;
        uint256 memory_base_rate;
    }

    mapping(string => InfrastructureInfo) infrastructureInfo;

    constructor(address _nemoFundsAddress) {
        nemoFunds = NemoFunds(_nemoFundsAddress);
    }

    function register(
        string memory clusterId,
        InfrastructureInfo memory info
    ) public {
        require(
            !nemoFunds.isCustomerRegistered(clusterId),
            "The customer is already registered!"
        );
        string memory _identifier = "InfrastructureOwner";
        nemoFunds.registerCustomer(clusterId, _identifier);

        infrastructureInfo[clusterId] = InfrastructureInfo({
            cluster_name: info.cluster_name,
            totalCpu: info.totalCpu,
            totalMemory: info.totalMemory,
            totalDisk: info.totalDisk,
            ipfsLink: info.ipfsLink,
            availability: info.availability,
            green_energy: info.green_energy,
            cost: info.cost,
            cpu_base_rate: info.cpu_base_rate,
            memory_base_rate: info.memory_base_rate
        });
    }

    function getInfrastructureInfo(
        string memory clusterId
    )
        public
        view
        returns (
            string memory,
            uint256,
            uint256,
            uint256,
            string memory,
            string memory,
            string memory,
            string memory,
            uint256,
            uint256
        )
    {
        require(
            nemoFunds.isCustomerRegistered(clusterId),
            "The customer is not registered!"
        );

        InfrastructureInfo storage info = infrastructureInfo[clusterId];

        return (
            info.cluster_name,
            info.totalCpu,
            info.totalMemory,
            info.totalDisk,
            info.ipfsLink,
            info.availability,
            info.green_energy,
            info.cost,
            info.cpu_base_rate,
            info.memory_base_rate
        );
    }
}
```

Table 7: The InfrastructureOwnerModel smart contract

3.3.4.4 Workload provision and usage calculation

The *ServiceProviderModel* contract is responsible for registering the NEMO workloads joining NEMO and calculating their impact on the cluster resources. It communicates with the *NemoFunds* contract to complete the registration and the *NemoTokenEstimation* to retrieve the costs associated with the regions. The registration process provides the workload with 5 tokens as an initialization sum.

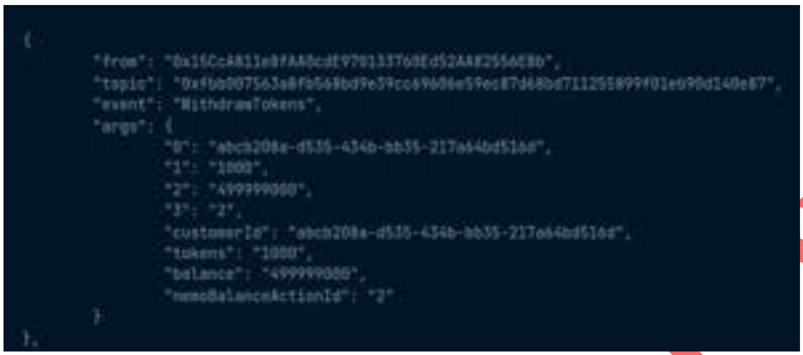
An example transaction is available in Figure 51, where its logs show the emitted event with the registration's info.



```
[
    {
        "from": "[illegible]",
        "topic": "[illegible]",
        "event": "CustomerRegistered",
        "args": {
            "0": "abcb298a-d535-434b-bb35-217a64bd516d",
            "1": "service",
            "2": "500000000",
            "3": "1",
            "customerId": "abcb298a-d535-434b-bb35-217a64bd516d",
            "customerType": "service",
            "balance": "500000000",
            "nemoBalanceActionId": "1"
        }
    }
]
```

Figure 51: Example of the transaction logs of the workload registration

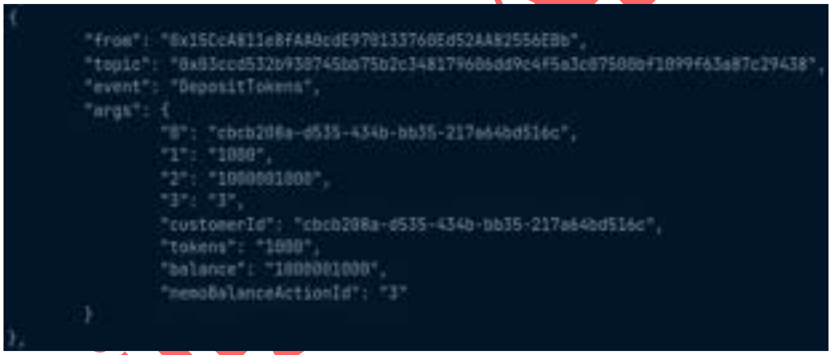
The main function of the contract is to calculate the tokens that will be charged to the workload for the usage it made in a period, for example, 5 minutes. The next figures show the logs for the usage calculation, and it will be explained how these reflect on the involved actors (clusters, running workloads). Figure 52 shows how the balance of the workload was affected. In this simple example, for its usage of the cluster resources, it was charged 0,00001 tokens.



```
{
        "from": "0x15CcA811e8fAA0cdE970133760Ed52AA82556EBb",
        "topic": "0xfbb007563a8fb568bd9e39cca9606e59ec87d68bd711255899f01eb90d140e87",
        "event": "WithdrawTokens",
        "args": {
                "0": "abcb208a-d535-434b-bb35-217a64bd516d",
                "1": "1000",
                "2": "499999000",
                "3": "2",
                "customerId": "abcb208a-d535-434b-bb35-217a64bd516d",
                "tokens": "1000",
                "balance": "499999000",
                "nemoBalanceActionId": "2"
        }
},
```

Figure 52: Example of the transaction logs of the workload usage fee

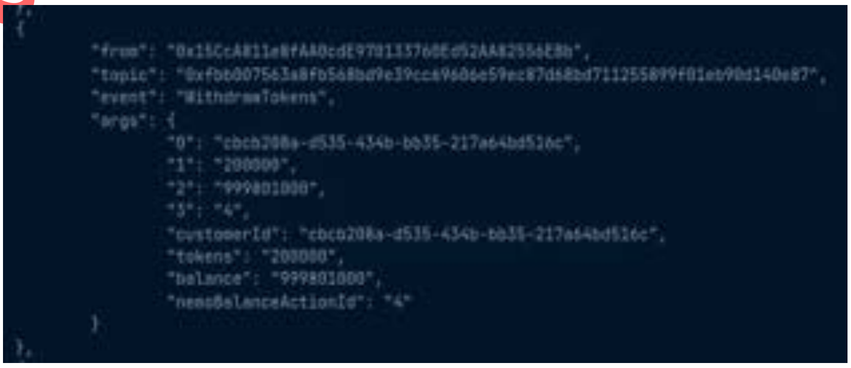
These tokens are credited to the cluster, as shown in Figure 53.



```
{
        "from": "0x15CcA811e8fAA0cdE970133760Ed52AA82556EBb",
        "topic": "0x83ccd532b938745bb75b2c348179606dd9c4f5a3c87508bf1099f63a87c29438",
        "event": "DepositTokens",
        "args": {
                "0": "cbcb208a-d535-434b-bb35-217a64bd516c",
                "1": "1000",
                "2": "1000001000",
                "3": "3",
                "customerId": "cbcb208a-d535-434b-bb35-217a64bd516c",
                "tokens": "1000",
                "balance": "1000001000",
                "nemoBalanceActionId": "3"
        }
},
```

Figure 53: Example of the transaction logs of the cluster reward

Figure 54 shows that a 0.02% rate is rewarded to the NEMO account from the cluster's balance to reward NEMO with a small payment for the services provided.



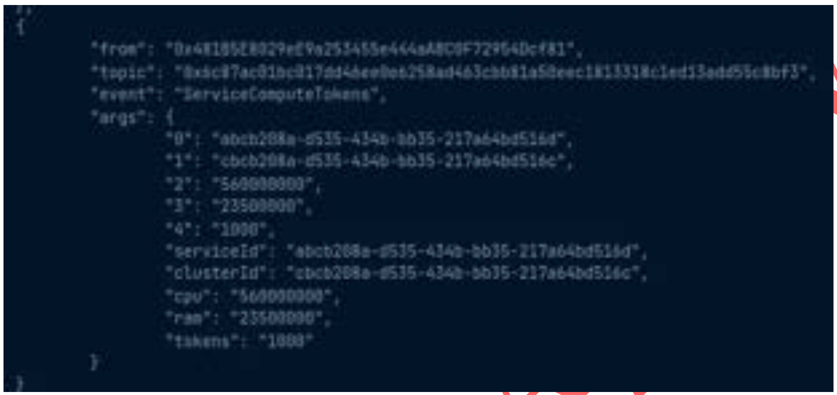
```
},
{
        "from": "0x15CcA811e8fAA0cdE970133760Ed52AA82556EBb",
        "topic": "0xfbb007563a8fb568bd9e39cca9606e59ec87d68bd711255899f01eb90d140e87",
        "event": "WithdrawTokens",
        "args": {
                "0": "cbcb208a-d535-434b-bb35-217a64bd516c",
                "1": "200000",
                "2": "999801000",
                "3": "4",
                "customerId": "cbcb208a-d535-434b-bb35-217a64bd516c",
                "tokens": "200000",
                "balance": "999801000",
                "nemoBalanceActionId": "4"
        }
},
```

Figure 54: Example of the transaction logs of the NEMO fee paid by the cluster owner

Finally, Figure 55 gives a detailed account on the computation details (how many resources were used to justify the cost). In this example, the workload was charged 0,00001 tokens for utilizing 5,6 milliecycles of CPU and 0,235 MB RAM for a 5-minute time window that the metrics were collected.

The token calculations are examining the usage of the workload in question against the total resource usage of the deployment cluster, in order to reflect the real-time pressure on the cluster. Additionally, if the workload has exceeded the base resource limitations of the region it is assigned to, then that will be added to the total cost.

A more thorough example of how the workload usage calculation is performed end-to-end through MOCA will be presented in the next section.



```
},
{
        "from": "0x4818SE8029eE9a253455e444aA8C0F72954Dcf81",
        "topic": "0xdc87ac01bc017dd4ee0eb258ad463cbb81a50eec1813318c1ed13add55c8bf3",
        "event": "ServiceComputeTokens",
        "args": {
                "0": "abcb208a-d535-434b-bb35-217a64bd516d",
                "1": "cbcb208a-d535-434b-bb35-217a64bd516c",
                "2": "560000000",
                "3": "23500000",
                "4": "1000",
                "serviceId": "abcb208a-d535-434b-bb35-217a64bd516d",
                "clusterId": "cbcb208a-d535-434b-bb35-217a64bd516c",
                "cpu": "560000000",
                "ram": "23500000",
                "tokens": "1000"
        }
}
```

Figure 55: Example of the transaction logs of the calculation of the workload usage fee

Table 8 shows the details of the contract.

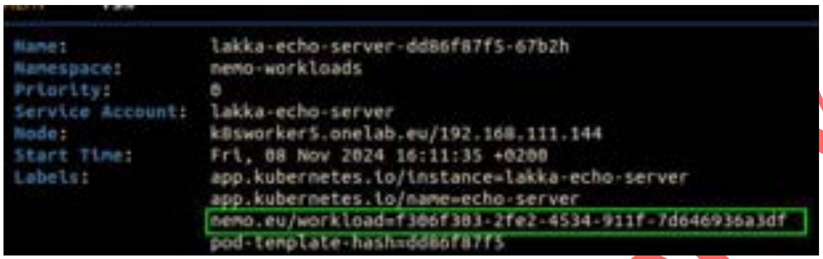
ServiceProviderModel.sol

```
pragma solidity ^0.8.0;
import "./NemoTokenEstimationSetupContract.sol";
import "./NemoFunds.sol";

contract ServiceProviderModel {
    NemoTokenEstimationSetupContract public nemoTokenEstimationSetup;
    NemoFunds public nemoFunds;
    address public owner;

    struct ServiceMetrics {
        string serviceId;
        string clusterId;
        string region;
        uint256 cpuUsage;
        uint256 memoryUsage;
        uint256 clusterCpuUsage;
        uint256 clusterMemoryUsage;
    }

    event ServiceComputeTokens(
        string serviceId,
        string clusterId,
        uint256 cpu,
        uint256 ram,
        uint256 tokens
    );

    mapping(string => string[]) public ServiceProviderWorkflows;

    constructor(
        address _nemoTokenEstimationSetupContractAddress,
        address _nemoFundsAddress
    ) {
        nemoTokenEstimationSetup = NemoTokenEstimationSetupContract(
            _nemoTokenEstimationSetupContractAddress
        );
        nemoFunds = NemoFunds(_nemoFundsAddress);
    }

    modifier checkRegistration(string memory serviceId) {
        require(
            !nemoFunds.isCustomerRegistered(serviceId),
            "The customer is already registered!"
        );
        _;
    }

    modifier checkRegionData(string memory region) {
        require(
            nemoTokenEstimationSetup.isRegionSet(region),
            "Data for region must be set before calling this function."
        );
        _;
    }

    function register(
        string memory serviceId
    ) public checkRegistration(serviceId) {
        string memory _identifier = "ServiceProvider";
        nemoFunds.registerCustomer(serviceId, _identifier);
    }

    function computeCredits(
        ServiceMetrics memory _metrics
    ) public checkRegionData(_metrics.region) {
        require(
            nemoFunds.isCustomerRegistered(_metrics.clusterId),
            "The cluster is not registered!"
        );
        require(
            nemoFunds.isCustomerRegistered(_metrics.serviceId),
            "The service is not registered!"
        );
        (
            bool _highDemand,
            uint256 _highDemandCost,
            uint256 _regionalCpuCosts,
            uint256 _regionalMemoryCosts
        ) = nemoTokenEstimationSetup.getRegionInfo(_metrics.region);

        string memory _serviceId = _metrics.serviceId;
        string memory _clusterId = _metrics.clusterId;
        uint256 _tokens = 0;

        // CPU
        uint256 _cpuTokens;
        uint256 _cpuUsage = _metrics.cpuUsage * 10 ** 3;
        uint256 _maxCpuUsage = _metrics.clusterCpuUsage;

        //RAM
        uint256 _ramTokens;
        uint256 _ramUsage = _metrics.memoryUsage * 10 ** 3;
        uint256 _maxRamUsage = _metrics.clusterMemoryUsage;

        if (_cpuUsage > _regionalCpuCosts) {
            _cpuTokens = (_cpuUsage / _maxCpuUsage) * 10 ** 8;
        } else {
            _cpuTokens = 0;
        }

        if (_ramUsage > _regionalMemoryCosts) {
            _ramTokens = (_ramUsage / _maxRamUsage) * 10 ** 8;
        } else {
            _ramTokens = 0;
        }

        _tokens = _cpuTokens + _ramTokens;

        if (_highDemand) {
            _tokens += _highDemandCost;
        }

        _tokens = _tokens / 1000;

        nemoFunds.makeTransaction(_serviceId, _clusterId, _tokens);

        emit ServiceComputeTokens(
            _serviceId,
            _clusterId,
            _cpuUsage,
            _ramUsage,
            _tokens
        );
        }
}
```

Table 8: The ServiceProviderModel smart contract

3.3.4.5 Accountability Service

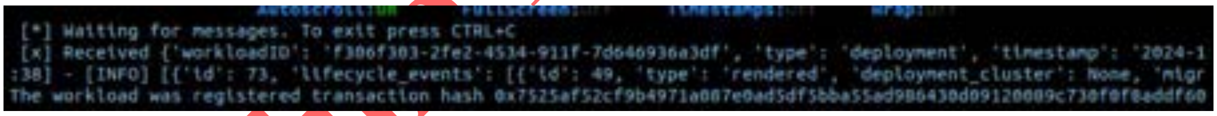
In this section we present an example of how MOCA computes the tokens that will be charged for the resource usage of the deployment cluster end-to-end. For this example, we will use a workload deployed in the NEMO OneLab cluster (Figure 56).



```
Name:               lakka-echo-server-dd86f87f5-67b2h
Namespace:          nemo-workloads
Priority:           0
Service Account:    lakka-echo-server
Node:               k8sworker5.onelab.eu/192.168.111.144
Start Time:         Fri, 08 Nov 2024 16:11:35 +0200
Labels:             app.kubernetes.io/instance=lakka-echo-server
                    app.kubernetes.io/name=echo-server
                    nemo.eu/workload=f306f303-2fe2-4534-911f-7d646936a3df
                    pod-template-hash=dd86f87f5
```

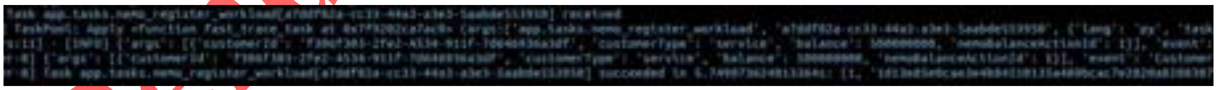
Figure 56: OneLab workload details

This workload is first registered though the Intent-Based API and is deployed to the cluster through the Meta Orchestrator. After the successful deployment an event is published in the NEMO RabbitMQ which is consumed by MOCA. (Figure 57 shows for the workload of Figure 56 that the Meta Orchestrator has sent the payload which informs of the successful deployment.) Then, MOCA registers in the appropriate smart contract for the specific workload (for more details refer to section 3.3.4.4 workload provision and usage calculation) (Figure 58) and give the owner of the workload five initialization tokens (Figure 59). The registration event can also be viewed though the */moca/api/v1/accounting_events* endpoint (Figure 60).



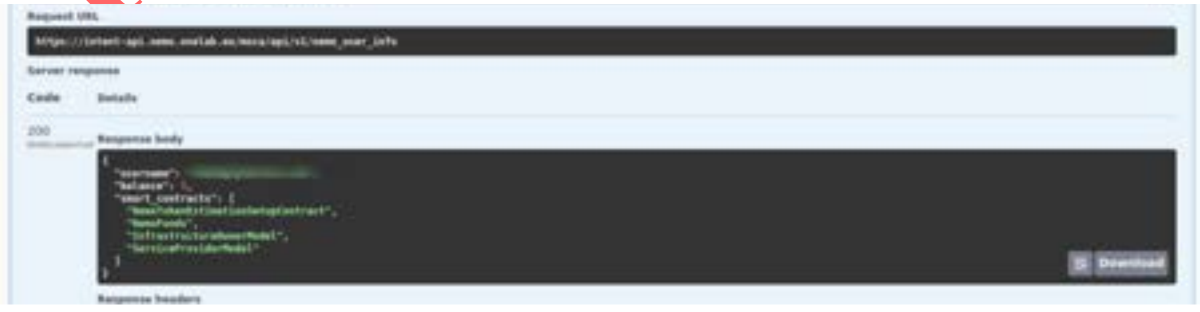
```
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received {'workloadID': 'f306f303-2fe2-4534-911f-7d646936a3df', 'type': 'deployment', 'timestamp': '2024-1
:38] - [INFO] [{'id': 73, 'lifecycle_events': [{'id': 49, 'type': 'rendered', 'deployment_cluster': None, 'migr
The workload was registered transaction hash 0x7525af52cf9b4971a087e0ad5df5bba55ad986430d09120089c730f0f8addf60
```

Figure 57: RabbitMQ logs of workload deployment



```
Task app.tasks.nemo_register_workload[a70df82a-cc33-44a3-a3e5-5aabde553958] received
 TaskPool: Apply <function fast_trace_task at 0x7f9202ca7ac0> (args:('app.tasks.nemo_register_workload', 'a70df82a-cc33-44a3-a3e5-5aabde553958', {'lang': 'py', 'task
s:11] - [INFO] {'args': [{'customerId': 'f306f303-2fe2-4534-911f-7d646936a3df', 'customerType': 'service', 'balance': 5000000000, 'nemoBalanceActionId': 1}], 'event':
r:0] {'args': [{'customerId': 'f306f303-2fe2-4534-911f-7d646936a3df', 'customerType': 'service', 'balance': 5000000000, 'nemoBalanceActionId': 1}], 'event': 'Customer
r:0] Task app.tasks.nemo_register_workload[a70df82a-cc33-44a3-a3e5-5aabde553958] succeeded in 6.749973624813364s: [1, '1d13ed5e0cae3e4b84101135e4805cac7e282...
```

Figure 58: Workload registration to blockchain



Request URL

```
https://intent-api.nemo.onelab.eu/moca/api/v1/nemo_user_info
```

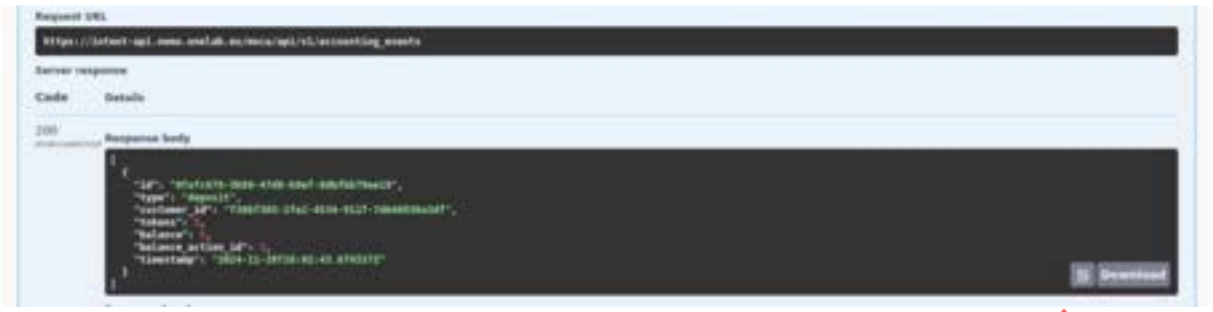
Server response

Code	Details
200	Response body

```
{
  "username": [redacted]
  "balance": 5,
  "smart_contracts": [
    "NemoTokenDistributionSetupContract",
    "NemoFunds",
    "InfrastructureOwnerModel",
    "ServiceProviderModel"
  ]
}
```

Response headers

Figure 59: User info for workload owner after registration



Request URL
```
https://intent-api.nemo.onelab.eu/moca/api/v1/accounting_events
```
Server response

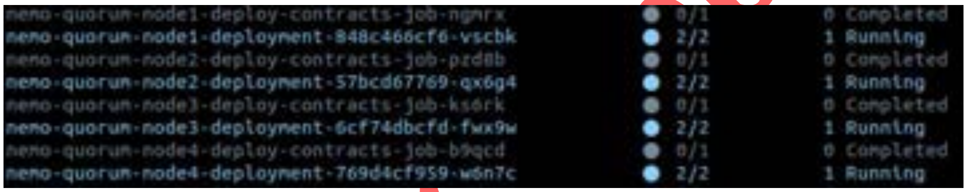
Code Details

200

Response body
```
[
  {
    "id": "[illegible]",
    "type": "deposit",
    "customer_id": "[illegible]",
    "tokens": 1,
    "balance": 1,
    "balance_action_id": 1,
    "timestamp": "2024-11-28T[illegible]"
  }
]
```
Download

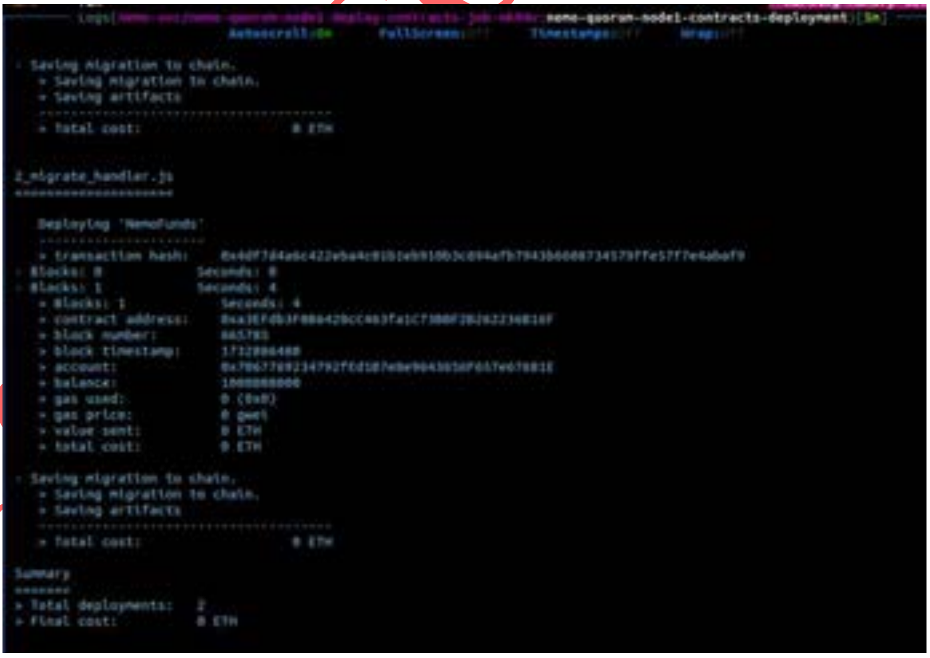
Figure 60: Accounting event for workload registration

At this point, it should be noted that the smart contracts are already deployed in the blockchain and ready to be executed when the right conditions are triggered (for example the registration of a workload that was examined before). The deployment of the contracts, at this point of the development, is performed with the help of the MOCA Helm chart available here[19]. Figure 61 shows the Kubernetes jobs that are created to perform the deployment of the contracts and Figure 62 gives an example of the logs for the successful deployment of one of the contracts (in this case *NemoFunds*).



```
nemo-quorum-node1-deploy-contracts-job-ngnrx        ● 0/1    0 Completed
nemo-quorum-node1-deployment-848c466cf6-vscbk       ● 2/2    1 Running
nemo-quorum-node2-deploy-contracts-job-pzd8b        ● 0/1    0 Completed
nemo-quorum-node2-deployment-57bcd67769-qx6g4       ● 2/2    1 Running
nemo-quorum-node3-deploy-contracts-job-ks6rk        ● 0/1    0 Completed
nemo-quorum-node3-deployment-6cf74dbcfd-fwx9w       ● 2/2    1 Running
nemo-quorum-node4-deploy-contracts-job-b9qcd        ● 0/1    0 Completed
nemo-quorum-node4-deployment-769d4cf959-w6n7c       ● 2/2    1 Running
```

Figure 61: Smart Contracts deployment though Helm chart

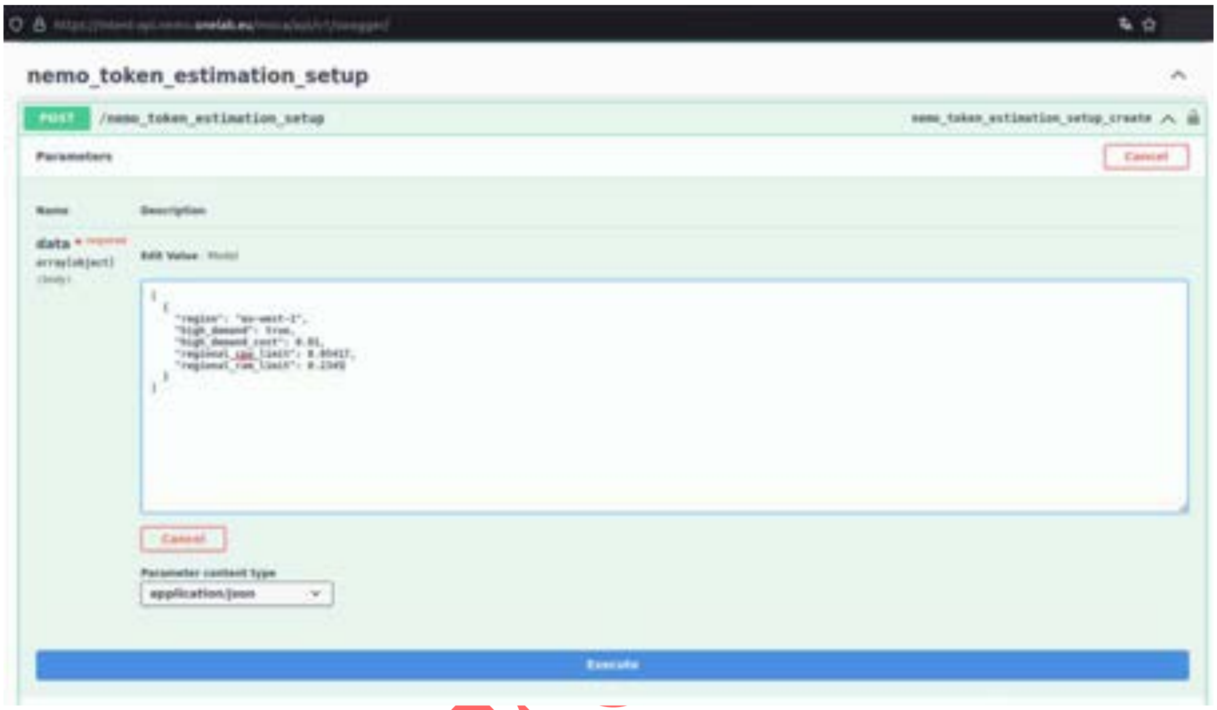


```
 Logs[[illegible]] nemo-quorum-node1-contracts-deployment [1m]
          Autoscroll:On     FullScreen:Off     Timestamps:Off     Wrap:Off

 - Saving migration to chain.
   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:                   0 ETH


2_migrate_handler.js
====================

   Deploying 'NemoFunds'
   ---------------------
   > transaction hash:    [illegible]
 - Blocks: 0            Seconds: 0
 - Blocks: 1            Seconds: 4
   > Blocks: 1            Seconds: 4
   > contract address:    [illegible]
   > block number:        [illegible]
   > block timestamp:     [illegible]
   > account:             [illegible]
   > balance:             1000000000
   > gas used:            0 (0x0)
   > gas price:           0 gwei
   > value sent:          0 ETH
   > total cost:          0 ETH

 - Saving migration to chain.
   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:                   0 ETH

Summary
=======
> Total deployments:   2
> Final cost:          0 ETH
```

Figure 62: Logs of deployment of NemoFunds contracts

[19] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-service-management/monetization-and-consensus-based-accountability/moca/-/tree/main/bc-network?ref_type=heads

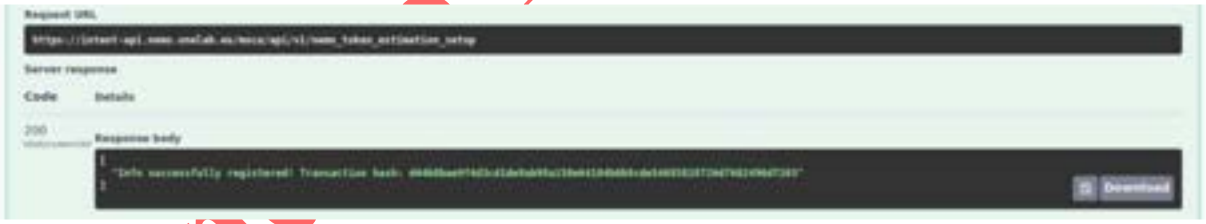
In this example we are using the region *"eu-west-1"* of the Onelab cluster, in which we are going to register the region as high demand and charge a fee of 0.01 tokens and set the base utilization limit as 0.05417 milliseconds for the CPU and 0.2345 MBs for the memory. If these limits are exceeded, the workload will be charged accordingly (more details on the costing mechanism can be found in section 3.3.4.4 workload provision and usage calculation). Figure 63 and Figure 64 show the successful registration of the region to the blockchain though MOCA.


nemo_token_estimation_setup

POST /nemo_token_estimation_setup

```
[
  {
    "region": "eu-west-1",
    "high_demand": true,
    "high_demand_cost": 0.01,
    "regional_cpu_limit": 0.05417,
    "regional_ram_limit": 0.2345
  }
]
```

Figure 63: Registering NEMO OneLab Cluster regional info


Request URL

```
https://intent-api.nemo.onelab.eu/moca/api/v1/nemo_token_estimation_setup
```

Server response

Code Details

200

Response body

```
[
  "Info successfully registered! Transaction hash: [illegible]"
]
```

Figure 64: Response for successful registration

MOCA has in place automatic mechanisms that communicate with the PPEF component to acquire the resource usage of all the deployed workloads in periodic intervals (e.g. 5 minutes). Figure 65 shows the logs of MOCA, which has received from the appropriate smart contract the event with the calculation details.

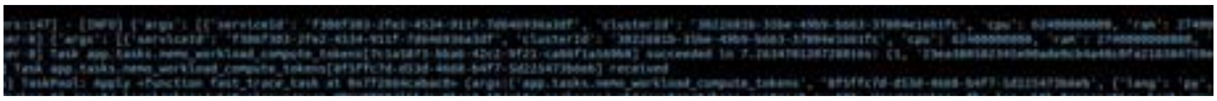
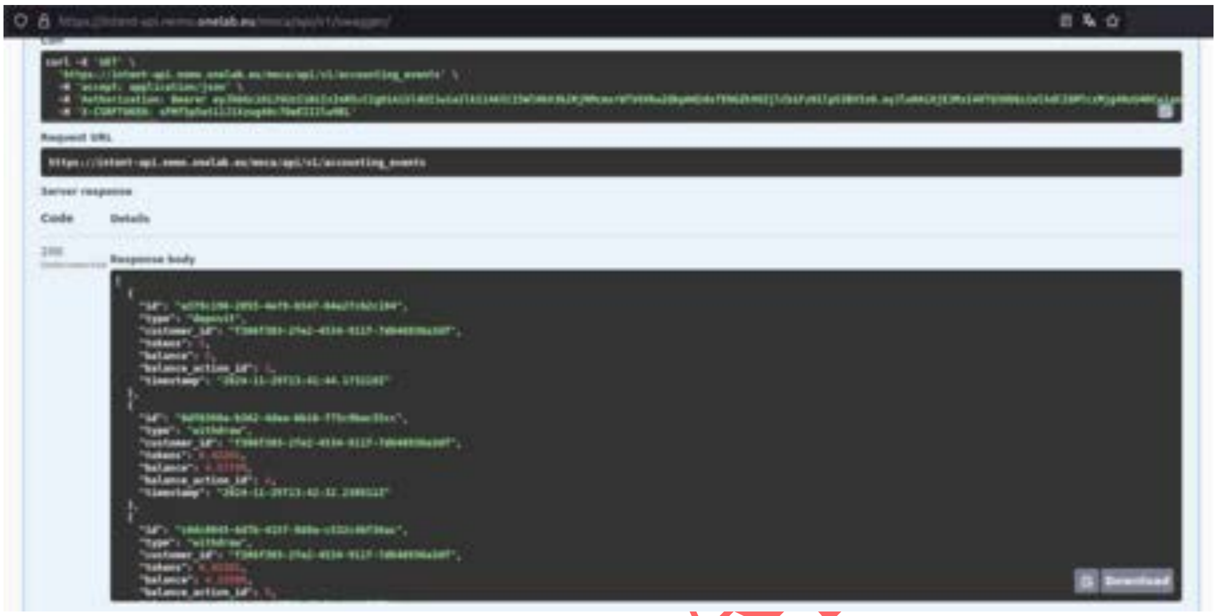

```
rs:147] - [INFO] {'args': [{'serviceId': 'f306f303-2fe2-4534-911f-7d0409 [illegible]', 'clusterId': '38220b1b-33be-4969-b663-3f [illegible]', 'cpu': 6240000000, 'ram': 274[illegible]
er-8] {'args': [{'serviceId': 'f306f303-2fe2-4534-911f-7d0409 [illegible]', 'clusterId': '38220b1b-33be-4969-b663-3f [illegible]', 'cpu': 6240000000, 'ram': 2740000000000,
er-8] Task app.tasks.nemo_workload_compute_tokens[7c1a5873-bba6-42c2-9f23-ca08f1a5606b] succeeded in 7.263470128720818s: (1, '[illegible]
] Task app.tasks.nemo_workload_compute_tokens[0f5ffc7d-d53d-4668-b4f7-5d2254730eb6] received
] TaskPool: Apply <function fast_trace_task at 0x7f2884cabac0> (args:('app.tasks.nemo_workload_compute_tokens', '0f5ffc7d-d53d-4668-b4f7-5d2254730eb6', {'lang': 'py'
```

Figure 65: MOCA logs of the DApps component calculating the resource usage of a NEMO workload

Querying the endpoint for the accounting events activity (Figure 66), as the workload owner, we notice that the events hold information like the type of the transaction (deposit or withdraw), the workload ID (*customer_id*), the tokens the workload was charged depending on its usage (tokens), the state of the balance for the workload (balance), the ID of the accounting event as it is registered in the smart contract (*balance_action_id*) and the timestamp of the registration of the accounting event.



```
curl -X 'GET' \
  'https://intent-api.nemo.onelab.eu/moca/api/v1/accounting_events' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer [illegible]' \
  -H 'X-CSRFTOKEN: [illegible]'
```

Request URL

```
https://intent-api.nemo.onelab.eu/moca/api/v1/accounting_events
```

Server response

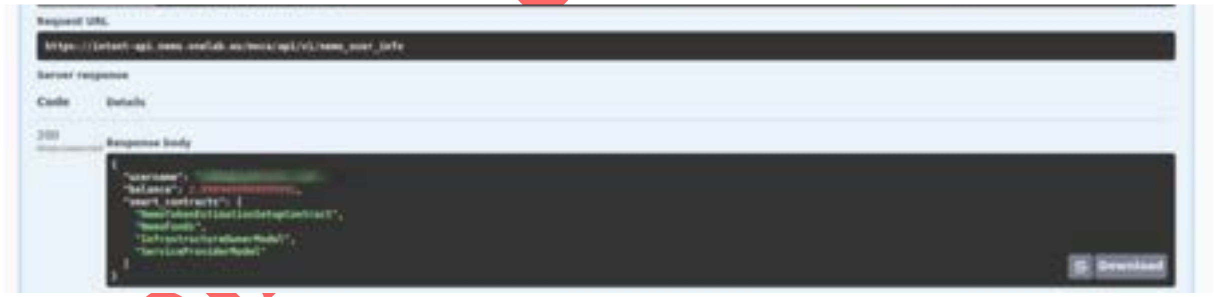
Code Details

200 Response body

```
[
  {
    "id": "[illegible]",
    "type": "deposit",
    "customer_id": "[illegible]",
    "tokens": [illegible],
    "balance": [illegible],
    "balance_action_id": [illegible],
    "timestamp": "2024-11-29T13:41:44.[illegible]"
  },
  {
    "id": "[illegible]",
    "type": "withdraw",
    "customer_id": "[illegible]",
    "tokens": [illegible],
    "balance": [illegible],
    "balance_action_id": [illegible],
    "timestamp": "2024-11-29T13:42:32.[illegible]"
  },
  {
    "id": "[illegible]",
    "type": "withdraw",
    "customer_id": "[illegible]",
    "tokens": [illegible],
    "balance": [illegible],
    "balance_action_id": [illegible],
```

Figure 66: The accounting events of the workload user

Querying the endpoint available for the user's information, we can see that the balance has been updated accordingly for the previous transactions Figure 67.



Request URL

```
https://intent-api.nemo.onelab.eu/moca/api/v1/nemo_user_info
```

Server response

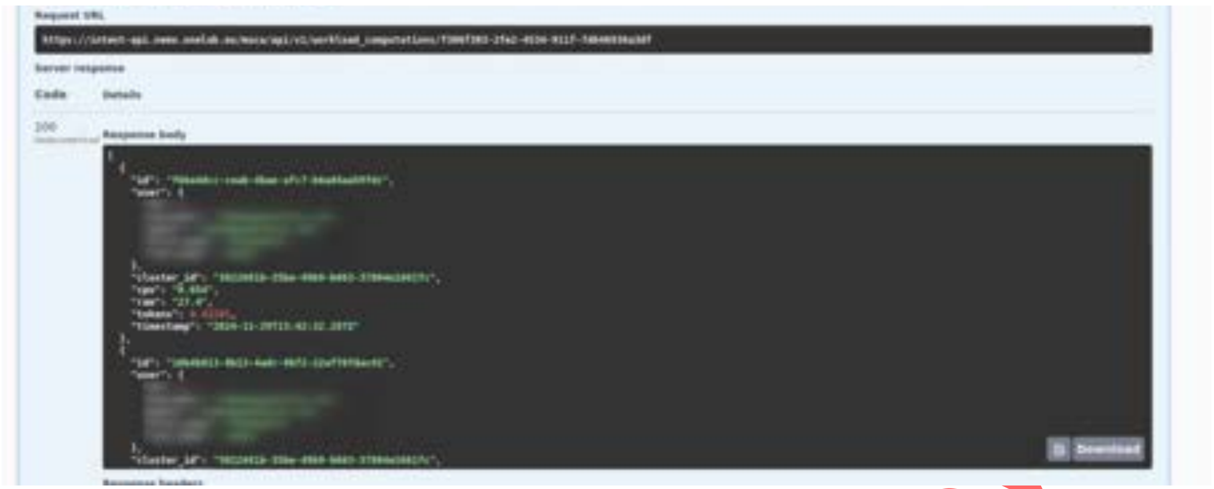
Code Details

200 Response body

```
{
  "username": [redacted],
  "balance": [illegible],
  "smart_contracts": [
    "[illegible]",
    "NemoFunds",
    "InfrastructureOwnerModel",
    "ServiceProviderModel"
  ]
}
```

Figure 67: Workload user information

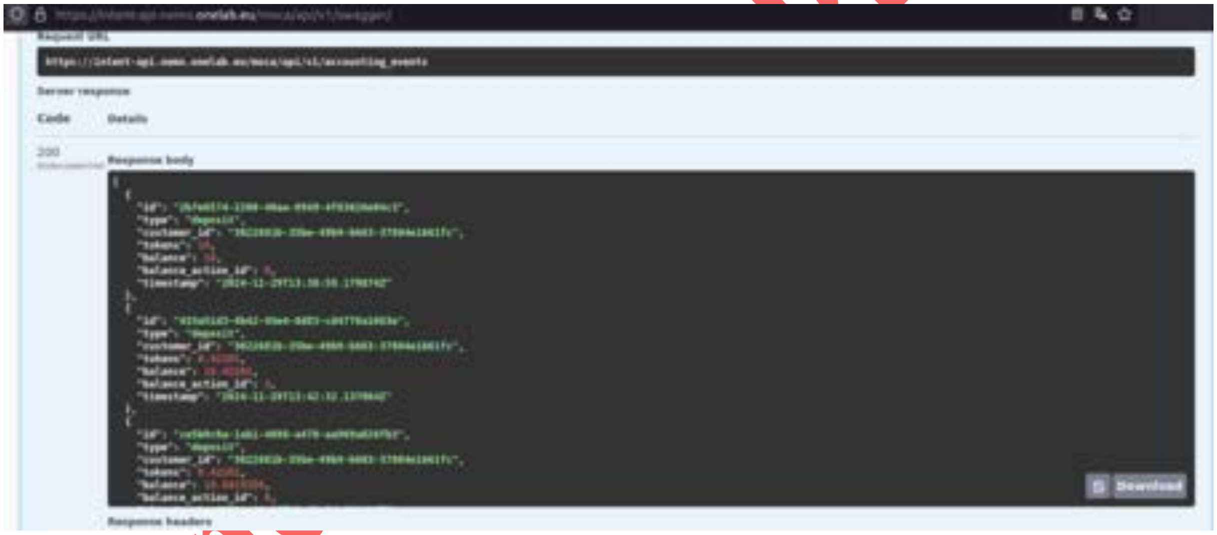
The details of the amount of resources used and the tokens charged, are also available through the */moca/api/v1/workload_computations* endpoint (Figure 68).



```
Request URL
https://intent-api.nemo.onelab.eu/moca/api/v1/workload_computations/[illegible]
Server response
Code    Details
200     Response body
[
  {
    "id": "[illegible]",
    "user": {
[redacted]
    },
    "cluster_id": "[illegible]",
    "cpu": "[illegible]",
    "ram": "27.4",
    "tokens": [illegible],
    "timestamp": "2024-11-29T13:42:12.[illegible]"
  },
  {
    "id": "[illegible]",
    "user": {
[redacted]
    },
    "cluster_id": "[illegible]",
Response headers
```

Figure 68: Details of the workload computation events

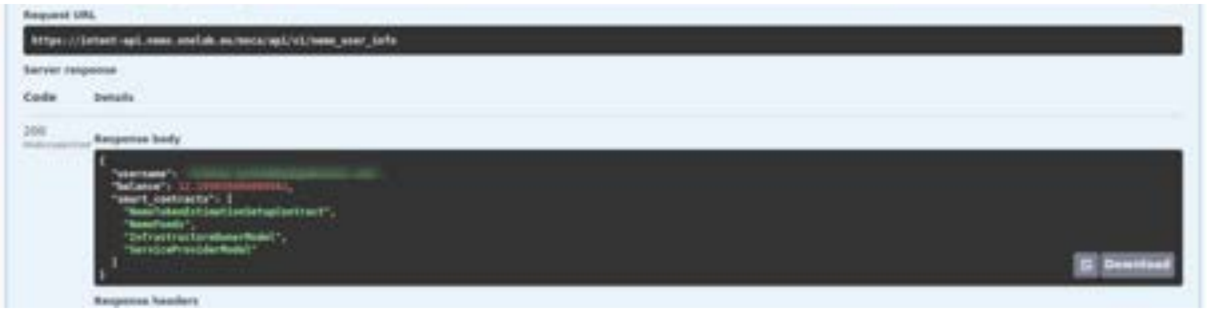
If we also query the accounting events as the cluster owner, we can see that events of depositing the payment to the owner are registered (Figure 69).



```
Request URL
https://intent-api.nemo.onelab.eu/moca/api/v1/accounting_events
Server response
Code    Details
200     Response body
[
  {
    "id": "[illegible]",
    "type": "deposit",
    "customer_id": "[illegible]",
    "tokens": [illegible],
    "balance": [illegible],
    "balance_action_id": [illegible],
    "timestamp": "2024-11-29T13:38:38.[illegible]"
  },
  {
    "id": "[illegible]",
    "type": "deposit",
    "customer_id": "[illegible]",
    "tokens": [illegible],
    "balance": [illegible],
    "balance_action_id": [illegible],
    "timestamp": "2024-11-29T13:42:12.[illegible]"
  },
  {
    "id": "[illegible]",
    "type": "deposit",
    "customer_id": "[illegible]",
    "tokens": [illegible],
    "balance": [illegible],
    "balance_action_id": [illegible],
Response headers
```

Figure 69: Cluster owner accounting events

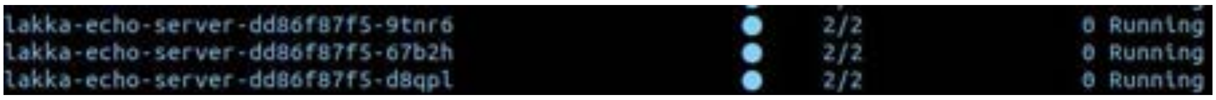
Through the endpoint for the user's information, we can also check that the balance has been updated appropriately (Figure 70).



```
Request URL
https://intent-api.nemo.onelab.eu/moca/api/v1/nemo_user_info
Server response
Code    Details
200     Response body
{
  "username": [redacted]
  "balance": [illegible],
  "smart_contracts": [
    "NemoTokenEstimationSetupContract",
    "NemoFunds",
    "InfrastructureOwnerModel",
    "ServiceProviderModel"
  ]
}
Response headers
```

Figure 70: Cluster owner user information

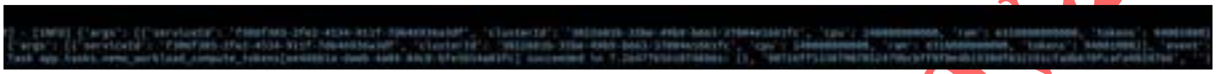
Now, we will examine the case of scaling up an already deployed workload from one to three pods (Figure 71).



```
lakka-echo-server-dd86f87f5-9tnr6     ●   2/2     0 Running
lakka-echo-server-dd86f87f5-67b2h     ●   2/2     0 Running
lakka-echo-server-dd86f87f5-d8qpl     ●   2/2     0 Running
```

Figure 71: Scaled up deployment

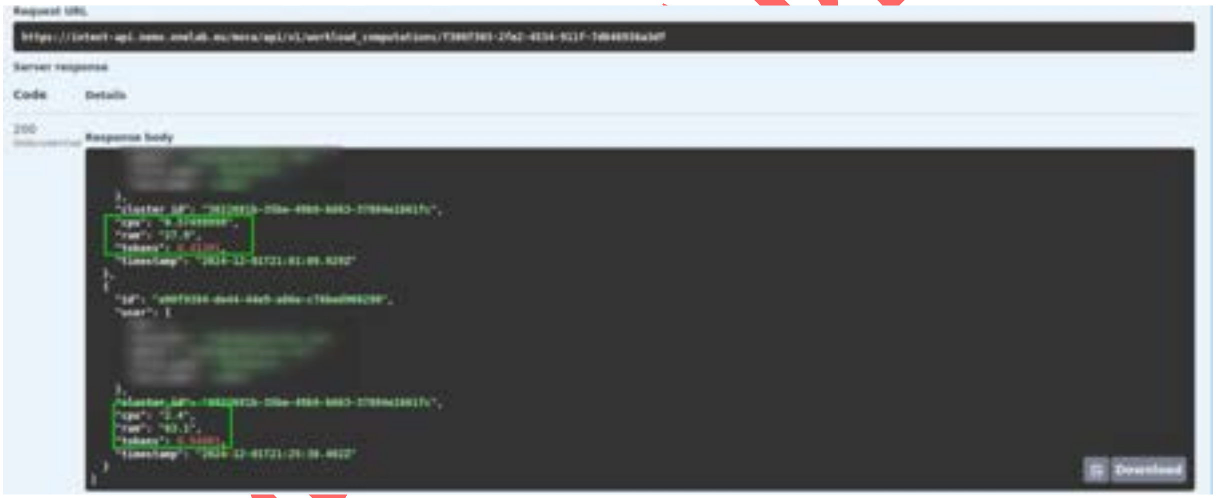
Since the deployment has been scaled up, its usage has increased, as well as the tokens it will be charged. Figure 72 shows the logs of MOCA when receiving the token computation for the workload resources.



```
[illegible]
```

Figure 72: MOCA logs for scaled workload usage

The last two entries in the */moca/api/v1/workload_computations* endpoint show the computations before and after the workload scale, respectively. Both the memory and the CPU have increased and, as a result, the usage cost has a slight increase, as well (Figure 73).



Request URL

```
https://intent-api.nemo.[illegible]/moca/api/v1/workload_computations/[illegible]
```

Server response

Code Details

200 Response body

```
[illegible]
```

Figure 73: Comparison of workload usage results

3.3.5 Conclusion and roadmap

The conducted developments that materialized the first release of MOCA as part of the 1st integrated version of the NEMO meta-OS are summarized below.

- The development of smart contracts that facilitate the accounting process supporting several business models.
- The integration with the Service Management Layer components as part of the 1st integrated version of the NEMO meta-OS, namely the Intent-Based API and PPEF
- The development of MOCA to handle the different types of users and the calculations and updates made in a private blockchain network.

The associated results were presented in this section verified the functional competence of the component. Although majority of the required functionality and the corresponding integration with the meta-OS platform has already been achieved, the final release of the MOCA component in view of the final version of the NEMO meta-OS, concern the following activities:

- Finalization of the accounting approach used in the smart contracts, by enriching the calculations.
- Provision of a management mechanism for adding, updating and deleting smart contracts via API.
- Enhancements on the provided functionalities to further enrich the information available for the NEMO users and their resources.
- Integration with the final version of the NEMO meta-OS.

3.4 Intent-based SDK/API

3.4.1 Overview

The NEMO Intent-based Application Programming Interface (IB-API) and Software Development Kit (SDK) act as the programmatic interface of NEMO to external users and/or services. It exposes NEMO functionality, as delivered by NEMO components, through RESTful calls to the API. It also supports configuration and for clusters and workloads in a declarative manner, realized as intent-based orchestration of network and computing loads. The Intent-based API stores NEMO workloads and their instances as API objects, which can be queried and managed via HTTP API calls. The Intent-based API can be accessed programmatically by NEMO users, as well as graphically via the LCM UI.

3.4.2 Architecture

The Intent-based API follows a modular architecture for delivering its main capabilities:

- Intent-based management that is consistent for both network and computing tasks
- Workload management and discovery
- NEMO functionalities exposure

The final version of the architecture features a simplified design and is depicted in Figure 74.

The IB-API services delivering intent-based orchestration include:

- NEMO Intent Manager
- NEMO Intent Validator
- NEMO Intent Collector

The IB-API services responsible for workload management and delivery include:

- NEMO Workload Manager
- NEMO Workload Validator
- NEMO Workload Registry, including the workload documents
- NEMO Intent Validator
- NEMO Intent Collector

In addition, the NEMO functionality exposure is directly offered through the Intent-based API Server, which provides RESTful endpoints for the supported operations.

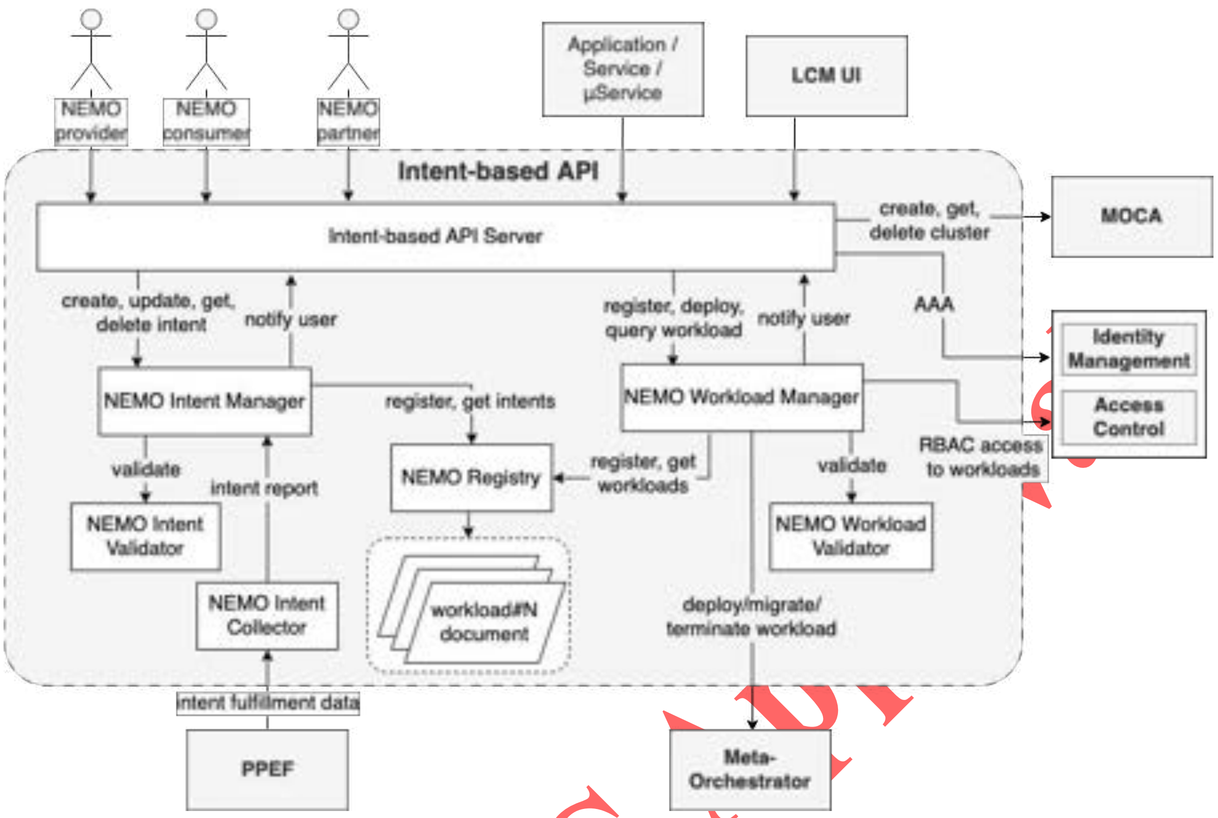


Figure 74: The final Intent-based API architecture

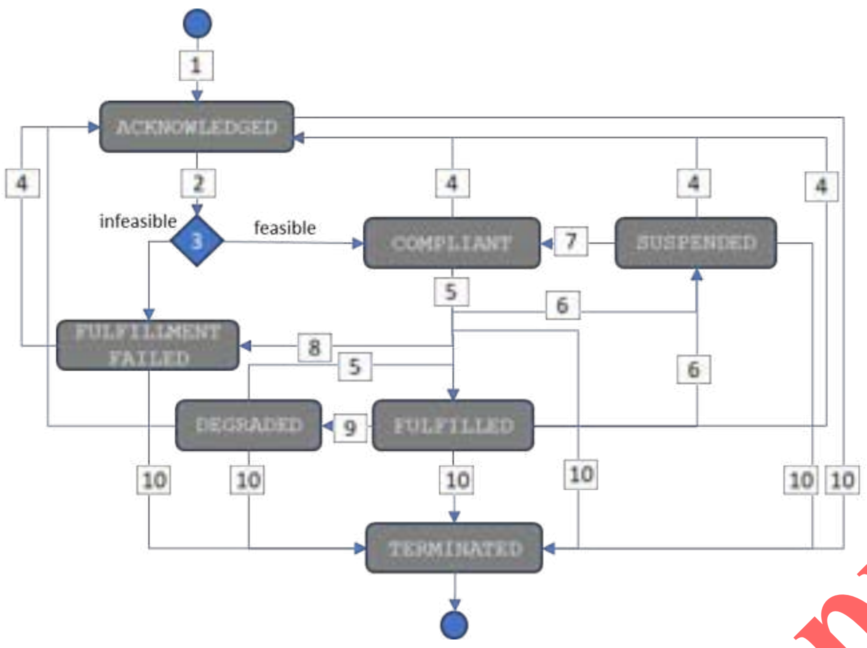
3.4.2.1 NEMO Intent Manager

Intent-driven Management has been introduced originally for automating and adding intelligence into network systems. Network management in 5G systems is becoming too complex to deal with, considering increased human intervention or policy-based network management, present in 3G and 4G architectures. Intent-Driven Management (IDM) has, then, arisen to simplify network management and interaction of involved stakeholders, with operators having been alleviated from the burden of having technical awareness of network infrastructure, their policies, etc. [7]. 3GPP [8] defines an intent as an expression of the desired state of a system used to describe an intended network or service. In other words, an intent defines operator's expectations in a declarative yet concise manner in order for it to be understandable by both humans and machines.

Adopting this approach, in NEMO we extend intent-driven management to both network and computing workloads' management. Within the Intent-based API, the *NEMO Intent Manager* subcomponent undertakes the intents' -both for network and computing- lifecycle management within the NEMO ecosystem. This subcomponent provides backend logic for the management of the NEMO intents, following 3GPP TS 28.312 V18.3.0 (2024-03) [9]. Based on this technical specification, an intent has the following properties:

- It is typically understandable by humans and also needs to be interpreted by the machine without any ambiguity.
- It expresses in a declarative manner on the desired result ("what") and not the way it will be achieved ("how"). So, the intent includes metrics and target values, allowing alternative options to achieve them.
- The expectations expressed by an intent is agnostic to the underlying system implementation, technology and infrastructure.

Following TS 28.312, the *NEMO Intent Manager* subcomponent supports state management of the intents as per their lifecycle, as defined in Figure 75, borrowed from [9].



State transitions
1. Intent creation
2. Feasibility check
3. Check feasibility outcomes
4. Intent updated
5. Fulfilment
6. Intent suspended or event occurs during fulfillment
7. Intent suspension lifted
8. Fulfillment failed
9. Event after fulfillment
10. Intent deleted

Figure 75: State transitions and reporting events for Intents delivered for fulfilment. [9], supported also in NEMO

So far, five intents have been defined and integrated into the Intent-based API, namely:

- Cloud continuum (network-oriented intent)
- Deliver computing workload (computing-oriented intent)
- Secure execution (computing-oriented intent)
- Federated learning (computing-oriented intent)
- Energy carbon efficiency (computing-oriented intent)

Indicatively, the "Deliver computing workload" intent is depicted in Figure 76.

```
- id: 1
  user_label: DeliverComputingWorkload
  intent_preemption_capability: 'FALSE'
  observation_period: 60
  intent_expectations:
  - id: 1
    expectation_id: '1'
    expectation_verb: ENSURE
    expectation_object:
      id: 1
      object_type: NEMO_WORKLOAD
      object_instance: b6a77b9a-4cb2-41e9-953b-0a0b569c8cdb
      context_selectivity: null
      object_contexts: []
    expectation_targets:
    - id: 1
      target_name: cpuUsage
      target_condition: IS_LESS_THAN
      target_value_range: '20'
      target_contexts: []
    - id: 2
      target_name: ramUsage
      target_condition: IS_LESS_THAN
      target_value_range: '200'
      target_contexts: []
    expectation_contexts: []
  intent_report_reference:
    id: 1
    intent_fulfilment_report:
      id: 1
      intent_fulfilment_info:
        fulfilment_status: FULFILLED
        not_fulfilled_state: COMPLIANT
        not_fulfilled_reasons: []
      expectation_fulfilment_results:
      - expectation_fulfilment_info:
          fulfilment_status: FULFILLED
          not_fulfilled_state: COMPLIANT
          not_fulfilled_reasons: []
        expectation_id: 1
        target_fulfilment_results:
        - target: 1
          target_achieved_value: '0.307'
          target_fulfilment_info:
            fulfilment_status: FULFILLED
            not_fulfilled_state: COMPLIANT
            not_fulfilled_reasons: []
        - target: 2
          target_achieved_value: '1.2'
          target_fulfilment_info:
            fulfilment_status: FULFILLED
            not_fulfilled_state: COMPLIANT
            not_fulfilled_reasons: []
    intent_feasibility_check_report:
      id: 1
      feasibility_check_type: FEASIBLE
      infeasibility_reason: null
    last_updated_time: '2024-11-05T15:10:07.949361Z'
  intent_contexts: []
```

Figure 76: The DeliverComputingWorkload intent definition in NEMO Intent-based API

3.4.2.2 NEMO Intent Validator

The *NEMO Intent Validator* is performing a set of validation checks on intents that are newly defined or desired to be updated. It interacts with the Intent Manager enhancing its provided functionality offering validation checks which include:

- *Schema validation:* This process ensures that data conforms to a defined structure or format, typically described in a schema. A schema acts as a blueprint, specifying the expected data types, required fields, and constraints for a dataset. The schema used for intents follows 3GPP TS 28.312 V18.3.0 specification, so the component ensures that the required fields are provided, within the acceptable value range, as well as in acceptable combinations (e.g. compatible target names, intent types and expectation verbs). Schema validation is crucial for maintaining data integrity, preventing malformed or unexpected data from causing errors or security vulnerabilities in applications. By enforcing these rules at an early stage, schema validation helps streamline data processing and reduces the likelihood of runtime issues.

- *Intent definition updates:* There are fields in the defined intents that are allowed to be updated, such as the period during which the intent will be active or the expectation target value range. However, such updates are allowed before the intent is activated, i.e. for intent states *"FULFILMENTFAILED"*, *"TERMINATED"* and *"ACKNOWLEDGED"*.
- *Intent operations:* Operations on intents can be performed, depending on both the desired intent action and the *"NotFulfilledState"* value. Intent operations include *"RESUME"*, *"SUSPEND"*, *"TERMINATE"*, etc. The *"Not_Fulfilled_State"* field represents the state of the intent, while it is not fulfilled. The attempted action must comply with the state transition schema in Figure 75.
- *Intent expectation updates:* Are allowed only for expectations defined for each intent. Depending on the selected *"userLabel"* (intent type), the corresponding allowed expectation targets are check for validity. For example, "*DeliverComputingWorkload"* support targets of type ram usage and/or cpu usage.
- *Additional Feasibility checks*: After intent validation, an asynchronous validation step is performed in order to test the feasibility of the intent in question. Reasons for failure in this step include "*serviceStartTimes"* & "*serviceEndTimes*" out of order and/or collisions with an already existing Intent for the given workload.

3.4.2.3 NEMO Intent Collector

The *NEMO Intent Collector* is a significant modality of the Intent-based API which facilitates the collection of the intent associated measurements governed by the PPEF component. More specifically, the NEMO Intent Collector receives as an input the intent related measurements that correspond to the expectation targets that have set by the NEMO user, as they are reported from the PPEF. The communication with the PPEF is achieved via a RabbitMQ listener service that is provided by the NEMO Intent Collector. Then the consumed information is processed and structured as intent fulfillment data which correspond to intents expectation targets' achieved values. Finally, the intent report is consumed by the NEMO Intent Manager which updates the corresponding information in the NEMO Registry.

3.4.2.4 NEMO Workload Manager

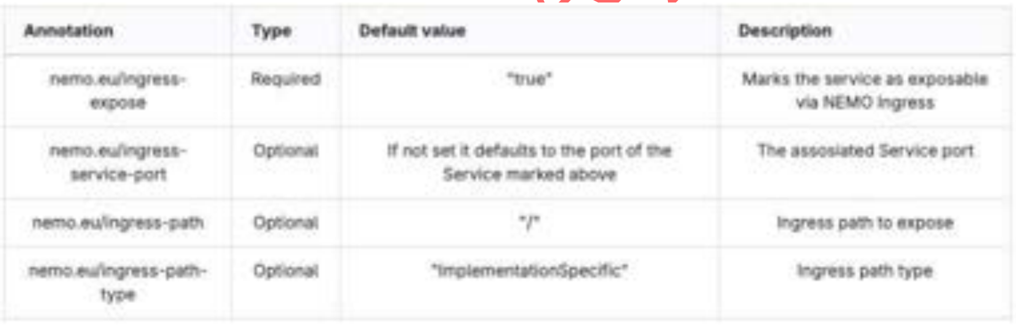
The *NEMO Workload Manager* modality is the heart of the Intent-based API component. Its functionality concerns the management and the governance of the NEMO workloads as it is dictated by the NEMO user. Specifically, the *NEMO Workload Manager* manages the processes for registration/deregistration, deployment and migration of workloads, including NEMO annotations, workflow execution, provisioning, logging and notification of external entities. As it is illustrated in Figure 74, the NEMO *Workload Manager* handles the workload requests that are dispatched by the Intent-based API. The requests can be triggered either from the LCM UI or directly from the Intent-based API Server. The workload Manager validates the workload registration and/or deployment configuration files that are issued through the *NEMO Workload Validator* (its respective activities are detailed in section 3.4.2.5). At the same time, the NEMO Workload Manager facilitates the update of the workload state in the NEMO registry and responds to workload queries. Once the workload request is pre-processed the workload's status changes to *"onboarding"* and subsequently it's communicated through the RabbitMQ message queue to the Meta-Orchestrator (MO). In case the workload validation process fails, the workload status is changing to rejected and the request is terminated. The MO executes the requested action and dispatches back the result of the requested activity (acknowledgement message). The NEMO Workload Manager updates the NEMO Registry accordingly.

Finally, the *NEMO workload Manager* supports automated provisioning, triggering authorization requests (RBAC access) for the workload to *Access Control component*.

3.4.2.5 NEMO Workload Validator

The *NEMO Workload Validator's* offered functionality, in the context of the *NEMO Workload Manager*, is described above. This section sheds light into the specific validation tests that are performed by the module. The validation tests are listed below.

1. **Helm chart structural validation.** The uploaded *.tgz helm* chart that corresponds to the *workload upload* request triggered by the NEMO user through LCM UI or directly through the Intent-based API (invokes the */workload/upload/* endpoint; described in Annex B) is validated by checking the existence of *Chart.yaml*, *values.yaml*, templates folder (with appropriate *.tpl .yaml* template files) alongside with the contents of *Chart.yaml* (matching version and naming scheme during the invocation of the */workload/* POST endpoint). In addition, all the *.yaml* files pass a syntax check.
2. **Helm chart template validation.** The uploaded *.tgz* helm chart is extracted and undergoes a rendering phase of the templates with the default provided *values.yaml*. (by using the standard helm template sub-command)
3. **Docker image access validation.** For every generated manifest (*Deployment, Statefulset, DaemonSet*) the container image of every mentioned image is tested for access. Provided *imagePullSecrets* are taken into consideration with additional secrets of type *kubernetes.io/dockerconfigjson* or against well-known docker repositories (e.g. NEMO repository)
4. **Ingress support validation.** If the uploaded NEMO workload supports ingress via the Access Control component, the validator checks for the existence of a Kubernetes Service with the annotations described in the figure below.



Annotation	Type	Default value	Description
nemo.eu/ingress-expose	Required	"true"	Marks the service as exposable via NEMO ingress
nemo.eu/ingress-service-port	Optional	If not set it defaults to the port of the Service marked above	The associated Service port
nemo.eu/ingress-path	Optional	"/"	Ingress path to expose
nemo.eu/ingress-path-type	Optional	"ImplementationSpecific"	Ingress path type

Figure 77: Kubernetes Service Annotations[20]

3.4.3 Initial results

The Intent-based API associated results stemming from the integration tests that are conducted in view of the 1st integrated version of the NEMO framework are documented in section 4.

3.4.4 Conclusion and Roadmap

The implementation of the core functionality that is offered by the Intent-based API in the context of the 1st integrated NEMO framework is considered completed. The component was integrated with the rest of the NEMO Service Management Layer components namely, the MOCA, PPEF, LCM and the Access Control. Moreover, the Intent-based API demonstrated its integration with the NEMO Kernel and the MO supporting the workload deployment and migration process.

With respect to the next steps, the Intent-based API as part of the 1st integrated version of the NEMO framework, will be deployed in NEMO pilots' infrastructures and will be further validated through the

[20] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-service-management/intent-based-sdk_api/intent-api#exposing-a-workload-document-instance-via-nemo

NEMO pilots' specific use cases and also via the integration with the NEMO OC1 offered meta-OS extensions and OC2 provided NEMO services.

In view of the final version of the NEMO meta-OS framework, the Intent-based API aims to further enhance its provided functionality where necessary (according also the feedback that will be gained through the abovementioned activities) improving the quality of its provided services.

4 NEMO scenario-driven verification & results

The purpose of this section is to provide insights on the integration tests conducted in the framework of the integration activities that materialized the 1st integrated version of the NEMO platform. NEMO integration verification approach that is defined and adopted, as described in section 2.3, establishes the foundation upon which the integration activities are conducted and documented. Emphasizing on cross-cutting functions of the NEMO meta-OS, NEMO defined four (4) system-level integration scenarios that aim to illustrate the technical readiness of the NEMO developed components. These scenarios, as detailed below, define the context of the integration activities conducted for the realization of the 1st integrated version of the NEMO meta-OS. The four (4) scenarios are the following:

- *NEMO cluster registration*
- *NEMO workload registration and provisioning*
- *NEMO workload scheduling and orchestration*
- *NEMO workload lifecycle management*

For each of the abovementioned test cases, the respective scenario that is followed is described. Based on that, the resulting process diagram highlights the steps that materialize the integration objective in each case. The scenario-driven process diagrams reflect the latest iteration/evolution of the process that was described in D1.3 [10]. Finally, the results that are collected for each of the steps are detailed and subsequently the summary checklist, presents the outcome of the conducted tests.

4.1 NEMO Cluster registration

This section describes the NEMO Cluster registration integration scenario. The Cluster registration workflow aims to provide technical details of the process that is followed allowing the NEMO partner (infrastructure owner/provider) to access the NEMO meta-OS service management layer through either via Intent-API or LCM UI and register a new resource (infrastructure) to be utilized and governed by the NEMO meta-OS. The associated sequence diagram is presented in Figure 78.

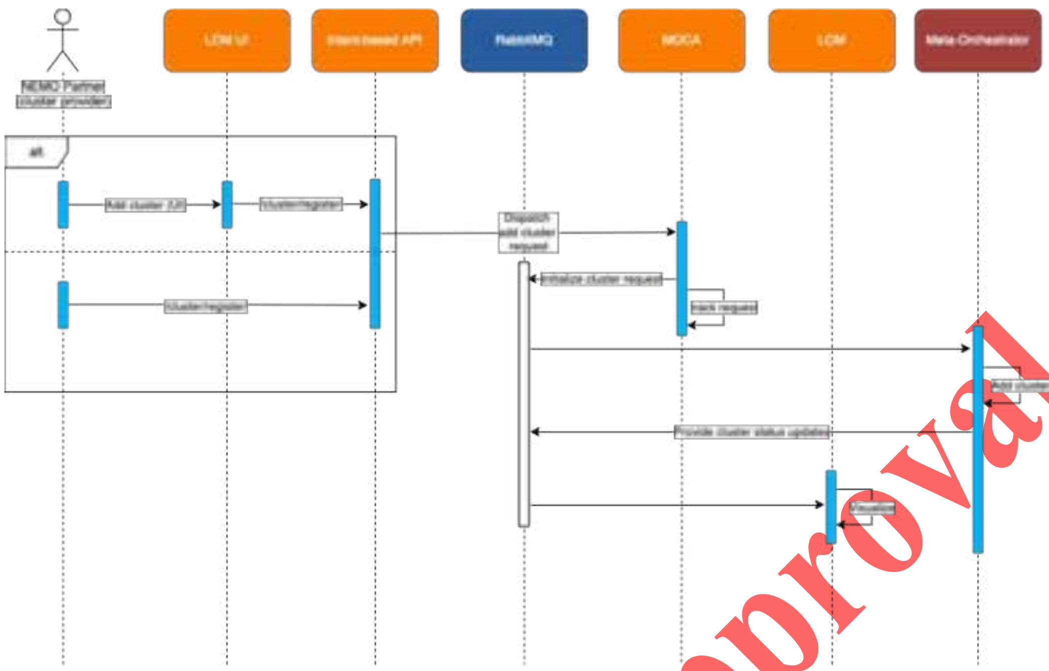


Figure 78: Process diagram for cluster registration

4.1.1 Verification scenario

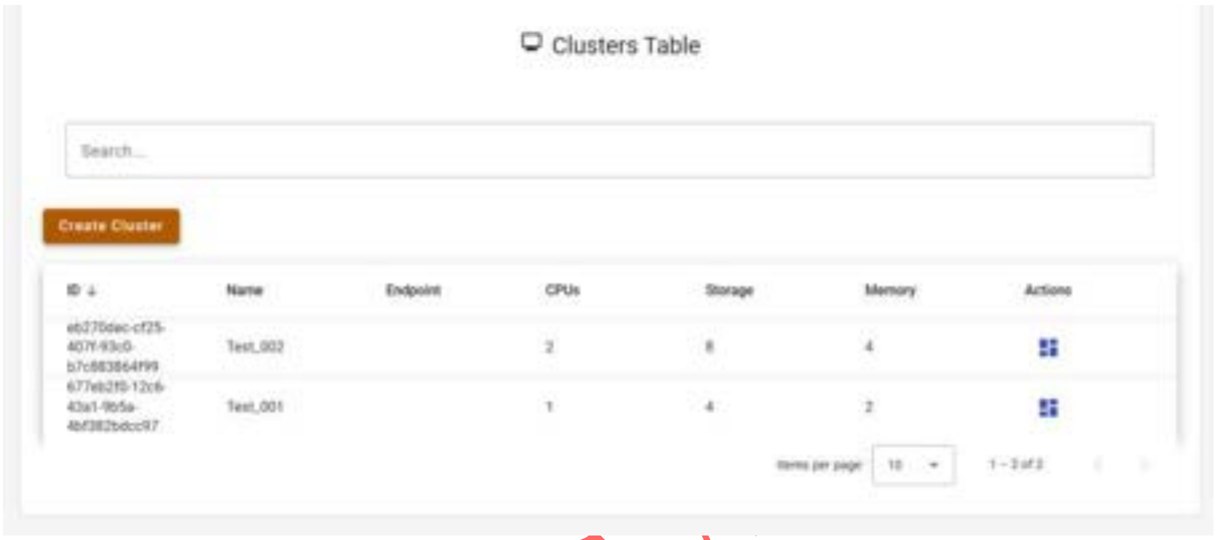
Test 1: NEMO Cluster registration	
Objective	To verify the cluster registration process in NEMO that facilitates the resource provisioning triggered by the NEMO partner (resource owner)
Components	• LCM • Intent-based API • MO • MOCA • RabbitMQ
Features to be tested	The feature that this scenario aims to test are the cluster registration process which is initiated by the NEMO partner (cluster provider) through the LCM UI & Intent-based API. Then, the newly registered cluster is added into the NEMO meta-OS ecosystem by the MO. The results (status) of this process are then visualized to the user.
Test setup	All the participating components were deployed in the NEMO meta-OS cloud/edge infrastructure at OneLab (dev cluster 1).
Steps	1. Cluster registration through the LCM UI 2. Cluster registration through the Intent-based API (realized in MOCA) 3. Cluster registration message communication to MO 4. Cluster addition process by MO 5. Cluster status provisioning to RabbitMQ 6. Cluster status update visualization in LCM UI

4.1.2 Results

This section documents the process that is described in the scenario above step by step.

4.1.2.1 Cluster registration through the LCM UI

The following figure (Figure 79) depicts the cluster table summary that the NEMO meta-OS governs. Through this interface the user is able to overview some high-level information that describes each of the provided infrastructures.

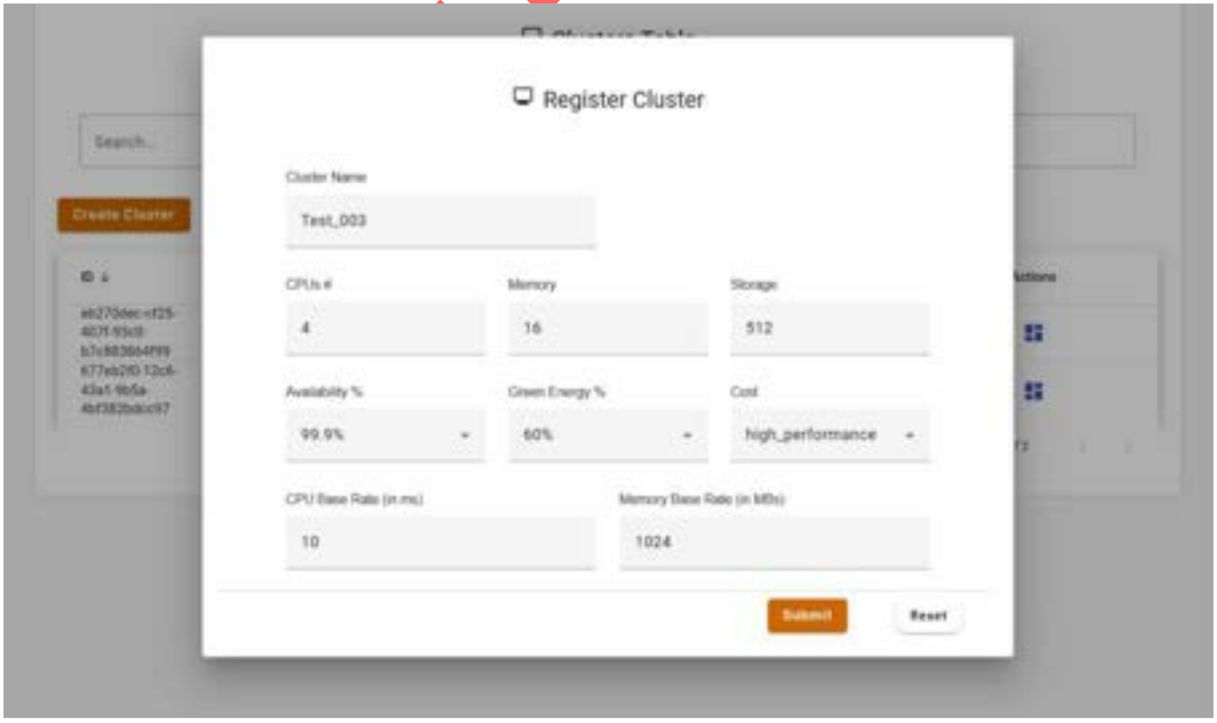


ID ↓	Name	Endpoint	CPUs	Storage	Memory	Actions
eb270dec-cf25-407f-93c0-b7c883864f99	Test_002		2	8	4	▦
677eb2f0-12c6-43a1-9b5a-4bf382bdcc97	Test_001		1	4	2	▦

Items per page: 10 1 – 2 of 2

Figure 79: Cluster summary view on LCM GUI

Figure 80, illustrates the form that corresponds to the "*create cluster*" button of the dashboard. Through this form the NEMO user is able to add the cluster description and subsequently initiate the cluster registration process. This form, once filled in by the user, triggers the corresponding endpoint that is provided through the MOCA API (section 3.3.3.1).



Cluster Name: Test_003

CPUs #: 4
Memory: 16
Storage: 512

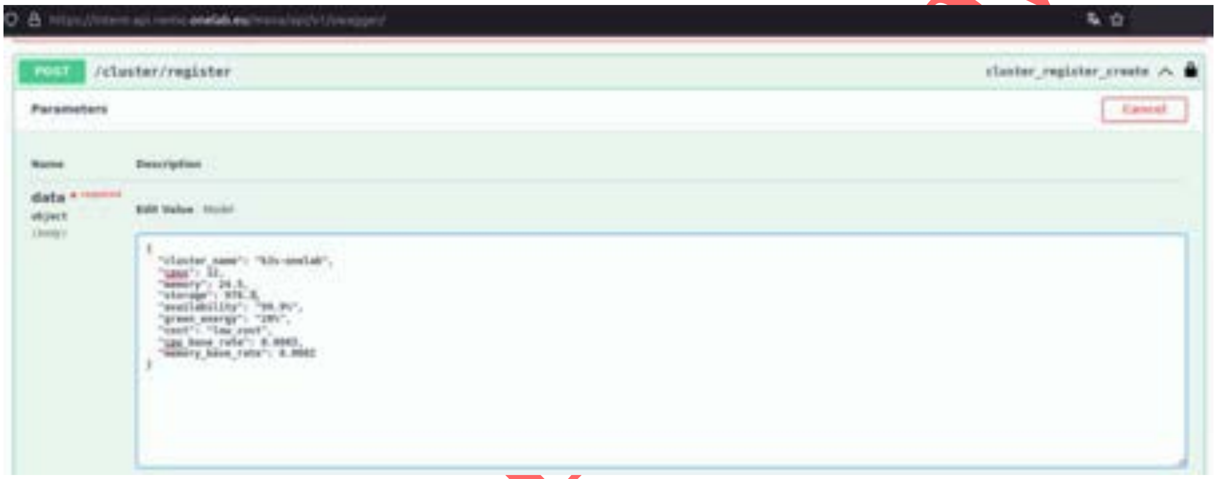
Availability %: 99.9%
Green Energy %: 60%
Cost: high_performance

CPU Base Rate (in ms): 10
Memory Base Rate (in MBs): 1024

Figure 80: Cluster registration page on LCM GUI

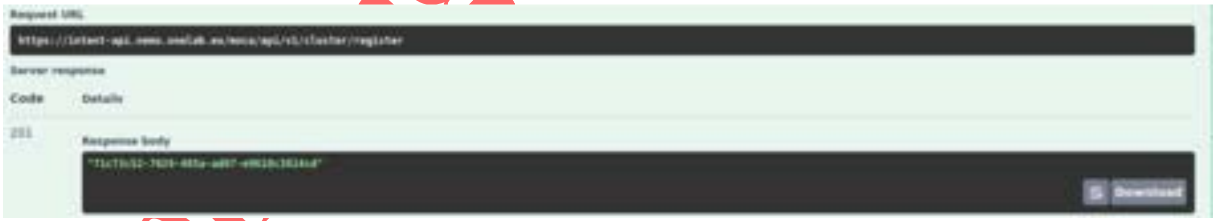
4.1.2.2 MOCA operations for cluster registration

When a new cluster comes for registration, the */moca/api/v1/cluster/register* endpoint can be used through the LCM UI. Alternatively, as indicated in the process diagram above, the user can trigger the associated endpoint directly from the Intent-based API (Figure 81). For both of these cases, the cluster registration request is executed via the MOCA API provided endpoint that is mentioned above. The cluster provider needs to provide the specifications of the cluster that is to be added, like its name, the resources it provides (CPUs, memory, disk), the availability percentage, the green energy percentage that reflects the amount of energy that used to power the cluster and comes from renewable energy sources, its cost category and the associated costs for its available resources. Figure 81 shows the registration payload of a cluster named "*k3s-onelab*". Figure 82 shows the response of the successful registration to MOCA. The response is the cluster's id.



```
{
  "cluster_name": "k3s-onelab",
  "cpus": 12,
  "memory": 24.3,
  "storage": 976.8,
  "availability": "99.9%",
  "green_energy": "10%",
  "cost": "low_cost",
  "cpu_base_rate": 0.0083,
  "memory_base_rate": 0.0002
}
```

Figure 81: MOCA Cluster registration demonstration



```
https://intent-api.nemo.onelab.eu/moca/api/v1/cluster/register
```

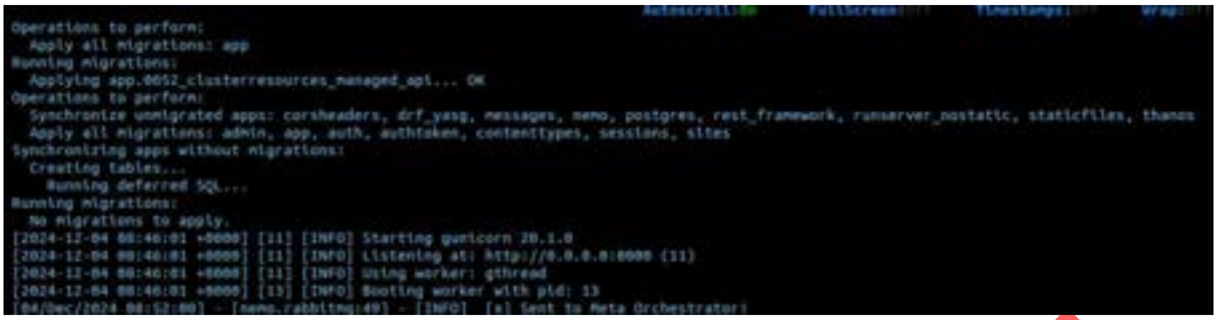
201

```
"f1c73c52-7020-485a-a897-e0618c3834cd"
```

Figure 82: MOCA Cluster registration response

4.1.2.3 MO cluster registration operation

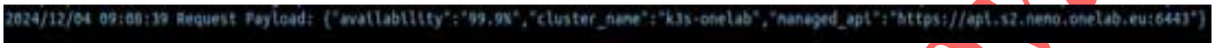
When MOCA receives a new request performs the necessary validation checks and sends the request to the NEMO Meta Orchestrator, through the NEMO RabbitMQ, in order to join the cluster with the NEMO platform. Figure 83 shows this step of the cluster registration workflow.



```
                                                    Autoscroll:On     FullScreen:Off     Timestamps:Off     Wrap:Off
Operations to perform:
  Apply all migrations: app
Running migrations:
  Applying app.0052_clusterresources_managed_api... OK
Operations to perform:
  Synchronize unmigrated apps: corsheaders, drf_yasg, messages, nemo, postgres, rest_framework, runserver_nostatic, staticfiles, thanos
  Apply all migrations: admin, app, auth, authtoken, contenttypes, sessions, sites
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
Running migrations:
  No migrations to apply.
[2024-12-04 08:46:01 +0000] [11] [INFO] Starting gunicorn 20.1.0
[2024-12-04 08:46:01 +0000] [11] [INFO] Listening at: http://0.0.0.0:8000 (11)
[2024-12-04 08:46:01 +0000] [11] [INFO] Using worker: gthread
[2024-12-04 08:46:01 +0000] [13] [INFO] Booting worker with pid: 13
[04/Dec/2024 08:52:00] - [nemo.rabbitmq:49] - [INFO]  [x] Sent to Meta Orchestrator!
```

Figure 83: MOCA sends cluster details to Meta Orchestrator

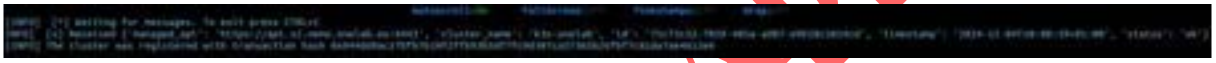
Figure 84 shows that the Meta Orchestrator has successfully received the cluster's details.



```
2024/12/04 09:08:19 Request Payload: {"availability":"99.9%","cluster_name":"k3s-onelab","managed_api":"https://api.s2.nemo.onelab.eu:6443"}
```

Figure 84: Meta Orchestrator receives the cluster registration request

After the request is processed successfully by the Meta Orchestrator, it informs MOCA again though the RabbitMQ. Figure 85 shows that MOCA received the Meta Orchestrator validation message for the successful registration. We can, additionally, see that the cluster was also registered in the blockchain.

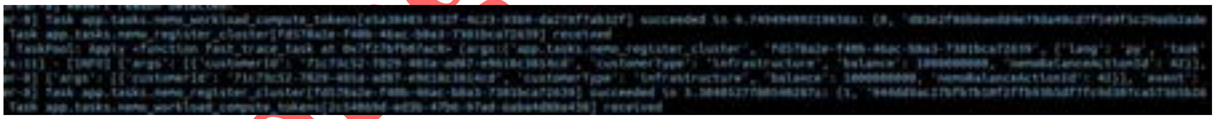


```
[INFO]  [*] Waiting for messages. To exit press CTRL+C
[INFO]  [x] Received {'managed_api': 'https://api.s2.nemo.onelab.eu:6443', 'cluster_name': 'k3s-onelab', 'id': [illegible], 'timestamp': [illegible], 'status': 'ok'}
[INFO]  The cluster was registered with transaction hash [illegible]
```

Figure 85: MOCA receives the Meta Orchestrator response

4.1.2.4 MOCA cluster registration to the blockchain

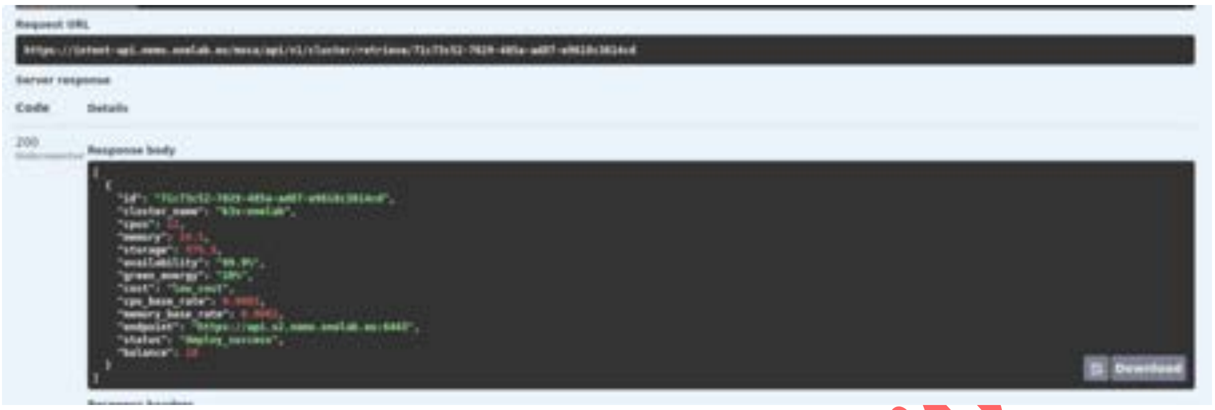
To register the cluster in the blockchain, MOCA communicates with the DApps deployed, specifically the one responsible for handling the cluster registration (see section 3.3.4.3). Figure 86 shows MOCA calling the contract and successfully registering the cluster, receiving back the appropriate response (the cluster registration initial tokens).



```
er-8] Task app.tasks.nemo_workload_compute_tokens[[illegible]] succeeded in [illegible]s: (0, [illegible]
 Task app.tasks.nemo_register_cluster[fd578a1e-f406-46ac-b8a3-7381bca72639] received
] TaskPool: Apply <function fast_trace_task at 0x7f27bfb0fac0> (args:('app.tasks.nemo_register_cluster', 'fd578a1e-f406-46ac-b8a3-7381bca72639', {'lang': 'py', 'task'
r.11] - [INFO] {'args': [{'customerId': '71c73c52-7829-481a-ad87-e9618c3814cd', 'customerType': 'infrastructure', 'balance': 1000000000, 'nemoBalanceActionId': 42}],
er-8] {'args': [{'customerId': '71c73c52-7829-481a-ad87-e9618c3814cd', 'customerType': 'infrastructure', 'balance': 1000000000, 'nemoBalanceActionId': 42}], 'event'
er-8] Task app.tasks.nemo_register_cluster[fd57e8a1e-f406-46ac-b8a3-7381bca72639] succeeded in 3.[illegible]s: (1, [illegible]
 Task app.tasks.nemo_workload_compute_tokens[[illegible]] received
```

Figure 86: Register cluster to blockchain

MOCA, then, appropriately, updates the details of the cluster and the user. Figure 87 shows that the status of the cluster has been updated, as well as the tokens provided to the cluster.



Request URL

```
https://intent-api.nemo.onelab.eu/moca/api/v1/cluster/retrieve/71c73c52-7829-485a-ad87-e9618c3814cd
```

Server response

Code Details

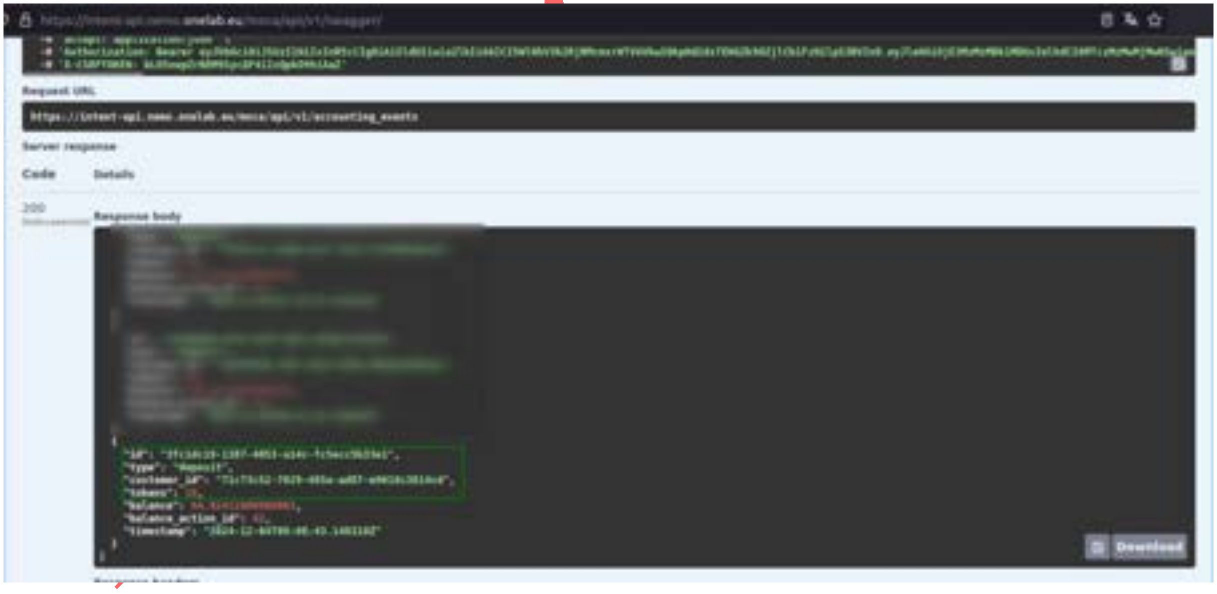
200

Response body

```
[
  {
    "id": "71c73c52-7829-485a-ad87-e9618c3814cd",
    "cluster_name": "k3s-onelab",
    "cpus": [illegible],
    "memory": [illegible],
    "storage": [illegible],
    "availability": "99.9%",
    "green_energy": "10%",
    "cost": "low_cost",
    "cpu_base_rate": [illegible],
    "memory_base_rate": [illegible],
    "endpoint": "https://api.s1.nemo.onelab.eu:6443",
    "status": "deploy_success",
    "balance": 10
  }
]
```

Figure 87: Updated cluster details

Finally, Figure 88 shows that the cluster provider can view through the */moca/api/v1/acounting_events* endpoint the event of the registration, and more specifically the deposit of the ten initialization tokens. It should be noted that the *balance* field is the total amount of tokens owned by a user. Here, the user owns a number of resources. Every time a user registers a new resource, the registration reward tokens will be added to his/her total balance.



https://intent-api.nemo.onelab.eu/moca/api/v1/swagger/

```
  -H 'accept: application/json' \
  -H 'Authorization: Bearer [illegible]' \
  -H 'X-CSRFTOKEN: [illegible]'
```

Request URL

```
https://intent-api.nemo.onelab.eu/moca/api/v1/accounting_events
```

Server response

Code Details

200

Response body

```
[redacted]
  {
    "id": "3fc1dc19-1387-4053-a14c-fc5ecc5b23e1",
    "type": "deposit",
    "customer_id": "71c73c52-7829-485a-ad87-e9618c3814cd",
    "tokens": 10,
    "balance": [illegible],
    "balance_action_id": [illegible],
    "timestamp": "2024-12-[illegible]"
  }
]
```

Figure 88: MOCA accounting event for cluster registration

4.1.3 Verification summary checklist

Checklist for Cluster registration scenario				
		Yes	**No**	**Comments**
1	Is the cluster registered by the user through the LCM UI	✓		Success
2	Is the cluster registered through the Intent-based API (realized in MOCA)	✓		Success
3	Is the registration process communicated to the MOCA	✓		Success
4	Is the MOCA validation check successfully executed?	✓		Success
5	Is MOCA communicating successfully the request to MO?	✓		Success
6	Is the MO provided functionality successfully executed?	✓		Success
7	Is the cluster successfully added to the NEMO meta-OS?	✓		Success
8	Is the updated status communicated successfully to MOCA through RabbitMQ?	✓		Success
9	Is the new cluster registered to the MOCA operated BC	✓		Success
10	Are the new cluster details available through the MOCA API?	✓		Success
11	Are the information visible to the LCM UI		✓	The provisioning of the accounting events to the LCM UI. Feature to be available in the final version.
12	Is the updated status visible to the NEMO user?	✓		Success

Table 9: Checklist for cluster registration scenario

4.2 NEMO workload registration, deployment & provisioning

This scenario concerns two specific operations. The first on is the NEMO workload registration process and the second one is the NEMO workload deployment process. For both of these activities, the corresponding sequence diagrams dictating the integration scenario that is followed are presented in Figure 89 and Figure 90, respectively. The former describes the steps necessary for the workload registration process while the latter the workload deployment and provisioning steps. The workload provisioning step that is facilitated by the Access Control is illustrated in Figure 91.

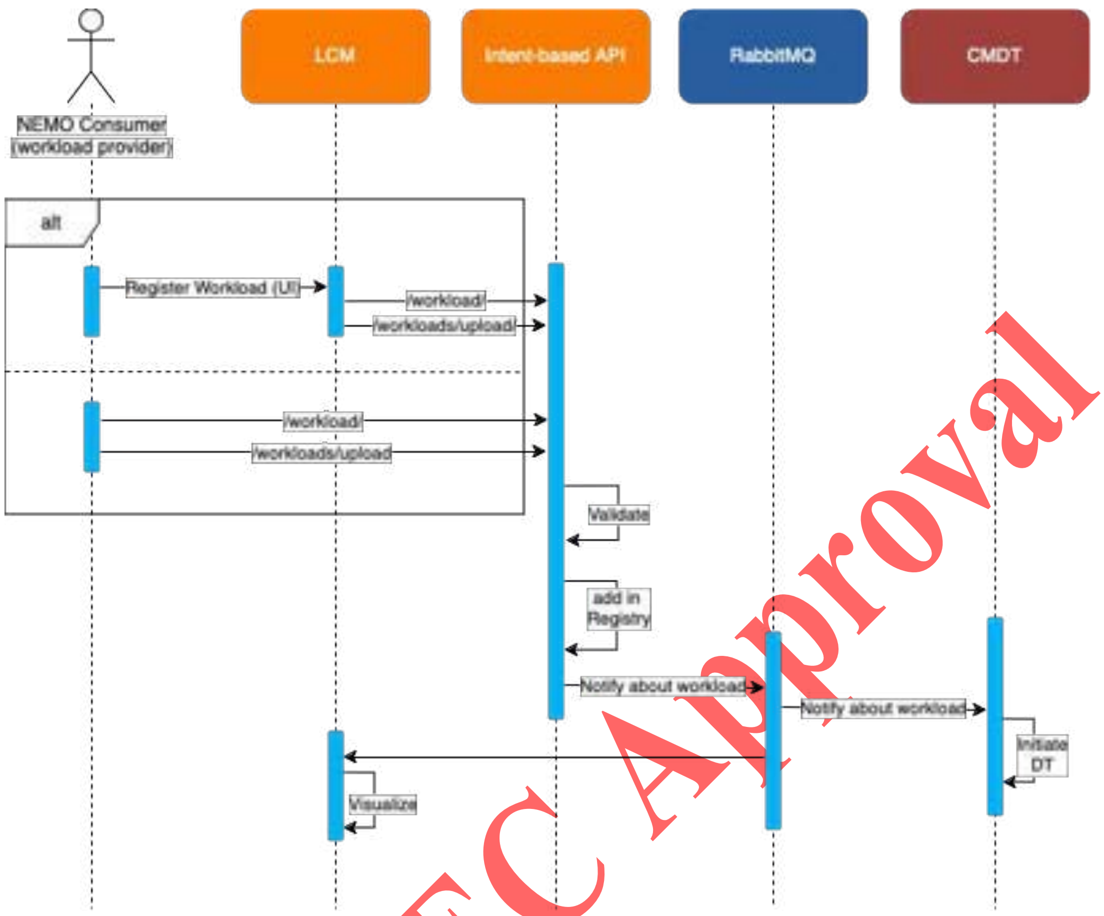


Figure 89: Process diagram for workload registration

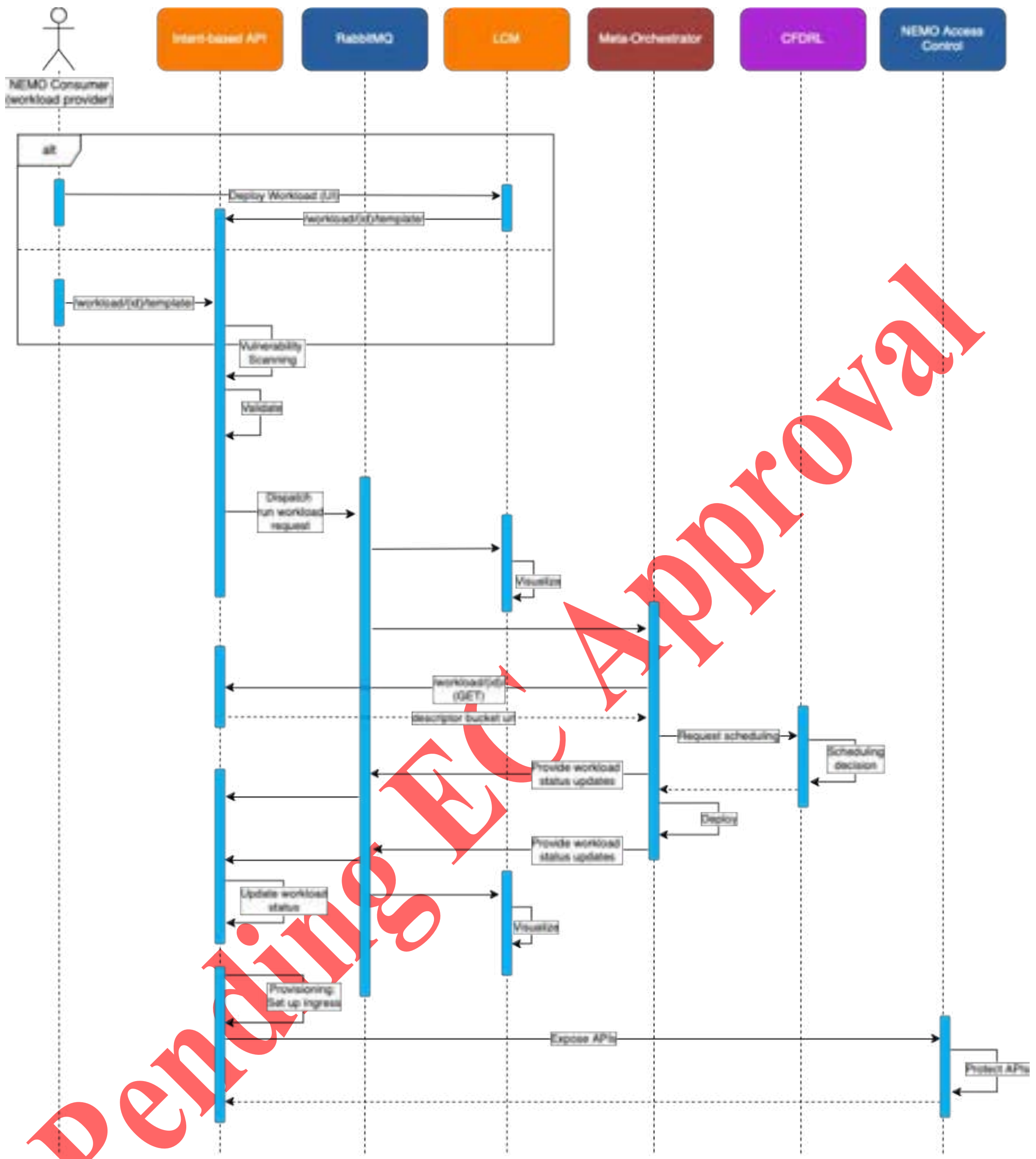


Figure 90: Process diagram for workload deployment (provisioning)

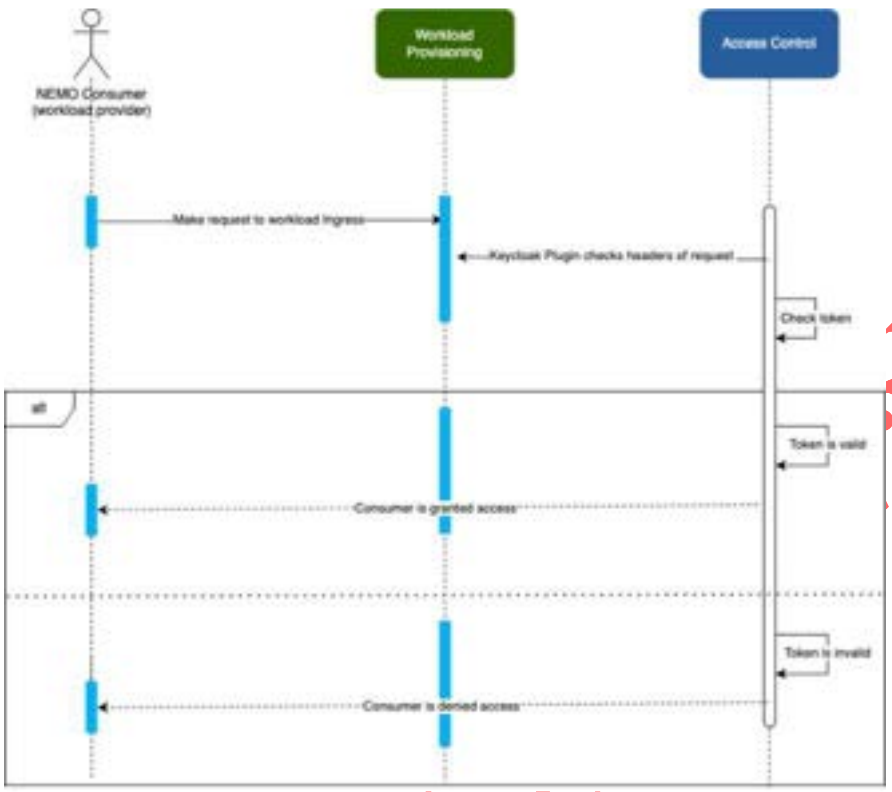

NEMO Consumer (workload provider)
Workload Provisioning
Access Control
Make request to workload ingress
Keycloak Plugin checks headers of request
Check token
alt
Token is valid
Consumer is granted access
Token is invalid
Consumer is denied access

Figure 91: Access control sequence diagram - detailed view

4.2.1 Verification scenario

Test 2: NEMO workload registration and provisioning	
Objective	Verify the NEMO workload registration, deployment and provisioning process
Components	• NEMO Workload Registration ○ LCM ○ Intent-based API ○ RabbitMQ • NEMO Workload deployment ○ LCM ○ Intent-based API ○ CMDT ○ RabbitMQ ○ Meta-Orchestrator ○ CFDRL ○ NEMO Access Control • NEMO Workload provisioning ○ Intent-based API

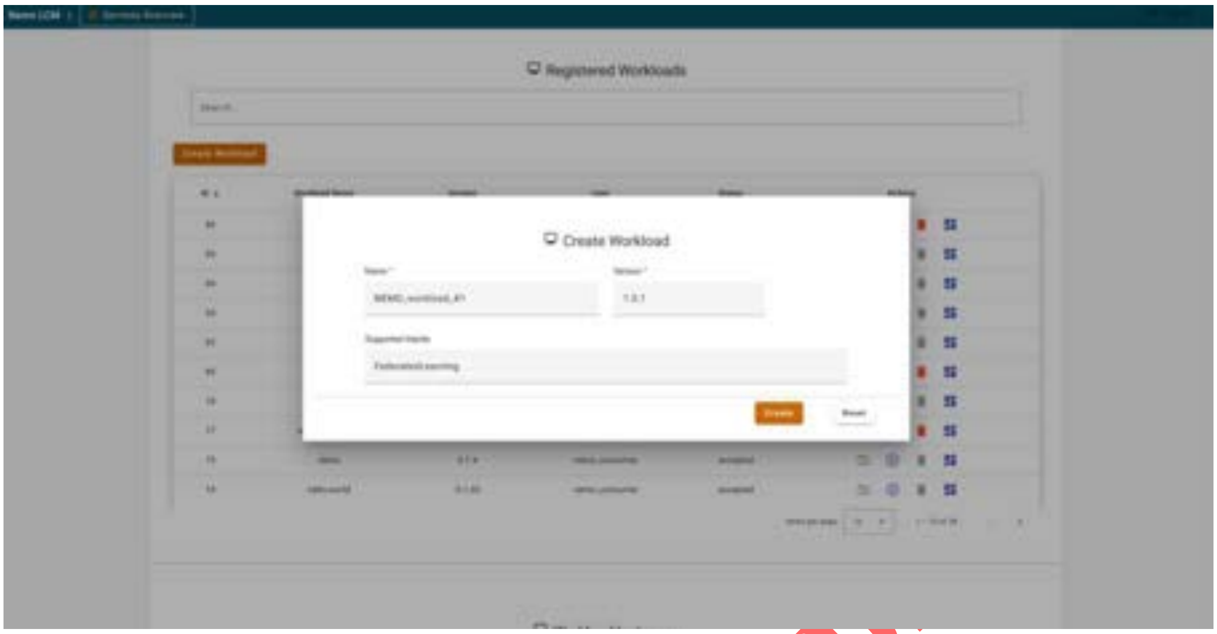
	○ Access Control
Features to be tested	To verify the workload registration, deployment and provisioning process in NEMO, the following features will be tested: • Workload Registration • Workload Deployment • Workload Provisioning The feature that this scenario aims to test are the workload registration process which is initiated by the NEMO consumer (workload provider) through the LCM UI & Intent-based API. Then, the newly registered workload is requested to be deployed into the NEMO meta-OS ecosystem by the MO. Finaly the workload provisioning process is triggered which is facilitated by the Intent-based API and the Access Control components. The results including the workload status are visualized to the user.
Test setup	The associated components are deployed in OneLab facilities (dev cluster 1 & staging cluster 2)
Steps	The steps identified in the associated sequence diagrams are listed below: 1. NEMO workload registration a. Workload registration by the NEMO user through the LCM UI b. Execution of workload validation process in Intent-based API c. Notification of the LCM UI about the status of the workload registration 2. NEMO workload deployment a. Workload deployment by the NEMO user through the LCM GUI b. Execution of workload validation in Intent-based API c. Communication of the deployment request to the LCM UI d. Communication of the deployment request to the MO e. Deployment operation process triggered by MO f. Request scheduling by the CFDRL component g. Deployment operation process executed by MO h. Communication and update of the deployment operation status to the Intent-API i. Visualization of the updated status to the LCM UI 3. NEMO workload provisioning a. NEMO workload provisioning is triggered by the Intent-based API b. NEMO Access Control workload setup c. NEMO Access Control Keycloak plugin functionality d. Performance resilience of Kong Plugin

4.2.2 Results

This section documents the process that is described in the scenario above step by step.

4.2.2.1 Workload registration through LCM GUI

The workload registration process is facilitated by the LCM component and its UI as it's described in section 3.2.4. The LCM component utilizes the Intent-based API provided functionality in order to realize the NEMO user triggered operations for the workload registration process. Figure 92, presents the form that corresponds to the workload registration and Figure 93 presents the list of the registered workloads.


Registered Workloads

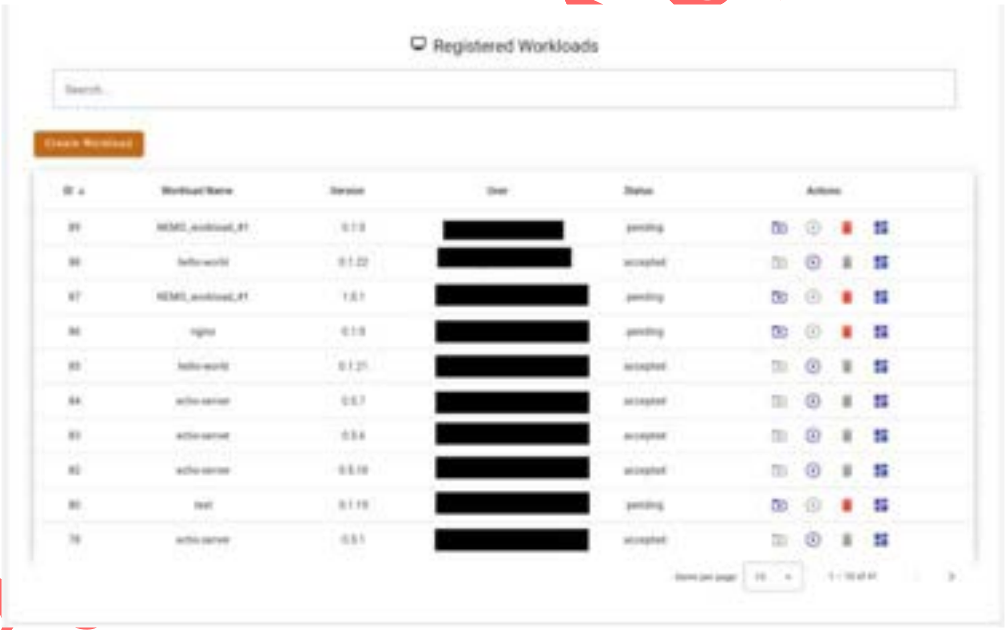
Create Workload

Name *: NEMO_workload_#1

Version *: 1.0.1

Supported Intents: Federated Learning

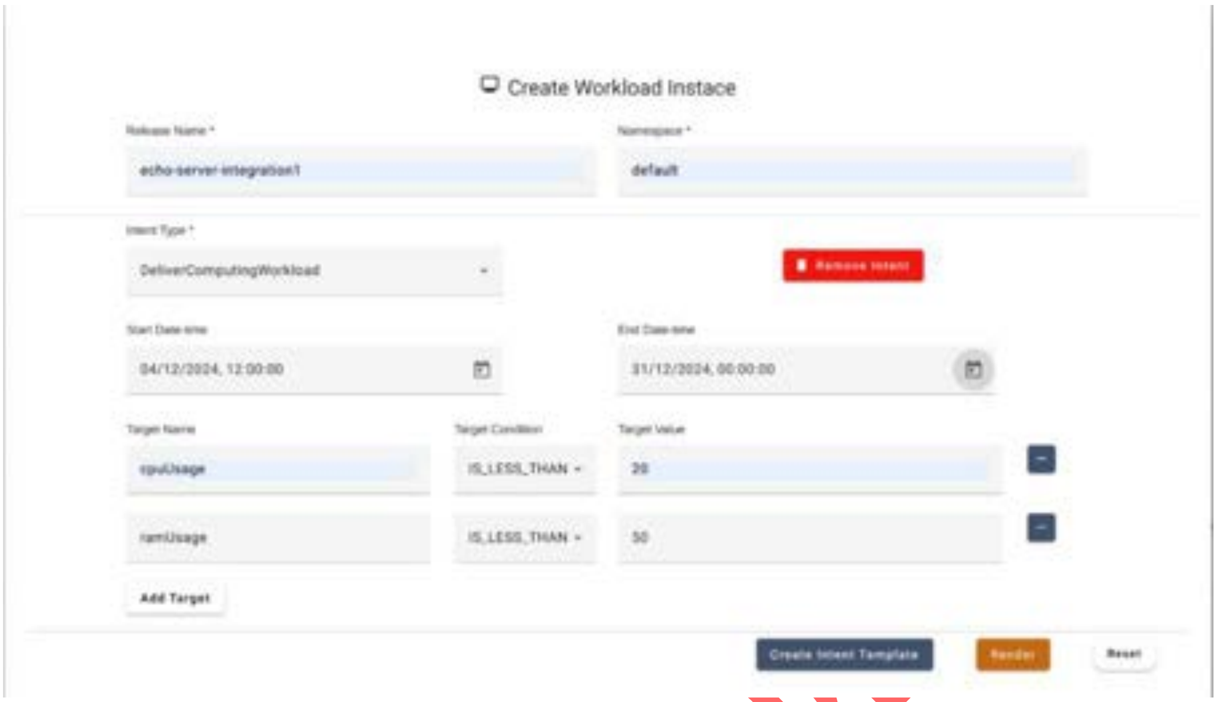
Figure 92: NEMO workload registration through LCM UI


Registered Workloads

ID	Workload Name	Version	User	Status
89	NEMO_workload_#1	0.1.0	[redacted]	pending
88	hello-world	0.1.22	[redacted]	accepted
87	NEMO_workload_#1	1.0.1	[redacted]	pending
86	nginx	0.1.0	[redacted]	pending
85	hello-world	0.1.21	[redacted]	accepted
84	echo-server	0.5.7	[redacted]	accepted
83	echo-server	0.5.6	[redacted]	accepted
82	echo-server	0.5.10	[redacted]	accepted
80	test	0.1.19	[redacted]	pending
79	echo-server	0.5.1	[redacted]	accepted

Figure 93: NEMO registered workloads

Intent Management provides the privileged user with the interfaces to create and manage intents. Figure 94 shows the create workload instance form.



Create Workload Instace

Release Name *	echo-server-integration1
Namespace *	default
Intent Type *	DeliverComputingWorkload
Start Date-time	04/12/2024, 12:00:00
End Date-time	31/12/2024, 00:00:00

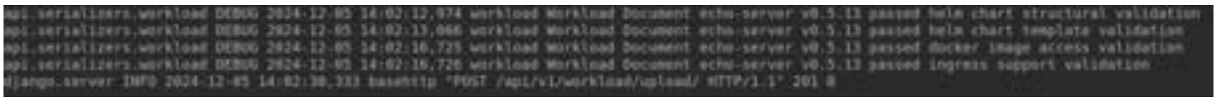
Target Name	Target Condition	Target Value
cpuUsage	IS_LESS_THAN	20
ramUsage	IS_LESS_THAN	50

Figure 94: NEMO workload instance creation (workload deployment process) through LCM UI

The newly created NEMO workload instance is visible in the *workload instances* table in LCM UI (Figure 95).

Figure 95: NEMO workload instances and their respective status in LCM UI

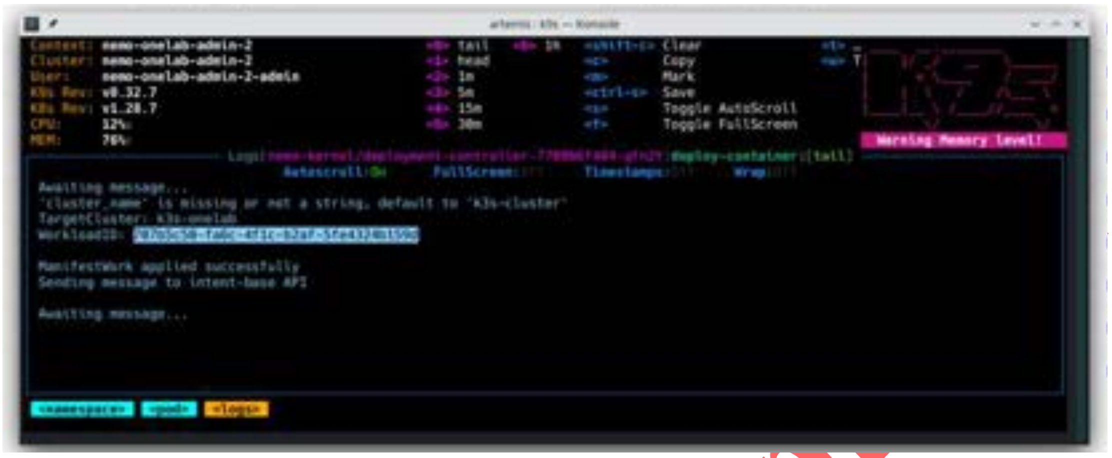
The workload is getting validated by the NEMO Workload Validator. Its functionality is described in section 3.4.2.5 which presents the validation checks that are performed by the validator. Once the tests have been successfully passed (Figure 96)then the workload upload request is dispatched.



```
api.serializers.workload DEBUG 2024-12-05 14:02:12,974 workload Workload Document echo-server v0.5.13 passed helm chart structural validation
api.serializers.workload DEBUG 2024-12-05 14:02:13,066 workload Workload Document echo-server v0.5.13 passed helm chart template validation
api.serializers.workload DEBUG 2024-12-05 14:02:16,725 workload Workload Document echo-server v0.5.13 passed docker image access validation
api.serializers.workload DEBUG 2024-12-05 14:02:16,726 workload Workload Document echo-server v0.5.13 passed ingress support validation
django.server INFO 2024-12-05 14:02:30,333 basehttp "POST /api/v1/workload/upload/ HTTP/1.1" 201 0
```

Figure 96: NEMO workload validation

Finally, the NEMO workload instance is deployed in the NEMO meta-OS platform as confirmed by the acknowledgment message received by the MO. The *workload ID* matches with the one that is visible in the LCM UI (Figure 97).

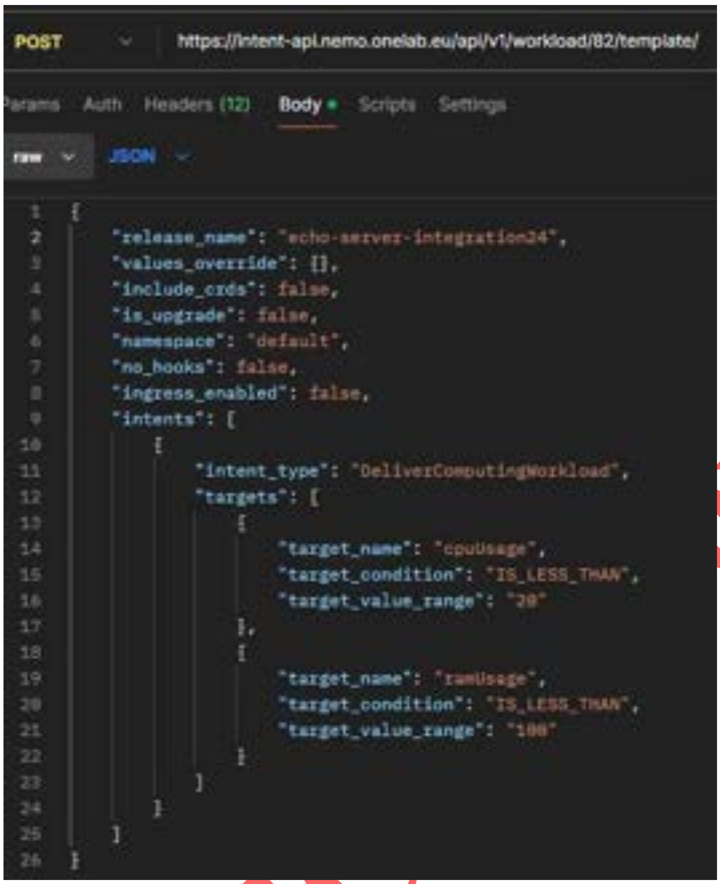


```
Context:  nemo-onelab-admin-2          <0> tail    <4> 1h   <shift-c> Clear
Cluster:  nemo-onelab-admin-2          <1> head             <c>       Copy
User:     nemo-onelab-admin-2-admin    <2> 1m               <m>       Mark
K9s Rev:  v0.32.7                      <3> 5m               <ctrl-s>  Save
K8s Rev:  v1.28.7                      <4> 15m              <s>       Toggle AutoScroll
CPU:      12%                          <5> 30m              <f>       Toggle FullScreen
MEM:      76%                                                                    Warning Memory level!
 Logs(nemo-kernel/deployment-controller-[illegible]:deploy-container)[tail]
      Autoscroll:On      FullScreen:Off      Timestamps:Off      Wrap:Off
 Awaiting message...
 'cluster_name' is missing or not a string, default to 'k3s-cluster'
 TargetCluster: k3s-onelab
 WorkloadID: [illegible]

 ManifestWork applied successfully
 Sending message to intent-base API

 Awaiting message...

 <namespace> <pod> <logs>
```

Figure 97: Workload deployment confirmation through RabbitMQ for the newly created workload instance

4.2.2.2 Meta-Orchestrator & Deployment Controller

The *Meta-Orchestrator (MO)* within its architecture has several subcomponents, including the *Deployment Controller (DC)*. This component handles communication, processes workload deployment configuration files, turns them into workloads' instances, and finally deploys those workloads' instances in a selected cluster. The *Intent-based API* sends the message with the workload to be deployed by the MO through RabbitMQ in JSON format, (Figure 98), and the message body created in the RabbitMQ queue, (Figure 99). The MO then processes that message, decoding it to adapt the JSON into a data structure within the programming language. It checks for metadata like labels and namespaces in the manifests and assigns defaults if they are missing.


POST https://intent-api.nemo.onelab.eu/api/v1/workload/82/template/

```
{
    "release_name": "echo-server-integration24",
    "values_override": [],
    "include_crds": false,
    "is_upgrade": false,
    "namespace": "default",
    "no_hooks": false,
    "ingress_enabled": false,
    "intents": [
        {
            "intent_type": "DeliverComputingWorkload",
            "targets": [
                {
                    "target_name": "cpuUsage",
                    "target_condition": "IS_LESS_THAN",
                    "target_value_range": "20"
                },
                {
                    "target_name": "ramUsage",
                    "target_condition": "IS_LESS_THAN",
                    "target_value_range": "100"
                }
            ]
        }
    ]
}
```

Figure 98: Intent-based API endpoint where the scenario starts.

Moreover from Figure 98 is important remark that the future workload has the name "*echo-server-integration24"* and it has a workload id with the value: "*bf4c24eb-c263-4854-b886-51b915d79264"*.

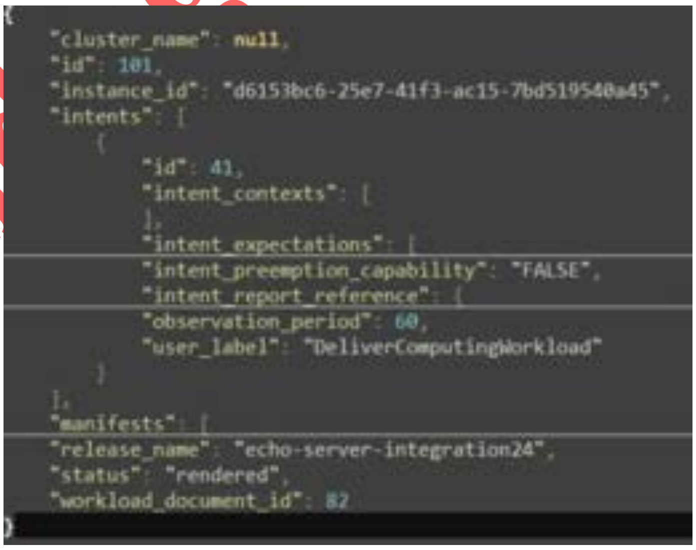
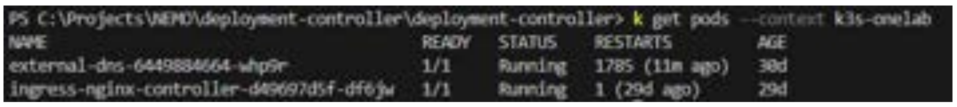

```
{
    "cluster_name": null,
    "id": 101,
    "instance_id": "d6153bc6-25e7-41f3-ac15-7bd519540a45",
    "intents": [
        {
            "id": 41,
            "intent_contexts": [
            ],
            "intent_expectations": [
            "intent_preemption_capability": "FALSE",
            "intent_report_reference": {
            "observation_period": 60,
            "user_label": "DeliverComputingWorkload"
        }
    ],
    "manifests": [
    "release_name": "echo-server-integration24",
    "status": "rendered",
    "workload_document_id": 82
}
```

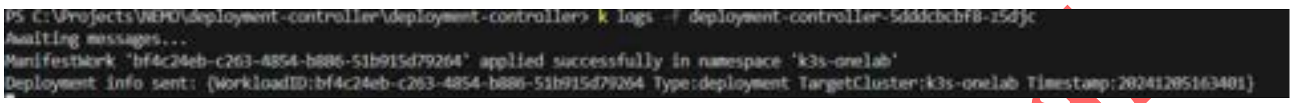
Figure 99: JSON published in RabbitMQ to be consume by MO.



```
PS C:\Projects\NEMO\deployment-controller\deployment-controller> k get pods --context k3s-onelab
NAME                                     READY   STATUS    RESTARTS       AGE
external-dns-6449884664-whp9r            1/1     Running   1785 (11m ago)   30d
ingress-nginx-controller-d49697d5f-df6jw 1/1     Running   1 (29d ago)      29d
```

Figure 100: Target cluster without the workload.

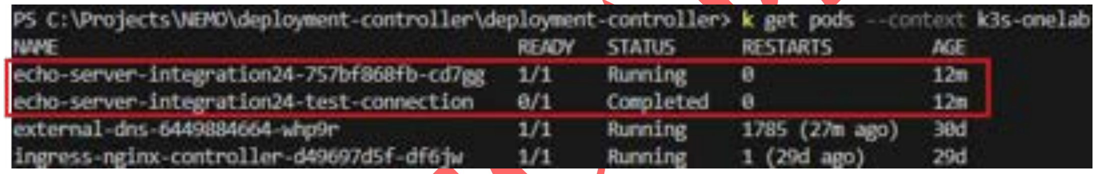
In Figure 100, before the workload deployment we can check that any workload is already deployed. Once the MO triggers the workload deployment, we can see in Figure 101 the pods of the workload "*echo-server-integration24"* deployed in the *k3s-onelab* as the target cluster chosen.



```
PS C:\Projects\NEMO\deployment-controller\deployment-controller> k logs -f deployment-controller-5dddcbcbf8-z5djc
Awaiting messages...
ManifestWork 'bf4c24eb-c263-4854-b886-51b915d79264' applied successfully in namespace 'k3s-onelab'
Deployment info sent: {WorkloadID:bf4c24eb-c263-4854-b886-51b915d79264 Type:deployment TargetCluster:k3s-onelab Timestamp:20241205163401}
```

Figure 101: Deployment Controller log, receiving the workload petition and deploying it.

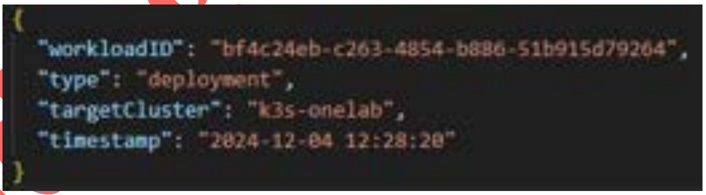
In parallel, the MO asks the CFDRL which cluster is best for deploying a workload; the CDFRL recommends a specific cluster in direct communication with the MO. The recommendation of the CFDRL which is the product of the inference that stems from AI model that CFDRL incorporates, was simulated as the development of the component is in progress. Once the MO has the workload ready and the CFDRL recommendation, the MO deploys the workload in the chosen cluster using the OCM libraries and the NEMO cluster network. The results are showed in Figure 102 up and running as pods.



```
PS C:\Projects\NEMO\deployment-controller\deployment-controller> k get pods --context k3s-onelab
NAME                                       READY   STATUS      RESTARTS         AGE
echo-server-integration24-757bf868fb-cd7gg 1/1     Running     0                12m
echo-server-integration24-test-connection  0/1     Completed   0                12m
external-dns-6449884664-whp9r              1/1     Running     1785 (27m ago)   30d
ingress-nginx-controller-d49697d5f-df6jw   1/1     Running     1 (29d ago)      29d
```

Figure 102: Workload already deployed in the cluster selected.

Finally, in Figure 103, the DC publishes the previous message into RabbitMQ to update the workload status. The Intent-Base API and other components will read this status.



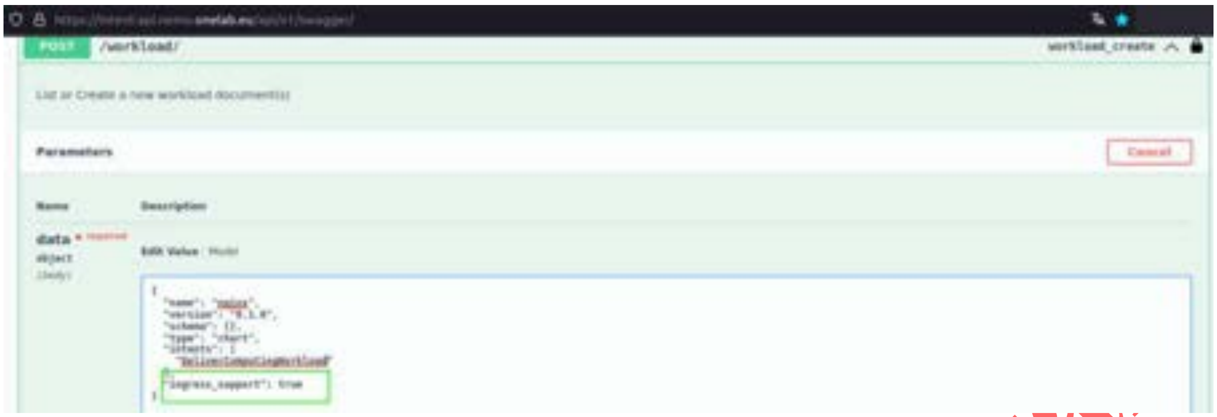
```
{
  "workloadID": "bf4c24eb-c263-4854-b886-51b915d79264",
  "type": "deployment",
  "targetCluster": "k3s-onelab",
  "timestamp": "2024-12-04 12:28:20"
}
```

Figure 103: Deployment Controller (MO) final response.

4.2.2.3 Access control provisioning

In deliverable D4.1 [2], the deployment and integration of the *NEMO Access Control* with the *Intent-based API* were presented. For the integration part, we had developed an API that would receive a payload with the necessary information to properly set the workload in the Access Control (set the Kong services, routes and plugins). In this section, we will present the improvements made to the Access Control workload provisioning to better automate and simplify the workflow.
During the initial creation of a workload, the Intent-based API offers the ability to choose whether to deploy the workload with an Ingress or not. For this scenario, we will create a workload for a simple NGINX[21] server (Figure 104).

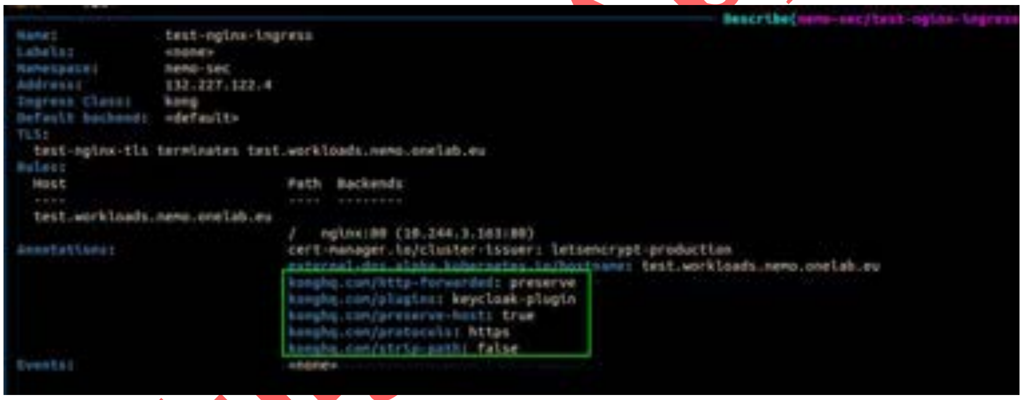
[21] https://nginx.org/



```
https://intent-api.nemo.onelab.eu/api/v1/swagger/
POST  /workload/                                            workload_create
List or Create a new workload document(s)
Parameters                                                          Cancel
Name        Description
data * required
object      Edit Value | Model
(body)
            {
              "name": "nginx",
              "version": "0.1.0",
              "schema": {},
              "type": "chart",
              "intents": [
                "DeliverComputingWorkload"
              ],
              "ingress_support": true
            }
```

Figure 104: Create workload through Intent-based API with Ingress support

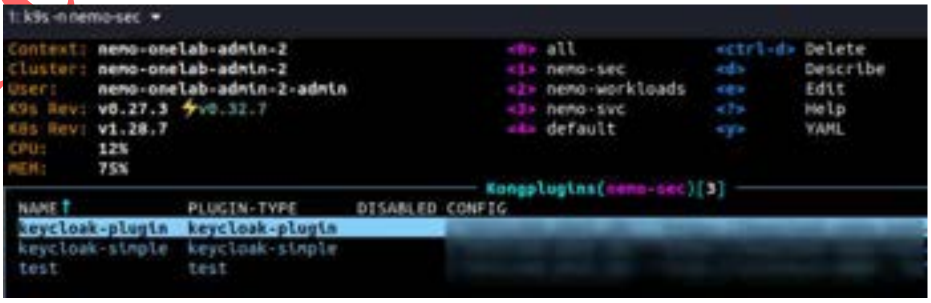
When the workload is successfully created, the *Intent-based API* produces an Ingress that holds specific annotations to integrate with the NEMO *Access Control*, and enable the *oAuth 2.0 plugin* for limiting access to unauthorized users. Figure 105 shows the annotations added to the Ingress to specify details for the services and routes that will be created in the Access Control (*konghq.com/protocols*, *konghq.com/http-forwarded etc*), as well as the plugins that will be applied to the deployment workload (*konghq.com/plugins*).



```
                                                           Describe(nemo-sec/test-nginx-ingress
Name:             test-nginx-ingress
Labels:           <none>
Namespace:        nemo-sec
Address:          132.227.122.4
Ingress Class:    kong
Default backend:  <default>
TLS:
  test-nginx-tls terminates test.workloads.nemo.onelab.eu
Rules:
  Host                             Path  Backends
  ----                             ----  --------
  test.workloads.nemo.onelab.eu
                                   /   nginx:80 (10.244.3.163:80)
Annotations:                       cert-manager.io/cluster-issuer: letsencrypt-production
                                   external-dns.alpha.kubernetes.io/hostname: test.workloads.nemo.onelab.eu
                                   konghq.com/http-forwarded: preserve
                                   konghq.com/plugins: keycloak-plugin
                                   konghq.com/preserve-host: true
                                   konghq.com/protocols: https
                                   konghq.com/strip-path: false
Events:                            <none>
```

Figure 105: Workload Ingress annotations for integrating with Access Control

The *oAuth 2.0* configuration (named *keycloak-plugin*) is deployed in the OneLab cluster as a *Kongplugin* [61] resource. This allows for the plugin to already be configured and ready to apply to new Ingresses. Figure 106 shows the Kong plugins resources available in the OneLab cluster.



```
1: k9s -n nemo-sec
Context:  nemo-onelab-admin-2                    <0> all             <ctrl-d> Delete
Cluster:  nemo-onelab-admin-2                    <1> nemo-sec        <d>      Describe
User:     nemo-onelab-admin-2-admin              <2> nemo-workloads  <e>      Edit
K9s Rev:  v0.27.3 ⚡v0.32.7                      <3> nemo-svc        <?>      Help
K8s Rev:  v1.28.7                                <4> default         <y>      YAML
CPU:      12%
MEM:      75%
                              Kongplugins(nemo-sec)[3]
 NAME↑            PLUGIN-TYPE       DISABLED CONFIG
 keycloak-plugin  keycloak-plugin
 keycloak-simple  keycloak-simple
 test             test
```

Figure 106: The Onelab KongPlugin resources

When the Ingress is successfully deployed to the cluster, the Access Control is automatically updated with the specifications in the Ingress annotations. Figure 107 and Figure 108 show the Access Control service that was created and its details, respectively.

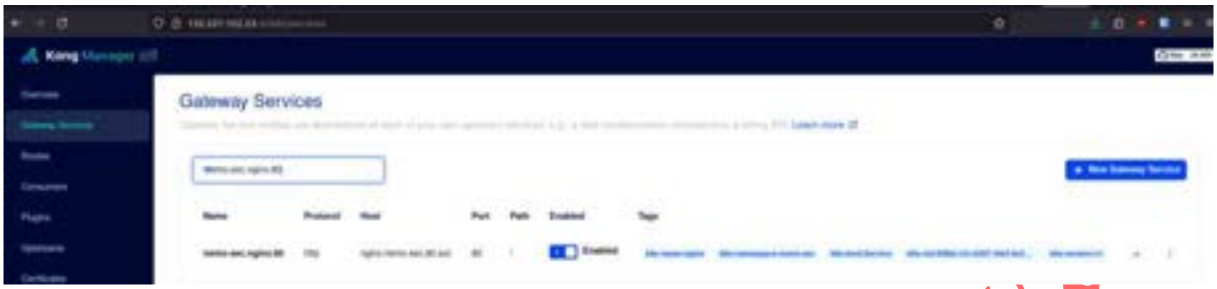
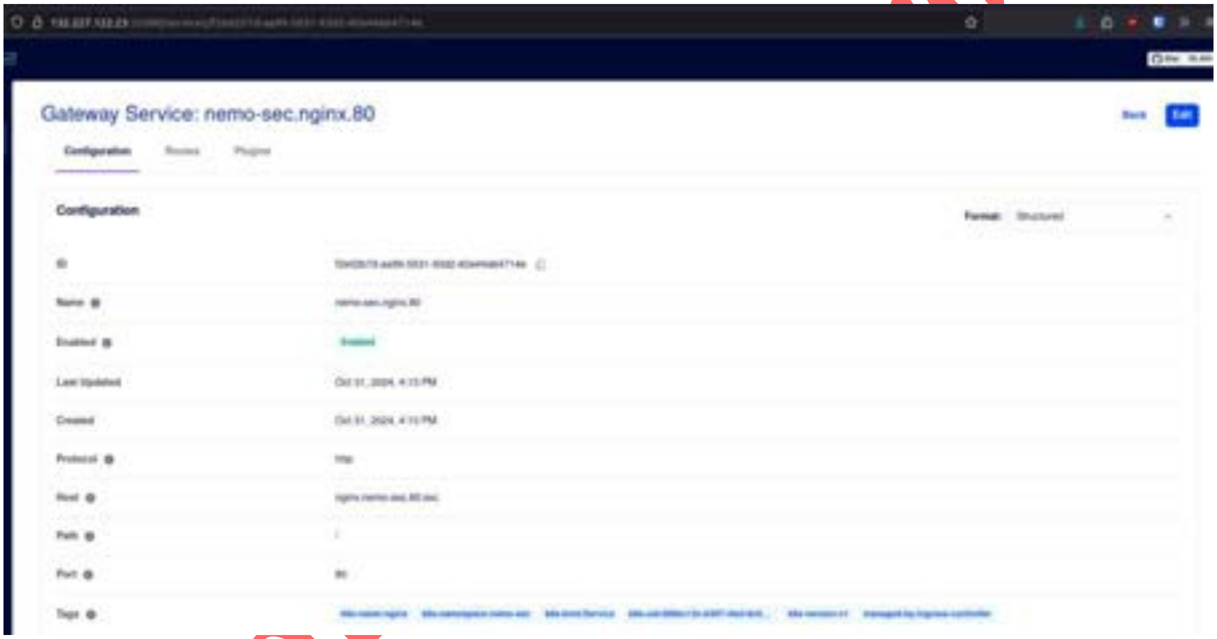

Gateway Services

Figure 107: Workload Access Control service


Gateway Service: nemo-sec.nginx.80

Configuration

ID	[illegible]
Name	nemo-sec.nginx.80
Enabled	Enabled
Last Updated	Oct 31, 2024, 4:15 PM
Created	Oct 31, 2024, 4:15 PM
Protocol	http
Host	[illegible]
Path	/
Port	80
Tags	[illegible]

Figure 108: Workload Access Control service details

Figure 109 and Figure 110 show the Access Control route that was created and its details, respectively. In both the service and the route details, we can see that the name of the workload, its path and its Ingress host name have all been registered successfully.

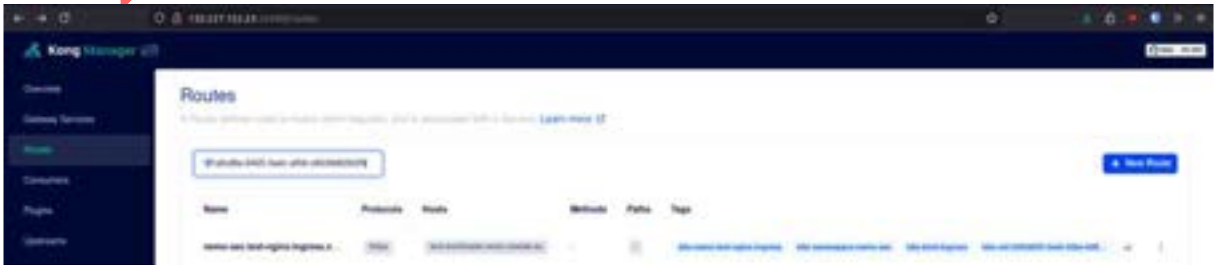
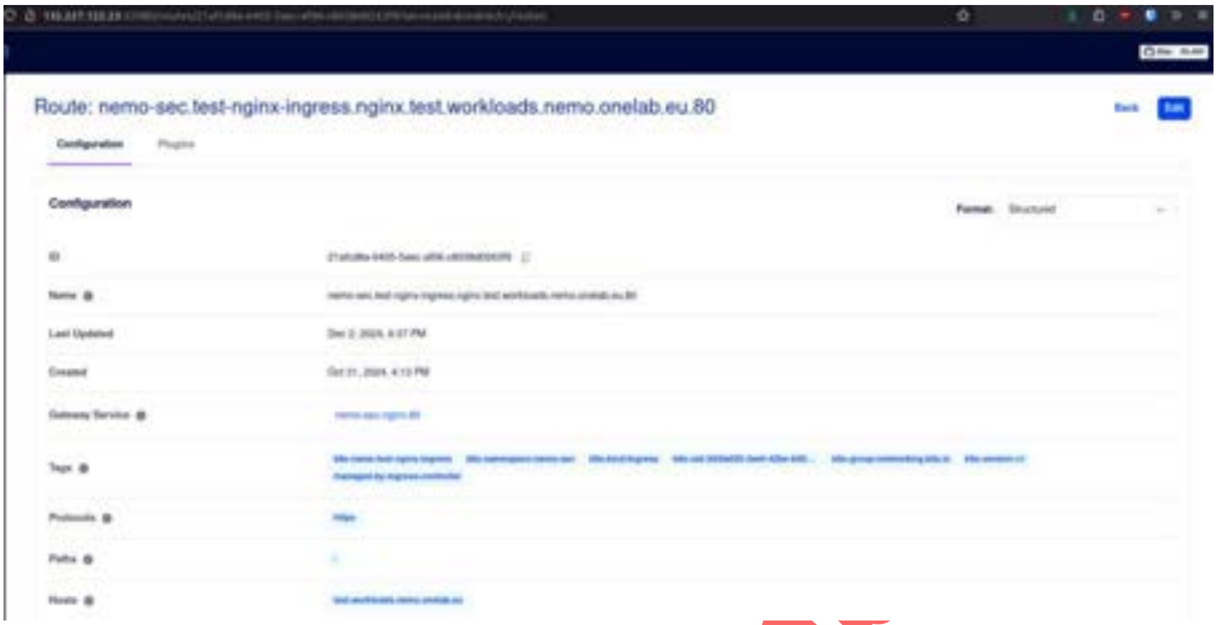

Routes

Figure 109: Workload Access Control route

Document name:	D4.2 Advanced NEMO platform & laboratory testing results. Initial version					**Page:**	96 of 146
Reference:	D4.2	**Dissemination:**	PU	**Version:**	1.0	**Status:**	final



Route: nemo-sec.test-nginx-ingress.nginx.test.workloads.nemo.onelab.eu.80

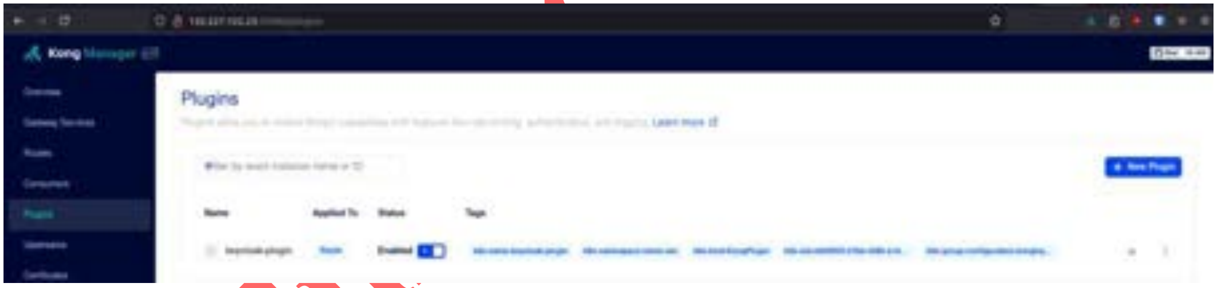
Configuration | Plugins

Configuration — Format: Structured

ID	[illegible]
Name	nemo-sec.test-nginx-ingress.nginx.test.workloads.nemo.onelab.eu.80
Last Updated	Dec 2, 2024, 4:07 PM
Created	Oct 31, 2024, 4:13 PM
Gateway Service	nemo-sec-nginx.80
Tags	[illegible] managed-by-ingress-controller
Protocols	https
Paths	/
Hosts	test.workloads.nemo.onelab.eu

Figure 110: Workload Access Control route details

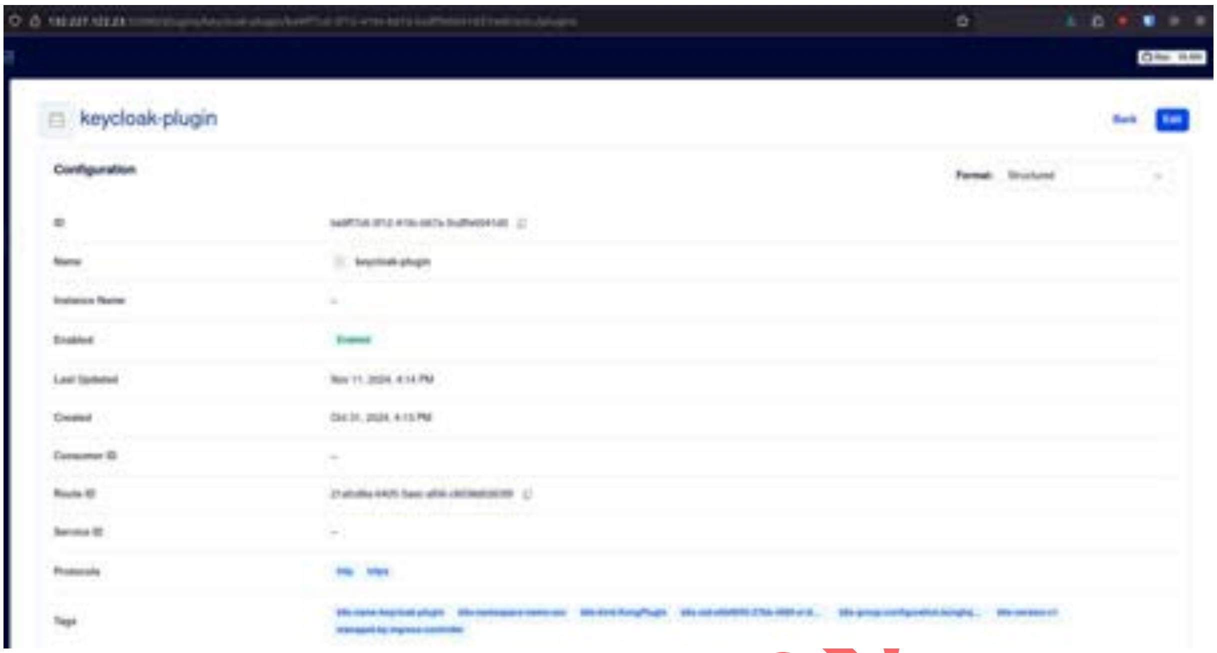
Finally, Figure 111, Figure 112 and Figure 113 present the oAuth2.0 plugin and its details.



Plugins

Name	Applied To	Status	Tags
keycloak-plugin	Route	Enabled	[illegible]

Figure 111: Workload oAuth2.0 plugin

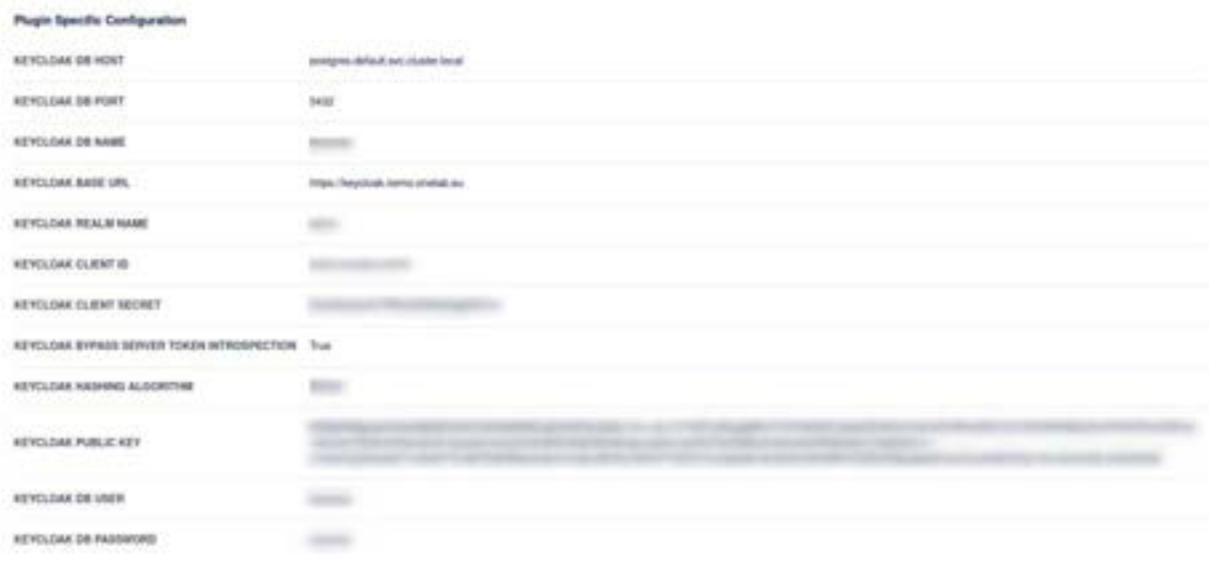


keycloak-plugin

Configuration

ID	[illegible]
Name	keycloak-plugin
Instance Name	–
Enabled	Enabled
Last Updated	Nov 11, 2024, 4:14 PM
Created	Oct 31, 2024, 4:15 PM
Consumer ID	–
Route ID	[illegible]
Service ID	–
Protocols	http https
Tags	[illegible]

Figure 112: Workload oAuth2.0 plugin details



Plugin Specific Configuration

KEYCLOAK DB HOST	postgres.default.svc.cluster.local
KEYCLOAK DB PORT	5432
KEYCLOAK DB NAME	[redacted]
KEYCLOAK BASE URL	https://keycloak.nemo.onelab.eu
KEYCLOAK REALM NAME	[redacted]
KEYCLOAK CLIENT ID	[redacted]
KEYCLOAK CLIENT SECRET	[redacted]
KEYCLOAK BYPASS SERVER TOKEN INTROSPECTION	True
KEYCLOAK HASHING ALGORITHM	[redacted]
KEYCLOAK PUBLIC KEY	[redacted]
KEYCLOAK DB USER	[redacted]
KEYCLOAK DB PASSWORD	[redacted]

Figure 113: Workload oAuth2.0 plugin (cont'd)

Now that the NEMO Access Control has been configured properly, we can test how the access to the workload works. In order to access the protected resource, all the requests should provide an *authorization* header with a Bearer token from the *NEMO Identity Management component*. The oAuth2.0 plugin is configured to test the provided token and, depending on its validity, deny or grant access. In Figure 114, the request provides no authorization header, therefore the user is denied access.

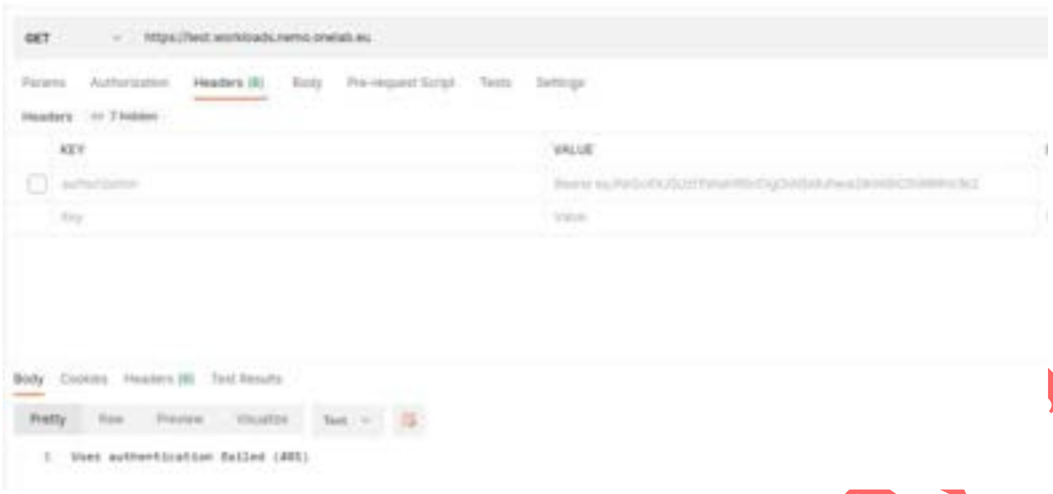

```
GET  https://test.workloads.nemo.onelab.eu
Params  Authorization  Headers (8)  Body  Pre-request Script  Tests  Settings
Headers  7 hidden
KEY             VALUE
authorization   Bearer [illegible]
Key             Value

Body  Cookies  Headers (8)  Test Results
Pretty  Raw  Preview  Visualize  Text
1  User authentication failed (401)
```

Figure 114: NEMO oAuth2.0 plugin test

In Figure 115, even if the token is provided, it could have expired, been tampered with or come from an unknown source. The plugin, again, denies access to the NGINX server.

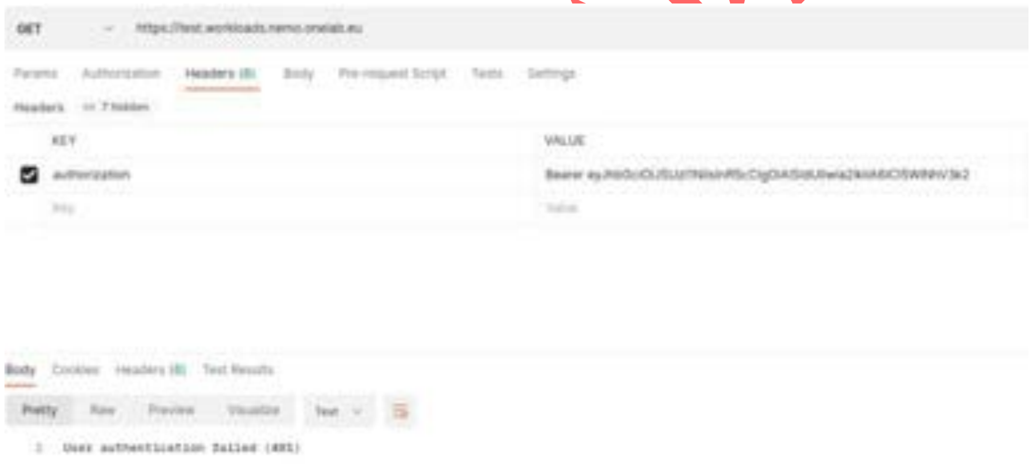
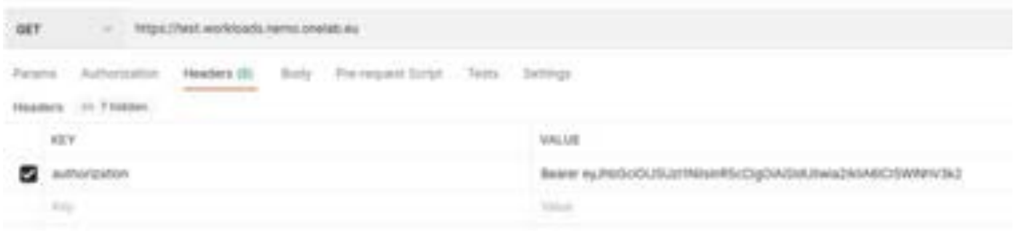

```
GET  https://test.workloads.nemo.onelab.eu
Params  Authorization  Headers (8)  Body  Pre-request Script  Tests  Settings
Headers  7 hidden
KEY                  VALUE
☑ authorization      Bearer [illegible]
Key                  Value

Body  Cookies  Headers (8)  Test Results
Pretty  Raw  Preview  Visualize  Text
1  User authentication failed (401)
```

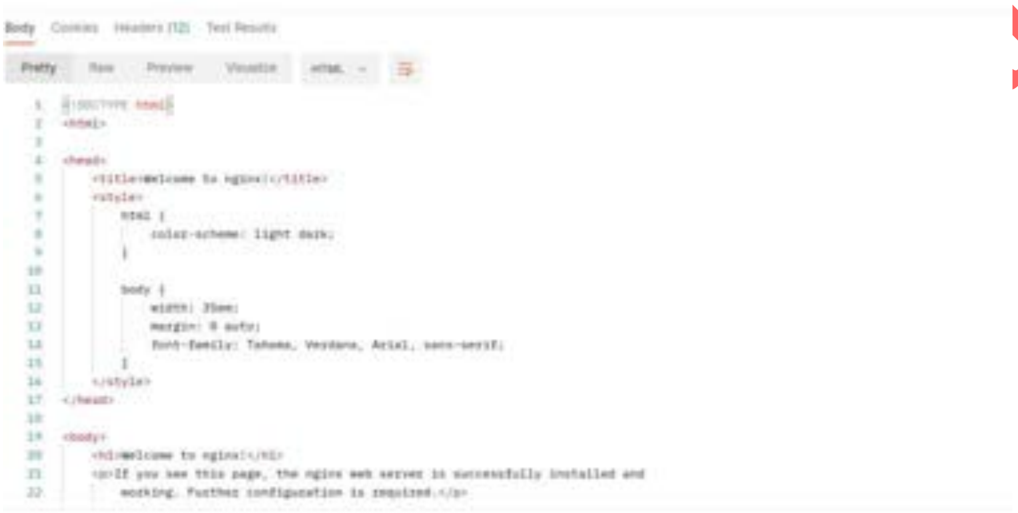
Figure 115: oAuth2.0plugin test - expired or false token

Finally, if the token is valid, the user can view the resource, in this case the welcome page of the NGINX server (Figure 116).



KEY	VALUE
authorization	Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJ5WWNV3k2
Key	Value

https://test.workloads.nemo.onelab.eu



```
<!DOCTYPE html>
<html>

<head>
    <title>Welcome to nginx!</title>
    <style>
        html {
            color-scheme: light dark;
        }

        body {
            width: 35em;
            margin: 0 auto;
            font-family: Tahoma, Verdana, Arial, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Welcome to nginx!</h1>
    <p>If you see this page, the nginx web server is successfully installed and
        working. Further configuration is required.</p>
```

Figure 116: oAuth2.0 plugin test - success

4.2.2.4 NEMO Access Control plugin response performance

The plugin, in order to reduce the time needed to validate the provided token, instead of using an HTTP request to connect to the *NEMO Identity Management component*, it connects directly to its database to perform the user token validation.

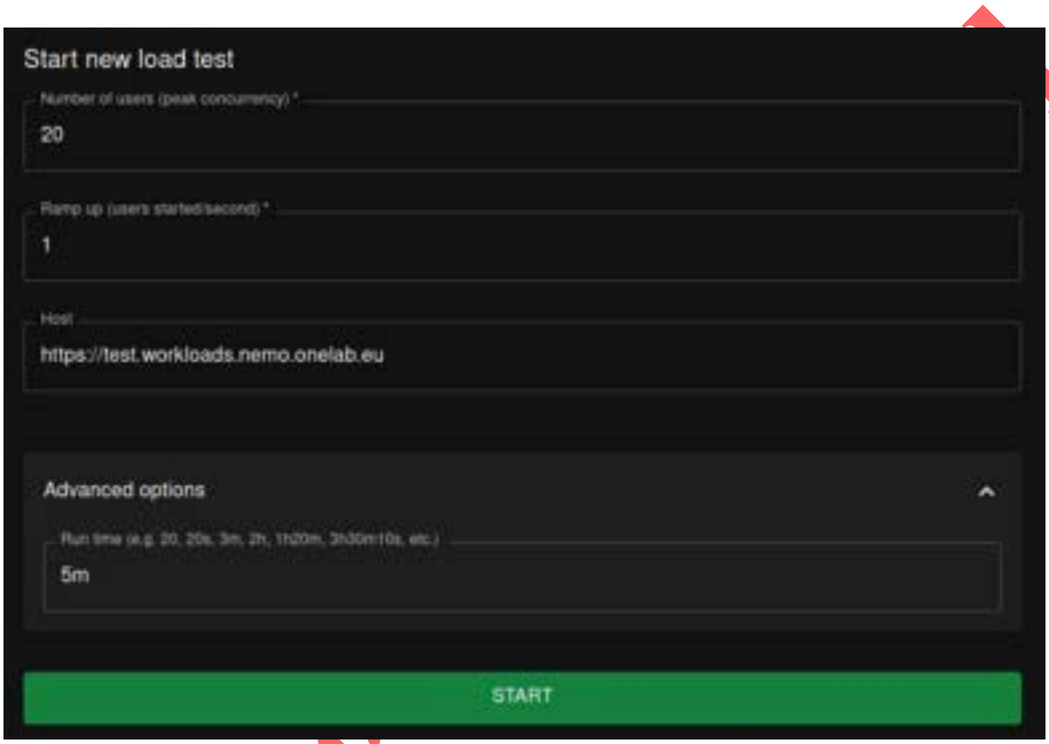
This decision has come from studying the code of the Identity Management component for the token introspection[22]. The outcome of this study indicated that the Identity Management component performs several checks which are not necessary for the project use cases. These can add significantly to the plugin response time, particularly under load-testing scenarios. Therefore, the oAuth2.0 implementation with the direct database connection performs only the necessary checks to verify the *token*, *user*, *realm* and *client_id* validity.

Notably, in order to increase the overall system performance and redundancy and avoid security risks that stem from directly connecting to the Identity Management database, the latter has been configured with streaming replication (WAL). Under this framework, the plugin does not directly connect to the master instantiation of the database itself but, instead, it connects to a ready-only live replica. Additionally, the database user used by the plugin has been configured with restricted access to the tables that are strictly necessary for the token introspection.

We will now present the performance of the plugin when it directly connects to the database, versus when it uses requests to perform the token validation. To conduct the performance tests, we are using

22 https://github.com/keycloak/keycloak/blob/83f8622d15d9a3559ee6d99a4c57033190a5392d/services/src/main/java/org/keycloak/protocol/oidc/endpoints/TokenIntrospectionEndpoint.java#L72

Locust[23], an open-source tool, which gives information on metrics, like the percentiles of the response times, the number of requests, minimum, max and average request times. For both experiments we will use the same setup parameters (Figure 117): (a) the max number of users will reach 20, the users will increase every one second and (b) each experiment will last 5 minutes. Each request will be executed randomly between the span of 1 and 5 seconds. The first run will use the implementation of the oAuth2.0 plugin that make a direct connection to the database (*standard oAuth2.0 plugin*), while in the second one we will apply to the same Ingress the version of the plugin which makes requests to the Identity Management API (*simplified oAuth2.0 plugin*).



Start new load test

Number of users (peak concurrency) *: 20

Ramp up (users started/second) *: 1

Host: https://test.workloads.nemo.onelab.eu

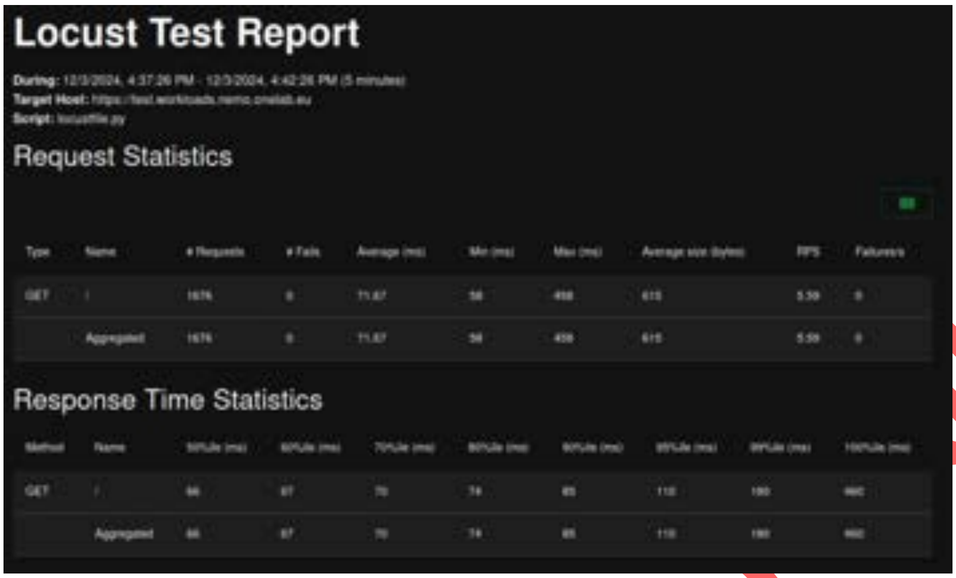
Advanced options

Run time (e.g. 20, 20s, 3m, 2h, 1h20m, 3h30m10s, etc.): 5m

Figure 117: Locust experiments' setup

Figure 118 and Figure 119 present the request and response statistics of Locust for the standard plugin. In the request statistics table, we can see the total number of requests, the average, minimum and max times of the requests. If we compare the two averages, we can see that the average request time of the simplified plugin is approximately four times larger than the standard implementation. The percentiles, also, in the response statistics table, show that the simplified version has greater response times in comparison.

[23] https://locust.io



Locust Test Report

During: 12/3/2024, 4:37:26 PM - 12/3/2024, 4:42:26 PM (5 minutes)
Target Host: https://test.workloads.nemo.onelab.eu
Script: locustfile.py

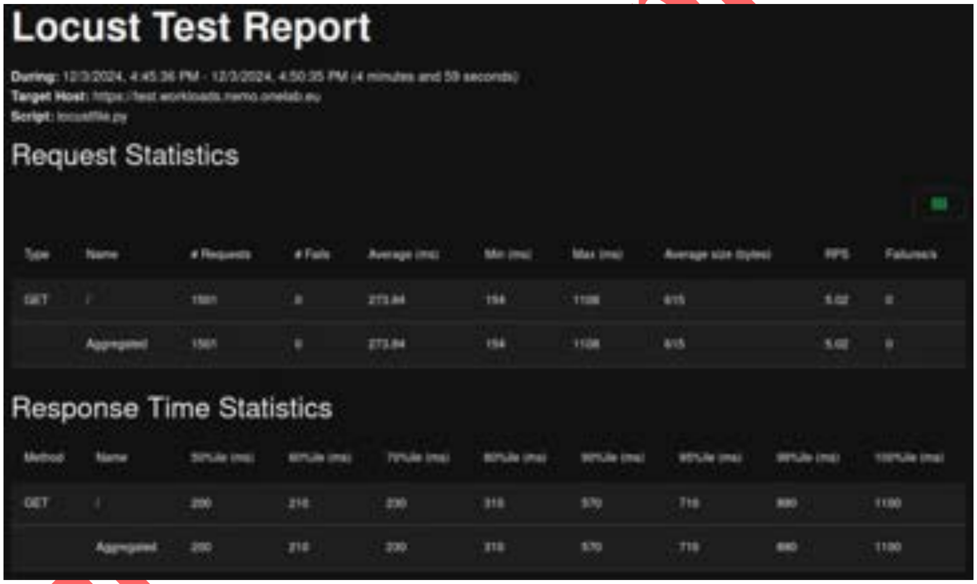
Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/	1676	0	71.67	58	458	615	5.59	0
	Aggregated	1676	0	71.67	58	458	615	5.59	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/	66	67	70	74	85	110	190	460
	Aggregated	66	67	70	74	85	110	190	460

Figure 118: Locust request and response statistics for standard *oAuth2.0* plugin



Locust Test Report

During: 12/3/2024, 4:45:36 PM - 12/3/2024, 4:50:35 PM (4 minutes and 59 seconds)
Target Host: https://test.workloads.nemo.onelab.eu
Script: locustfile.py

Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/	1501	0	273.84	154	1108	615	5.02	0
	Aggregated	1501	0	273.84	154	1108	615	5.02	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/	200	210	230	310	570	710	880	1100
	Aggregated	200	210	230	310	570	710	880	1100

Figure 119: Locust request and response statistics for simplified *oAuth2.0* plugin

Figure 120 compares side by side the two runs. If we compare the response times, we can see that in the first run, the plugin requires less time to stabilize the response times when the number of users have peaked for the experiment. During the first execution the response times were almost half of the second one. Finally, the deviation of the 50th and 95th percentiles in the first run has a smaller deviation, meaning that the majority of the users will experience a stable experience.

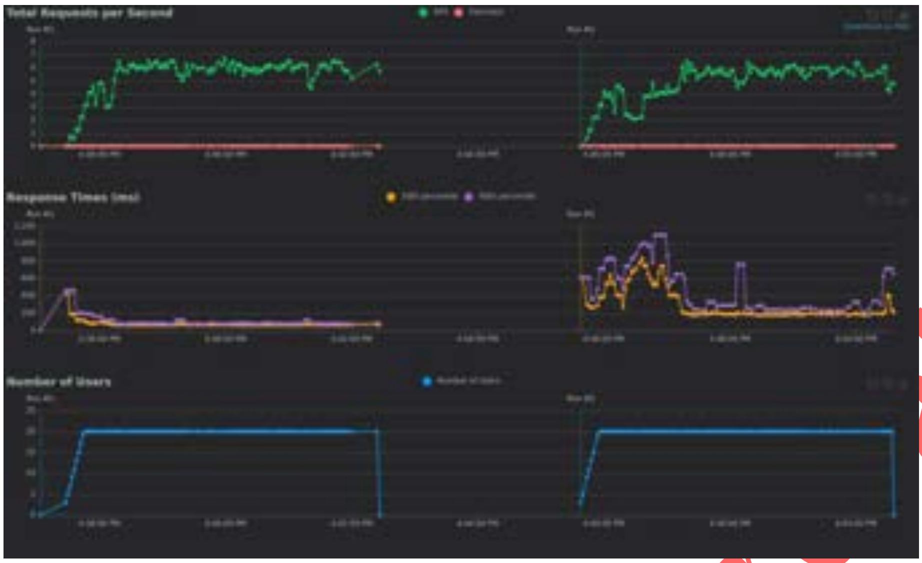


Figure 120: Locust charts for the *oAuth2.0* plugin implementations

4.2.3 Verification summary checklist

Checklist for Test2: NEMO workload registration, deployment and provisioning				
		Yes	No	Comments
1	NEMO workload registration by the NEMO user through LCM UI	✓		Success
2	NEMO workload registration through the Intent-based API	✓		Success
3	Notification of the LCM UI about the status of the workload registration	✓		Success
4	Execution of workload validation in Intent-based API	✓		Success
5	NEMO workload deployment by the NEMO user through LCM UI	✓		Success
6	Communication of the deployment acknowledgement to LCM UI	✓		Success
7	Communication of the deployment request to the MO	✓		Success
8	CFDRL deployment recommendation to MO		✓	Simulated step
9	Deployment operation process executed by the MO	✓		Success
10	Communication and update of the deployment operation status to the Intent-based API	✓		Success
11	Visualization of the updated status to the LCM UI	✓		Success
12	NEMO workload provisioning triggered by the Intent-based API	✓		Success
13	NEMO Access Control workload setup process	✓		Success
14	NEMO Access Control Keycloak plugin functionality executed	✓		Success
15	Performance resilience of the Kong Plugin	✓		Success

Table 10: Checklist for workload registration, deployment and provisioning scenario

4.3 NEMO workload migration

This integration scenario aims to illustrate the workload migration process. The process is facilitated by the Intent-based API, the MO and the Intent-based Migration Controller (IBMC). Figure 121, depicts the steps executed to complete the task.

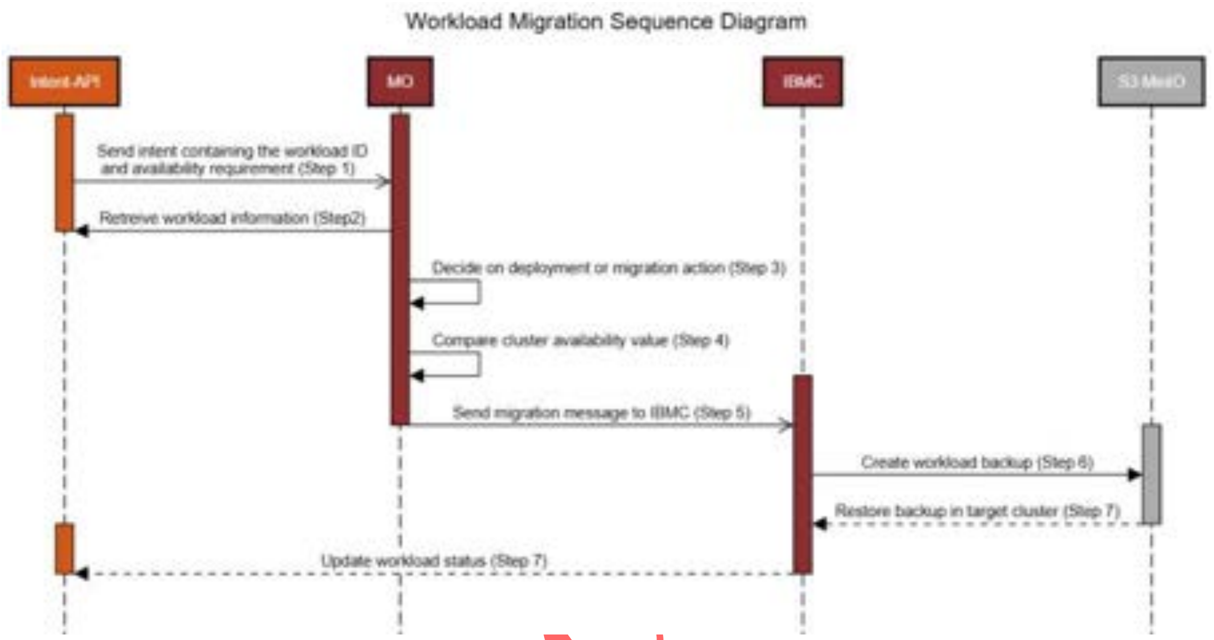

Workload Migration Sequence Diagram
Intent-API
MO
IBMC
S3 MinIO
Send intent containing the workload ID and availability requirement (Step 1)
Retrieve workload information (Step2)
Decide on deployment or migration action (Step 3)
Compare cluster availability value (Step 4)
Send migration message to IBMC (Step 5)
Create workload backup (Step 6)
Restore backup in target cluster (Step 7)
Update workload status (Step 7)

Figure 121: NEMO workload migration sequence diagram

4.3.1 Verification scenario

Test 3: NEMO workload migration	
Objective	The objective of this task is to validate the NEMO workload migration process.
Components	IBMC Intent-API MO MinIO
Features to be tested	The NEMO workload migration process is triggered by the CFDRL component once its inference states that it is preferable for the workload's optimal operation to be moved from cluster A to cluster B. Once the request is communicated to the Meta-Orchestrator component then the workload migration is executed.
Test setup	All the participating components were deployed in the NEMO meta-OS cloud/edge infrastructure at OneLab (dev cluster 1 and staging cluster 1).
Steps	1. Intent-API publishes a workload intent with an availability requirement. 2. MO retrieves workload status from the Intent-API.

	3. If the workload is already deployed in any cluster, then a migration action is triggered. 4. Check cluster availability. 5. MO sends a message to the IBMC containing the workload ID and the target cluster. 6. IBMC backs up the workload resources and uploads them MinIO. 7. IBMC restores the resources in the target cluster. After the resources get restored, the workload is removed from the source cluster. 8. The IBMC sends a message to the Intent-API updating the workload status, specifying the cluster where it has been deployed.

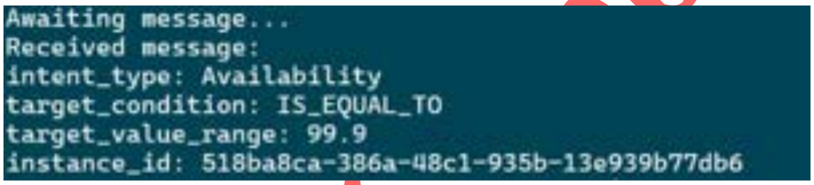
Table 11: Test 3 - NEMO workload migration

4.3.2 Results

The section below presents the steps that concern the execution of the workload migration integration workflow.

4.3.2.1 Step 1

An intent is published by the Intent-Based API in RabbitMQ and reaches the Meta-Orchestrator:



```
Awaiting message...
Received message:
intent_type: Availability
target_condition: IS_EQUAL_TO
target_value_range: 99.9
instance_id: 518ba8ca-386a-48c1-935b-13e939b77db6
```

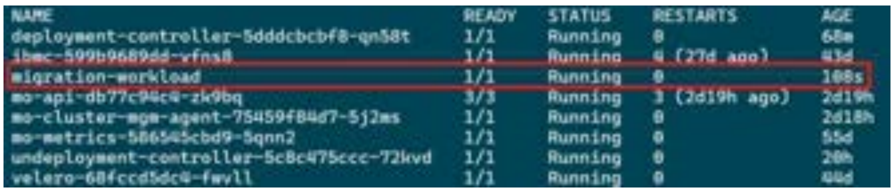
Figure 122: Intent message reaches MO

4.3.2.2 Step 2 & 3

The Meta-Orchestrator sends a query back to the Intent-Based API to retrieve a json with the workload status. As shown in the following code snippet, the workload appears to be already deployed in oneLab cluster, hence a migration action will take place:

```
{
"id": 6,
"instance_id": "a3177d01-863d-415b-a998-180c87113z50",
"workload_document_id": 9,
"release_name": "migration-workload",
"status": "deployed",
"manifests": [],
"cluster_name": onelab
}
```

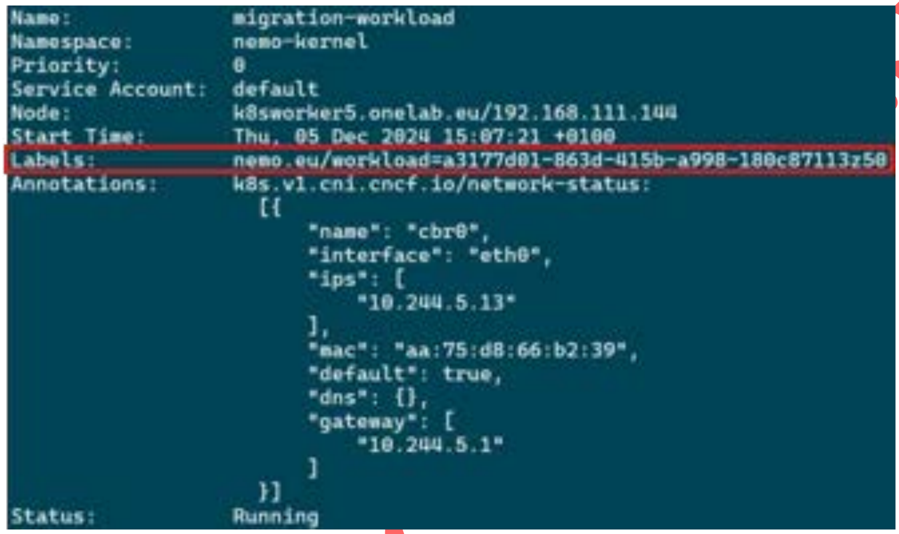
This can be verified in the oneLab cluster by executing `kubectl get pods --context onelab`. The list of pods shows the workload running:



```
NAME                                       READY   STATUS    RESTARTS        AGE
deployment-controller-5dddcbcbf8-qn58t     1/1     Running   0               68m
ibmc-599b9689dd-vfns8                      1/1     Running   4 (27d ago)     43d
migration-workload                         1/1     Running   0               108s
mo-api-db77c94c4-zk9bq                     3/3     Running   3 (2d19h ago)   2d19h
mo-cluster-mgm-agent-75459f84d7-5j2ms      1/1     Running   0               2d18h
mo-metrics-586545cbd9-5qnn2                1/1     Running   0               55d
undeployment-controller-5c8c475ccc-72kvd   1/1     Running   0               20h
velero-68fccd5dc4-fwvll                    1/1     Running   0               44d
```

Figure 123: Pods currently running in onelab

When `kubectl` describe pod migration-workload is executed, the workload ID can be found in the "labels" metadata:

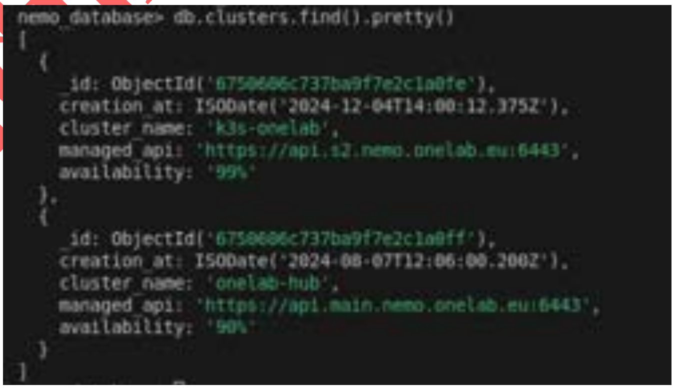


```
Name:             migration-workload
Namespace:        nemo-kernel
Priority:         0
Service Account:  default
Node:             k8sworker5.onelab.eu/192.168.111.144
Start Time:       Thu, 05 Dec 2024 15:07:21 +0100
Labels:           nemo.eu/workload=a3177d01-863d-415b-a998-180c87113z50
Annotations:      k8s.v1.cni.cncf.io/network-status:
                    [{
                        "name": "cbr0",
                        "interface": "eth0",
                        "ips": [
                            "10.244.5.13"
                        ],
                        "mac": "aa:75:d8:66:b2:39",
                        "default": true,
                        "dns": {},
                        "gateway": [
                            "10.244.5.1"
                        ]
                    }]
Status:           Running
```

Figure 124: Workload ID inspection

4.3.2.3 Step 4

The MO proceeds to check its internal database, which contains the availability information of every cluster. The availability value specified in the intent is compared with the one from the cluster where the workload is currently deployed. As seen in the Figure 125, the availability of the OneLab cluster (90%) is lower than the one required (99.9%). This will trigger the migration of the workload to a more suitable cluster. In this case, this is the k3s one.

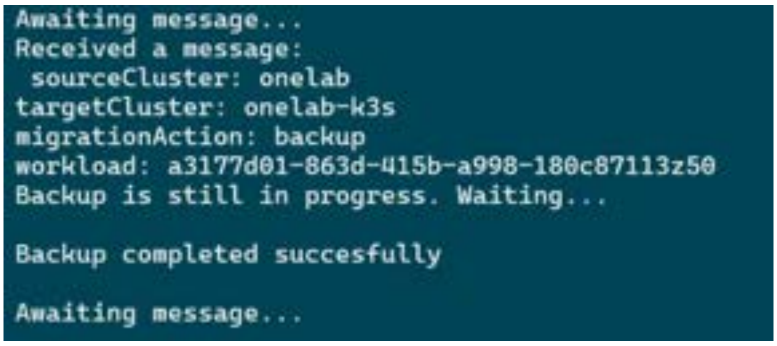


```
nemo_database> db.clusters.find().pretty()
[
  {
    _id: ObjectId('6750606c737ba9f7e2c1a0fe'),
    creation_at: ISODate('2024-12-04T14:00:12.375Z'),
    cluster_name: 'k3s-onelab',
    managed_api: 'https://api.s2.nemo.onelab.eu:6443',
    availability: '99%'
  },
  {
    _id: ObjectId('6750606c737ba9f7e2c1a0ff'),
    creation_at: ISODate('2024-08-07T12:06:00.200Z'),
    cluster_name: 'onelab-hub',
    managed_api: 'https://api.main.nemo.onelab.eu:6443',
    availability: '90%'
  }
]
```

Figure 125: Availability check

4.3.2.4 Step 5 & 6

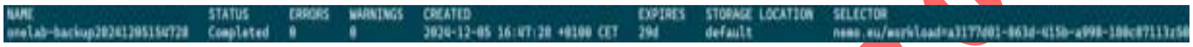
The MO sends a message via RabbitMQ to the IBMC containing the source and target clusters for the migration and the workload to be migrated. When the message reaches the IBMC, the migration process begins with the backup of the workload's resources, which is stored in the OneLab' MinIO instance.



```
Awaiting message...
Received a message:
 sourceCluster: onelab
targetCluster: onelab-k3s
migrationAction: backup
workload: a3177d01-863d-415b-a998-180c87113z50
Backup is still in progress. Waiting...

Backup completed succesfully

Awaiting message...
```

Figure 126: Migration message reaches source cluster's IBMC instance

The backup status can be checked at any moment by executing velero get backup -n nemo-kernel.



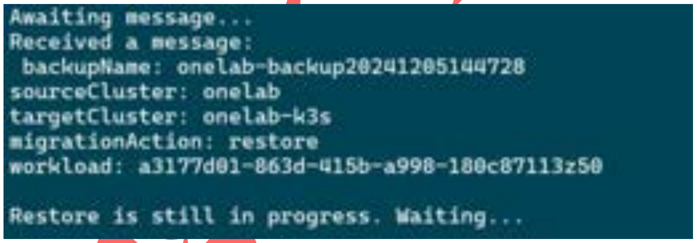
```
NAME                           STATUS      ERRORS   WARNINGS   CREATED                          EXPIRES   STORAGE LOCATION   SELECTOR
onelab-backup20241205144728    Completed   0        0          2024-12-05 16:47:28 +0100 CET    29d       default            nemo.eu/workload=a3177d01-863d-415b-a998-180c87113z50
```

Figure 127: Backup status

Once the backup is completed, a message is sent to the target cluster in order to continue with the migration process.

4.3.2.5 Step 7

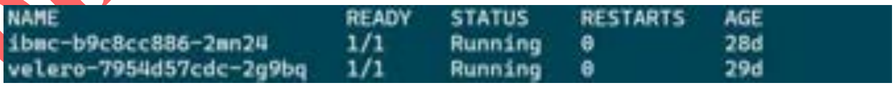
The IBMC instance running in the target cluster receives the message from the source cluster and proceeds to restore the workload resources.



```
Awaiting message...
Received a message:
 backupName: onelab-backup20241205144728
sourceCluster: onelab
targetCluster: onelab-k3s
migrationAction: restore
workload: a3177d01-863d-415b-a998-180c87113z50

Restore is still in progress. Waiting...
```

Figure 128: Restore message reaches target cluster's IBMC instance

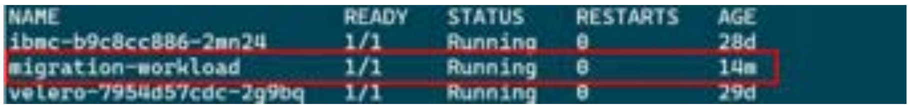
If `kubectl get pods --context onelab-k3s` is executed before the migration, it can be observed that the workload doesn't exist in the cluster:



```
NAME                       READY   STATUS    RESTARTS   AGE
ibmc-b9c8cc886-2mn24       1/1     Running   0          28d
velero-7954d57cdc-2g9bq    1/1     Running   0          29d
```

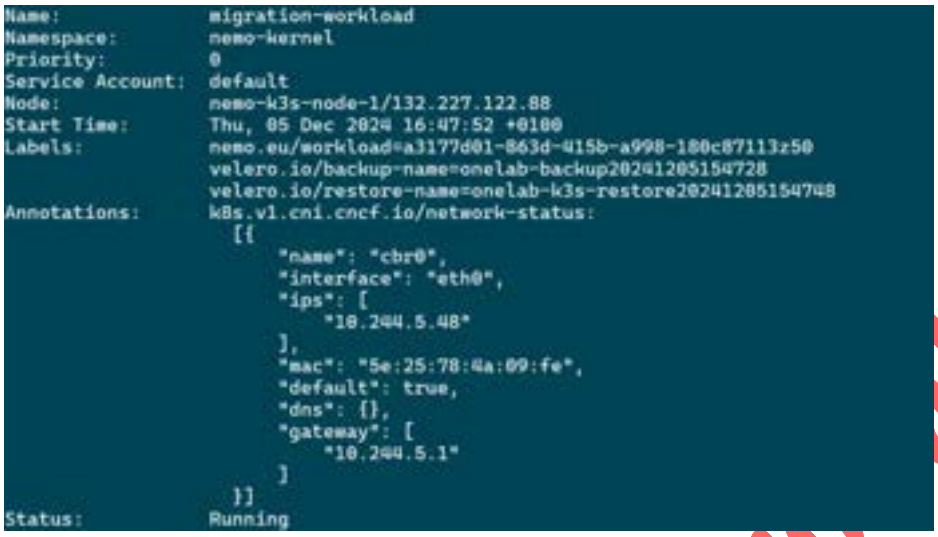
Figure 129: k3s cluster status before migration

Once the restore is completed, the workload is correctly deployed:



```
NAME                       READY   STATUS    RESTARTS   AGE
ibmc-b9c8cc886-2mn24       1/1     Running   0          28d
migration-workload         1/1     Running   0          14m
velero-7954d57cdc-2g9bq    1/1     Running   0          29d
```

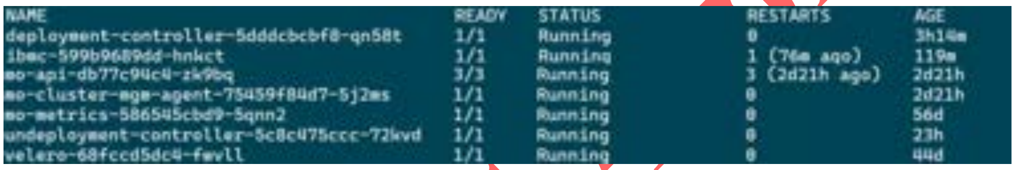
Figure 130: k3s cluster status after migration completion



```
Name:             migration-workload
Namespace:        nemo-kernel
Priority:         0
Service Account:  default
Node:             nemo-k3s-node-1/132.227.122.88
Start Time:       Thu, 05 Dec 2024 16:47:52 +0100
Labels:           nemo.eu/workload=a3177d01-863d-415b-a998-180c87113z50
                  velero.io/backup-name=onelab-backup20241205154728
                  velero.io/restore-name=onelab-k3s-restore20241205154748
Annotations:      k8s.v1.cni.cncf.io/network-status:
                    [{
                        "name": "cbr0",
                        "interface": "eth0",
                        "ips": [
                            "10.244.5.48"
                        ],
                        "mac": "5e:25:78:4a:09:fe",
                        "default": true,
                        "dns": {},
                        "gateway": [
                            "10.244.5.1"
                        ]
                    }]
Status:           Running
```

Figure 131: Description of workload in k3s cluster

The source cluster can be checked to verify that the workload has been deleted from it:



```
NAME                                       READY   STATUS    RESTARTS        AGE
deployment-controller-5dddcbcbf8-qn58t     1/1     Running   0               3h14m
ibmc-599b9689dd-hnkct                      1/1     Running   1 (76m ago)     119m
mo-api-db77c94c4-zk9bq                     3/3     Running   3 (2d21h ago)   2d21h
mo-cluster-mgm-agent-75459f84d7-5j2ms      1/1     Running   0               2d21h
mo-metrics-586545cbd9-5qnn2                1/1     Running   0               56d
undeployment-controller-5c8c475ccc-72kvd   1/1     Running   0               23h
velero-68fccd5dc4-fwvll                    1/1     Running   0               44d
```

Figure 132: OneLab cluster after migration

4.3.2.6 Step 8

When the migration is successfully completed, a message is sent to the Intent-Based API updating the workload status:

```
SourceCluster: onelab
TargetCluster: onelab-k3s
WorkloadID: a3177d01-863d-415b-a998-180c87113z50
Status: migrated
```

4.3.3 Verification summary checklist

Checklist for Test3: NEMO workload migration		Yes	No	Comments
		Yes	**No**	**Comments**
1	The CFDRL issues the workload migration request to the MO	✓		Success
2	Intent-API publishes a workload intent with an availability requirement.	✓		Success
3	MO retrieves workload status from the Intent-API.	✓		Success
4	If the workload is already deployed in any cluster, then a migration action is triggered.	✓		Success
5	Check cluster availability.	✓		Success
6	MO sends a message to the IBMC containing the workload ID and the target cluster.	✓		Success
7	IBMC backs up the workload resources and uploads them MinIO.	✓		Success
8	IBMC restores the resources in the target cluster. After the resources get resotred, the workload is removed from the source cluster.	✓		Success
9	The IBMC sends a message to the Intent-API updating the workload status, specifying the cluster where it has been deployed.	✓		Success

Table 12: Checklist for Test3 - NEMO workload migration

4.4 NEMO workload lifecycle management

NEMO incorporates the concept of intents for the declarative description of requirements for workload execution and operation within the meta-OS. Intent management processes are integrated by design in NEMO operations. Once the NEMO workload is registered by the NEMO user through the LCM UI and the NEMO workload instance specified the intents that define the required operation for the NEMO workload the monitoring process starts. The monitoring process concerns both the cluster and the NEMO workloads' dynamic resource consumption properties that adhere to the defined expectation targets. The abovementioned measurements are collected by the PPEF component. The PPEF's specific functionality is described in detail in D3.2 [11]. In addition, complementary metrics about the network and health (among others) characteristics of a workload are also collected via the CMDT component. The latter will be documented in D2.3 [4] in full detail. The collected information is visualized through the LCM UI and is available to the NEMO user (workload provider). This particular scenario concerns also the policy enforcement and notification of the NEMO user in case of a policy breach (for an intent). As indicated in the process diagram below, this information is captured by the Intent-based API (which is notified by the PPEF in advance) and is communicated to the LCM component.

4.4.1 Process diagram

The process diagram presented below summarizes the NEMO workload lifecycle management that concerns the workload monitoring and management of the asset by the NEMO user. The NEMO components that provide cluster level and workload level measurements are included in the relevant scenario. The collected information is communicated through the RabbitMQ and the Intent-based API to the LCM UI where they are visualized to the NEMO user.

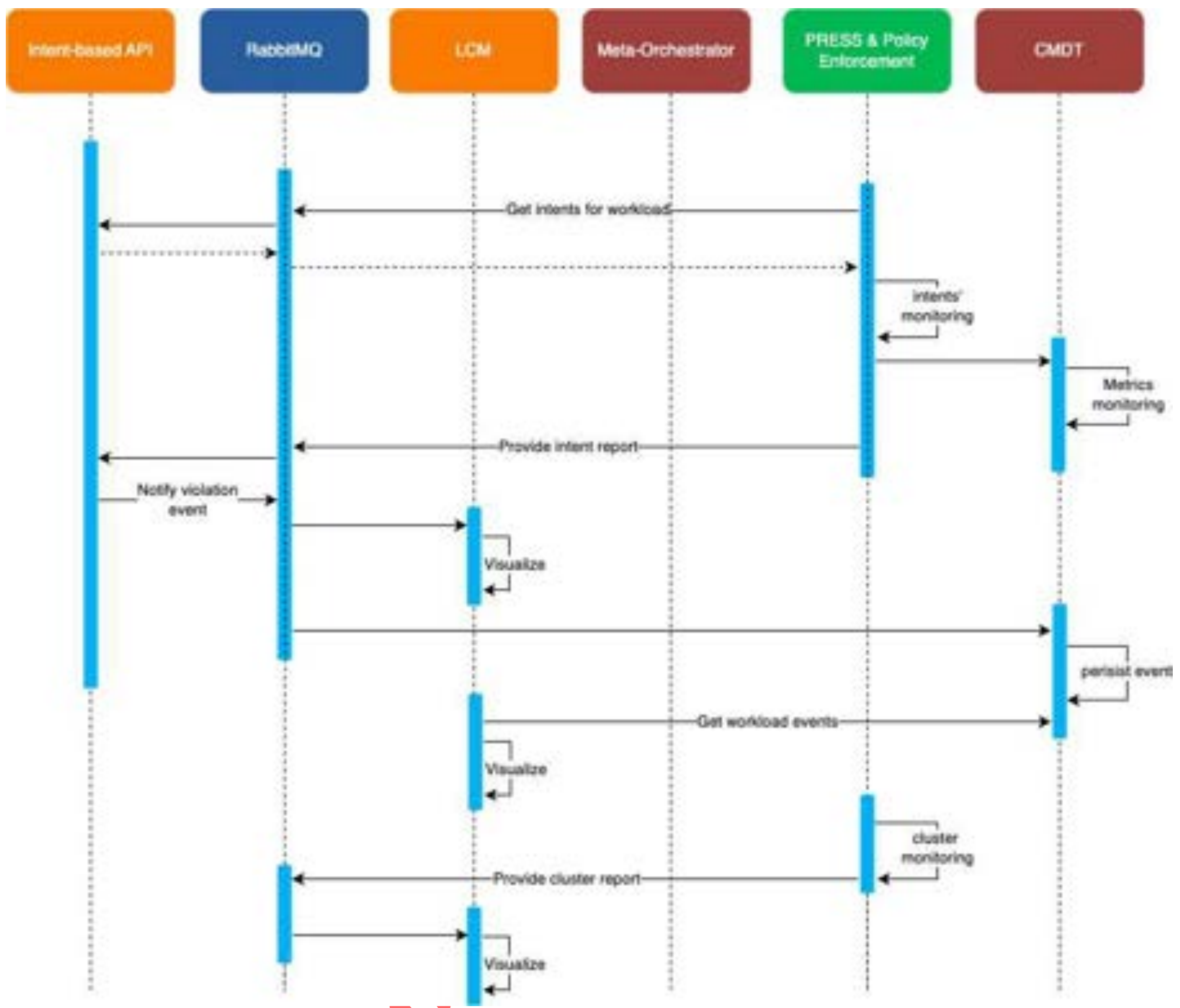


Figure 133: Process diagram for workload monitoring and enforcement

4.4.2 Verification scenario

Test 4: NEMO workload lifecycle management	
Objective	To verify the workload lifecycle management process in NEMO covering all the steps identified.
Components	• LCM • Intent-API • PPEF • CMDT • RabbitMQ
Features to be tested	This integration scenario aims to validate the workload lifecycle management. The NEMO workload intents and complementary measurements that concern the resources' consumption and the resulting performance and liveness of a workload are collected by the PPEF and the CMDT

	components. From there they are communicated through the RabbitMQ to the LCM UI where there are visualized to the NEMO user.
Test setup	The associated components are deployed in OneLab facilities at NEMO dev cluster 1 The CFDRL component which is undergoes its final stages of development.
Steps	1. The NEMO user accesses the LCM UI 2. NEMO workload monitoring collects metrics that correspond to the NEMO workload (PPEF) 3. The collected workload metrics are communicated to the Intent-API 4. The NEMO Cluster monitoring collects measurements that concern the NEMO meta-OS operated clusters (PPEF) 5. The collected cluster metrics are communicated to the RabbitMQ 6. NEMO workload complementary monitoring (CMDT) 7. The collected metrics are communicated to the Rabbit MQ 8. The LCM aggregates the collected metrics and visualize them to the NEMO user 9. The PPEF report intent violations to the Intent-based API

4.4.3 Results

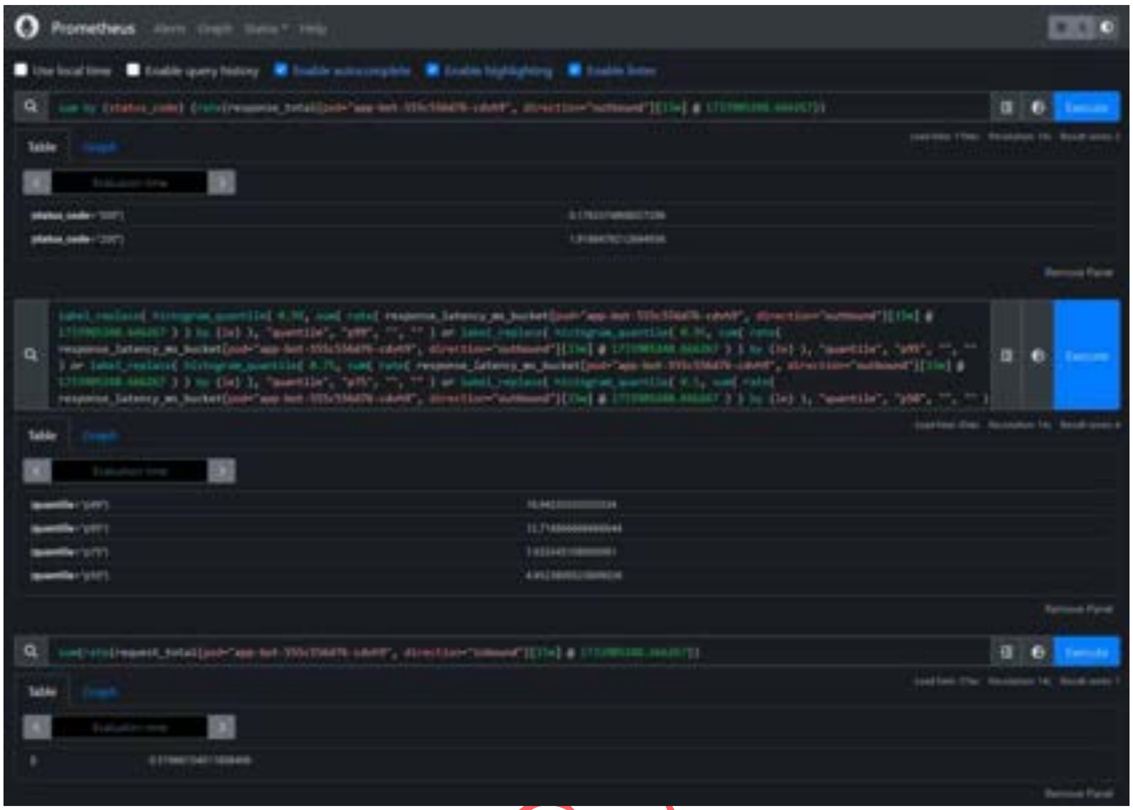
This section documents the process that is described in the scenario above step by step.

4.4.3.1 NEMO workload monitoring – CMDT

The CMDT collects network traffic characteristics and observes Kubernetes pod history. This is done through querying Kubernetes API, and Thanos/Prometheus. More detailed description of CMDT functionalities is available in D2.3 [4].

The Figure 134 illustrates how part of information is obtained through Prometheus to gain insight into network traffic characteristics, which were collected by Linkerd[24] service. The first query concerns pod's response rate per minute sorted by HTTP status code (5xx server side error, 2xx success), the second query provides the summary of maximum response latency for 99%, 95%, 75%, and 50% of connections, and the third the incoming request rate per minute.

[24] https://linkerd.io/



```
sum by (status_code) (rate(response_total{pod="app-bot-[illegible]", direction="outbound"}[1m] @ [illegible]))
```

{status_code="500"}	[illegible]
{status_code="200"}	[illegible]

```
label_replace( histogram_quantile( 0.99, sum( rate( response_latency_ms_bucket{pod="app-bot-[illegible]", direction="outbound"}[1m] @
[illegible] ) ) by (le) ), "quantile", "p99", "", "" ) or label_replace( histogram_quantile( 0.95, sum( rate(
response_latency_ms_bucket{pod="app-bot-[illegible]", direction="outbound"}[1m] @ [illegible] ) ) by (le) ), "quantile", "p95", "", ""
) or label_replace( histogram_quantile( 0.75, sum( rate( response_latency_ms_bucket{pod="app-bot-[illegible]", direction="outbound"}[1m] @
[illegible] ) ) by (le) ), "quantile", "p75", "", "" ) or label_replace( histogram_quantile( 0.5, sum( rate(
response_latency_ms_bucket{pod="app-bot-[illegible]", direction="outbound"}[1m] @ [illegible] ) ) by (le) ), "quantile", "p50", "", "" )
```

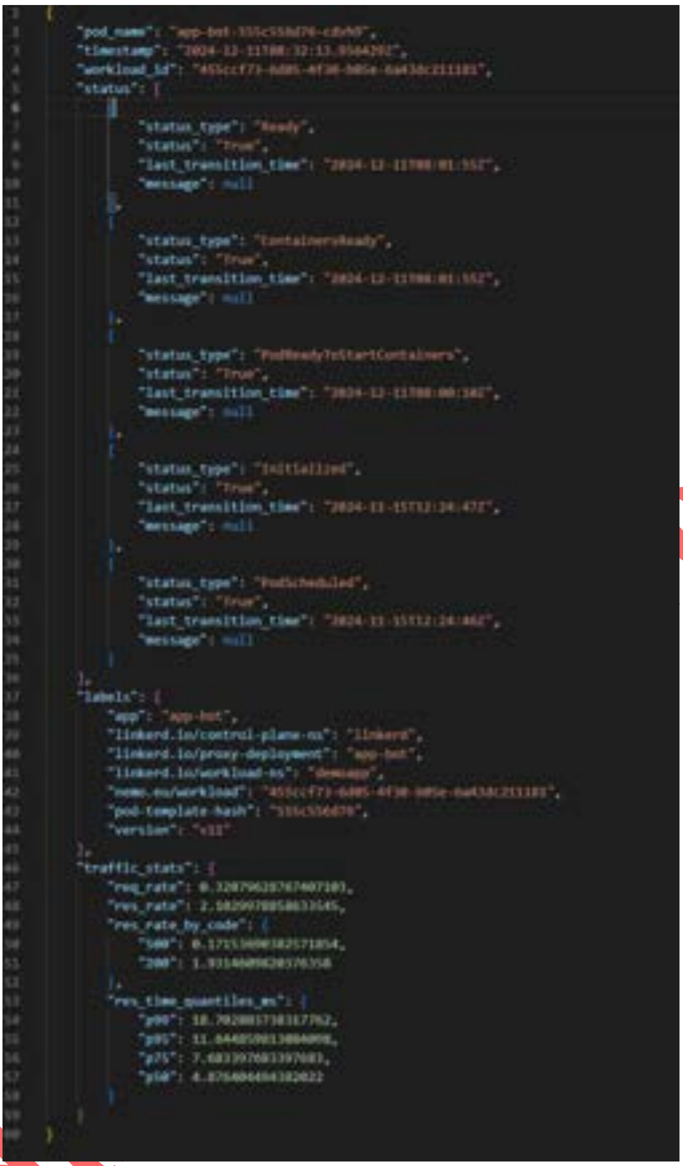
{quantile="p99"}	[illegible]
{quantile="p95"}	[illegible]
{quantile="p75"}	[illegible]
{quantile="p50"}	[illegible]

```
sum(rate(request_total{pod="app-bot-[illegible]", direction="inbound"}[1m] @ [illegible]))
```

{}	[illegible]

Figure 134: Three queries to obtain network traffic stats collected by CMDT through Linkerd

The CMDT instances then fuse data from different parts of monitoring infrastructure and produce per-pod summary message sent through RabbitMQ to other services (Figure 135) that contains pod's history, pod labels and traffic measurements i.e. request/response rate per minute, and latency.



```
{
    "pod_name": "app-bot-555c556d76-cdvh9",
    "timestamp": "2024-12-11T08:32:13.9564292",
    "workload_id": "455ccf73-6d05-4f30-b05e-6a43dc211181",
    "status": [
        {
            "status_type": "Ready",
            "status": "True",
            "last_transition_time": "2024-12-11T08:01:55Z",
            "message": null
        },
        {
            "status_type": "ContainersReady",
            "status": "True",
            "last_transition_time": "2024-12-11T08:01:55Z",
            "message": null
        },
        {
            "status_type": "PodReadyToStartContainers",
            "status": "True",
            "last_transition_time": "2024-12-11T08:00:38Z",
            "message": null
        },
        {
            "status_type": "Initialized",
            "status": "True",
            "last_transition_time": "2024-11-15T12:24:47Z",
            "message": null
        },
        {
            "status_type": "PodScheduled",
            "status": "True",
            "last_transition_time": "2024-11-15T12:24:46Z",
            "message": null
        }
    ],
    "labels": {
        "app": "app-bot",
        "linkerd.io/control-plane-ns": "linkerd",
        "linkerd.io/proxy-deployment": "app-bot",
        "linkerd.io/workload-ns": "demoapp",
        "nemo.eu/workload": "455ccf73-6d05-4f30-b05e-6a43dc211181",
        "pod-template-hash": "555c556d76",
        "version": "v11"
    },
    "traffic_stats": {
        "req_rate": 0.32879628767407103,
        "res_rate": 2.5829978858633545,
        "res_rate_by_code": {
            "500": 0.17153690382571054,
            "200": 1.9314609820576358
        },
        "res_time_quantiles_ms": {
            "p99": 18.702083738317762,
            "p95": 11.644859813084098,
            "p75": 7.683397683397683,
            "p50": 4.876404494382022
        }
    }
}
```

Figure 135: Expected RabbitMQ message data model

4.4.3.2 NEMO workload monitoring – PPEF

In the following paragraphs the workload monitoring associated results are presented.

Computing workload intent

The PPEF collects the computing workload intent measurements (CPU and RAM) by querying the deployed monitoring tool (Thanos[25]). The detailed description of the PPEF architecture and provided functionality is included in D3.2 [11]. The Figure 136 below illustrates CPU measurement collection and Figure 137 the RAM measurement collection.

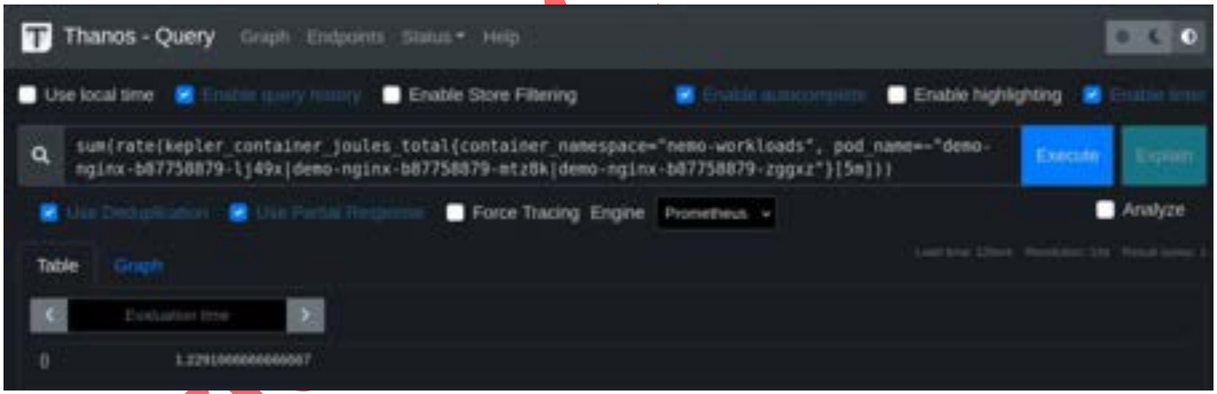
[25] https://thanos.io/

Figure 136: Workload – CPU usage

Figure 137: Workload - RAM usage

EnergyEfficiency intent

The Figure 138 below depicts the *Green Energy Consumption Rate* (the containers of the deployment "demo-nginx" are consuming approximately 1,23 joules per second averaged over the last 5 minutes).

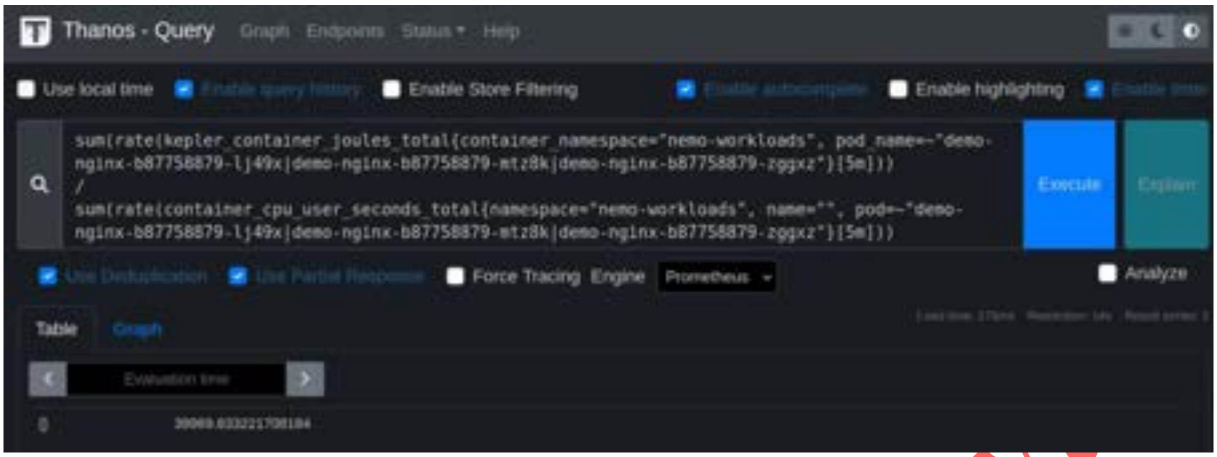


```
sum(rate(kepler_container_joules_total{container_namespace="nemo-workloads", pod_name=~"demo-nginx-b87758879-lj49x|demo-nginx-b87758879-mtz8k|demo-nginx-b87758879-zggxz"}[5m]))
```

```
{}                1.2291666666666667
```

Figure 138: Workload - Energy consumption rate

NEMO workload Energy Efficiency shows that the service consumes 40k Joules for every second of CPU time as illustrated in the collected query below (Figure 139).


```
sum(rate(kepler_container_joules_total{container_namespace="nemo-workloads", pod_name=~"demo-nginx-b87758879-lj49x|demo-nginx-b87758879-mtz8k|demo-nginx-b87758879-zggxz"}[5m]))
/
sum(rate(container_cpu_user_seconds_total{namespace="nemo-workloads", name="", pod=~"demo-nginx-b87758879-lj49x|demo-nginx-b87758879-mtz8k|demo-nginx-b87758879-zggxz"}[5m]))
```

```
{}    39069.833221708184
```

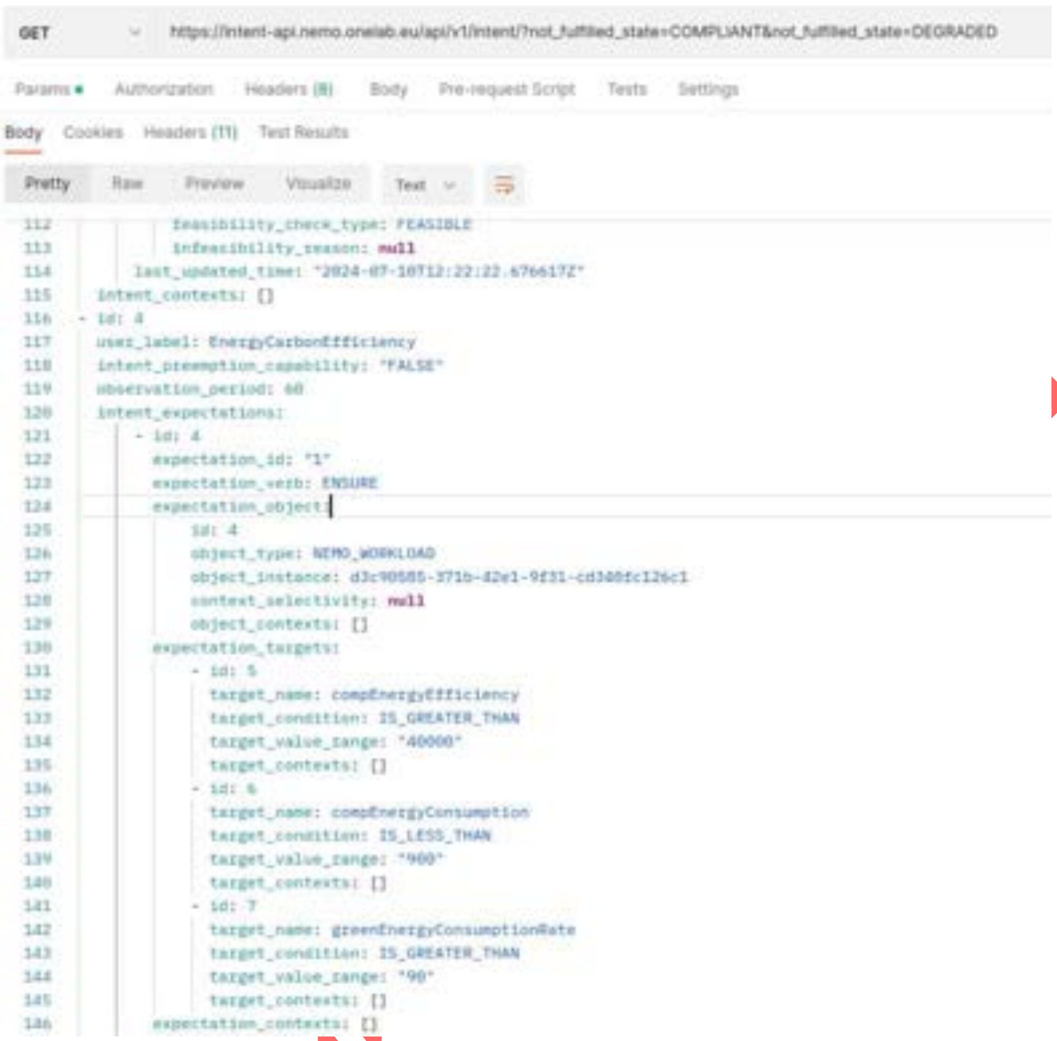
Figure 139: Workload - Energy efficiency

Energy consumption

The Energy consumption of a particular workflow is depicted in Figure 140 below.

Figure 140: Workload - Energy consumption

The Energy Efficiency intents provisioning in *Intent-based API* is presented below. Here the NEMO user can assign expectation targets for the Energy Efficiency related expectations.



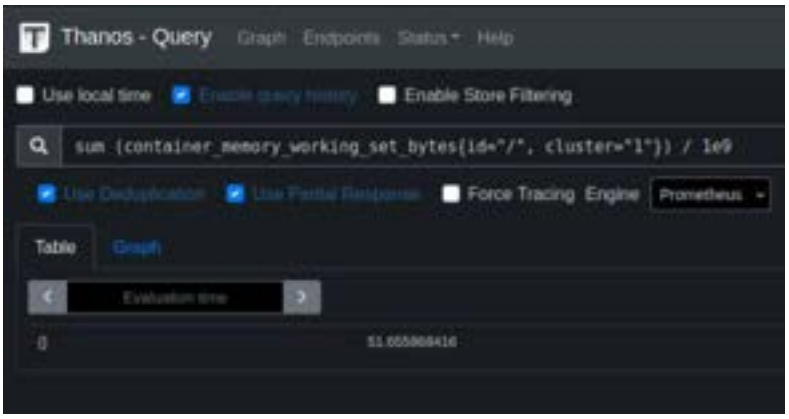
https://intent-api.nemo.onelab.eu/api/v1/intent/?not_fulfilled_state=COMPLIANT&not_fulfilled_state=DEGRADED

```
            feasibility_check_type: FEASIBLE
            infeasibility_reason: null
        last_updated_time: "2024-07-10T12:22:22.676617Z"
    intent_contexts: []
  - id: 4
    user_label: EnergyCarbonEfficiency
    intent_preemption_capability: "FALSE"
    observation_period: 60
    intent_expectations:
      - id: 4
        expectation_id: "1"
        expectation_verb: ENSURE
        expectation_object:
            id: 4
            object_type: NEMO_WORKLOAD
            object_instance: d3c90585-371b-42e1-9f31-cd340fc126c1
            context_selectivity: null
            object_contexts: []
        expectation_targets:
          - id: 5
            target_name: compEnergyEfficiency
            target_condition: IS_GREATER_THAN
            target_value_range: "40000"
            target_contexts: []
          - id: 6
            target_name: compEnergyConsumption
            target_condition: IS_LESS_THAN
            target_value_range: "900"
            target_contexts: []
          - id: 7
            target_name: greenEnergyConsumptionRate
            target_condition: IS_GREATER_THAN
            target_value_range: "90"
            target_contexts: []
        expectation_contexts: []
```

Figure 141: Intent-API EnergyEfficiency metrics update

4.4.3.3 NEMO Cluster monitoring

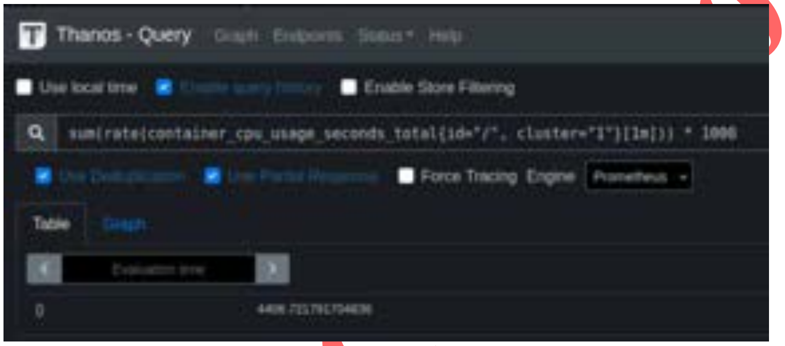
The Figure 142, Figure 143 and Figure 144 below summarize the cluster level metrics that are collected by the PPEF component for CPU, RAM and Disk storage respectively and communicated to the RabbitMQ.



```
sum (container_memory_working_set_bytes{id="/", cluster="1"}) / 1e9
```

```
{}    51.655868416
```

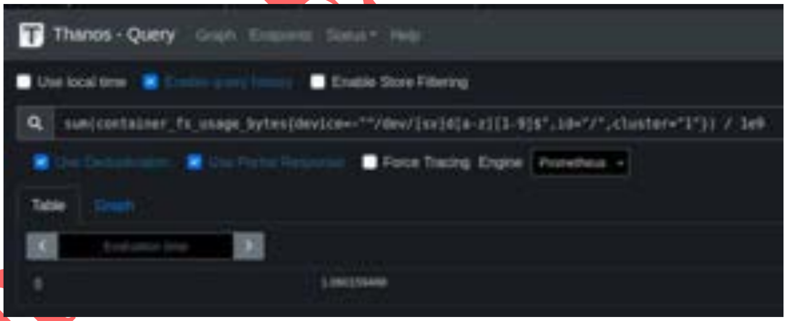
Figure 142: Cluster RAM usage



```
sum(rate(container_cpu_usage_seconds_total{id="/", cluster="1"}[1m])) * 1000
```

```
{}    4406.715791704836
```

Figure 143: Cluster CPU usage



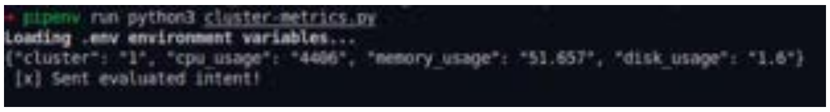
```
sum(container_fs_usage_bytes{device=~"^/dev/[sv]d[a-z][1-9]$",id="/",cluster="1"}) / 1e9
```

```
{}    1.596155648
```

Figure 144: Cluster Disk usage

4.4.3.4 Cluster metrics to RabbitMQ

Figure 145 below presents the cluster level metrics communication to the RabbitMQ from the PPEF component.



```
+ pipenv run python3 cluster-metrics.py
Loading .env environment variables...
{"cluster": "1", "cpu_usage": "4406", "memory_usage": "51.657", "disk_usage": "1.6"}
 [x] Sent evaluated intent!
```

Figure 145: cluster metrics published to RabbitMQ

4.4.4 Verification summary checklist

Checklist for Test4: NEMO workload lifecycle management		Yes	No	Comments
		Yes	**No**	**Comments**
2	NEMO workload monitoring collects metrics that correspond to the NEMO workload (PPEF)	✓		Success
3	The collected workload metrics are communicated to the Intent-API	✓		Success
4	The NEMO Cluster monitoring collects measurements that concern the NEMO meta-OS operated clusters (PPEF)	✓		Success
5	The collected cluster metrics are communicated to the RabbitMQ	✓		Success
6	NEMO workload complementary monitoring (CMDT)	✓		Success
7	The collected metrics are communicated to the Rabbit MQ	✓		Success
8	The LCM aggregates the collected metrics and visualize them to the NEMO user		✓	The LCM UI view that corresponds to this aspect is under development

Table 13: Checklist for Test4

5 Conclusions

This deliverable provided insights on the scenario-driven integration activities that produced the first integrated NEMO meta-OS. In addition, the NEMO meta-OS cloud/edge infrastructure established in OneLab facilities that supported the integration activities along with the CI/CD environment and configuration was presented in detail.

Moreover, the NEMO meta-OS components that belong into the NEMO Service Management Layer namely, the Intent-based API, the IBMC, the LCM and the MOCA were described presenting their provided functionalities, the updated architectures, the interfaces and data models and initial results.

Finally, the document provided a comprehensive description of the integration steps that were followed as part of end-to-end scenarios that reflected the technical capacity of the first integration version of the NEMO meta-OS.

The verification results will feed enhancements in the development of the NEMO meta-OS components for the next integration cycle that will produce the final version of the NEMO meta-OS platform and will be documented in D4.3 "Advanced NEMO platform & laboratory testing results. Final version".

6 References

[1] NEMO, "D4.3 - Advanced NEMO platform & laboratory testing results. Final version," HORIZON - 101070118 - NEMO Deliverable Report, 2025.

[2] NEMO, "D4.1 - Integration guidelines & initial NEMO WP4 integration," HORIZON - 101070118 - NEMO Deliverable Report, 2023.

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

[4] NEMO, "D2.3 - Enhancing NEMO Underlying Technology. Final version," HORIZON - 101070118 - NEMO Deliverable Report, 2024.

[5] Z. Anastasakis, T.-H. Velivassaki, A. Voulkidis, S. Bourou, K. Psychogyios, D. Skias and T. Zahariadis, "FREDY: Federated Resilience Enhanced with Differential Privacy," *Future Internet,* vol. 15, no. 9, 2023.

[6] N. Papernot, M. Abadi, Ú. Erlingsson, I. Goodfellow and K. Talwar, "Semi-supervised Knowledge Transfer for Deep Learning from Private Training Data," *arxiv.*

[7] S. Mwanje, A. Banerjee, J. Goerge, A. Abdelkader, G. Hannak, P. Szilágyi, T. Subramanya, J. Goser and T. Foth, "Intent-Driven Network and Service Management: Definitions, Modeling and Implementation," *ITU Journal on Future and Evolving Technologies,* vol. 3, no. 3, 2022.

[8] R. Xu, M. Scott and S. Mwanje, "Enabling intelligence and autonomation for 5G Advanced Networks," 3GPP, 2023. [Online]. Available: https://www.3gpp.org/technologies/intent.

[9] 3GPP, "3GPP TS 28.312 V18.3.0 (2024-03) - Technical Specification Group Services and System Aspects - Management and orchestration - Intent driven management services for mobile networks (Release 18)," 3GPP, 2024.

[10] NEMO, "D1.3 - NEMO meta-architecture, components and benchmarking. Final version," HORIZON - 101070118 - NEMO Deliverable Report, 2024.

[11] NEMO, "D3.2 - NEMO Kernel.Initial version," HORIZON - 101070118 - NEMO Deliverable Report, 2024.

7 Annex A – MOCA API & data models

The MOCA API swagger page (OpenApi) is online available in this location (https://intent-api.nemo.onelab.eu/moca/api/v1/swagger/)

7.1 MOCA Data models

MOCA allows for the exposure of the cluster registration process, as well as management of resource and accounting details. This section will provide the API documentation and the data models used by MOCA.

Attribute	Data type	Description
id	String	The id of the accounting event
type	String	The type of the accounting event
customer_id	String	The id of the customer (workloads, clusters)
tokens	number	The NEMO tokens that were deposited or withdrawn
balance	number	The NEMO tokens that the customer currently has left
balance_action_id	Integer	The id to retrieve the accounting event from the blockchain
timestamp	String	

Table 14: MOCA AccountingEvents Data Model

Attribute	Data type	Description
id	String	The id of the Cluster
cluster_name	String	The name of the Cluster that will be deployed
cpus	Integer	The number of CPUs of the Cluster
memory	Integer	The RAM of the Cluster in GB
storage	Integer	The disk storage of the Cluster in GB
availability	String	The percentage of time that the cluster is up (99.9%, 99%, 90%)
green_energy	String	The percentage of RES powering the cluster. (0%,20%,40%,60%,80%,100%)
cost	String	The cost type of a cluster (low cost, high performance)
cpu_base_rate	number	The CPU cost of the cluster by the CPU capacity of the cluster (in milliseconds)

memory_base_rate	number	The memory cost of the cluster by the memory capacity of the cluster (in MBs)
timestamp	String	
balance	number	The NEMO tokens of the cluster

Table 15: MOCA ClusterResources Data Model

Attribute	Data type	Description
cluster_resources	String	The id of the ClusterResources
status	Array	The status of the deployment of the Cluster
timestamp	String	

Table 16: MOCA ClusterState Data Model

Attribute	Data type	Description
id	String	The id of the workload
cluster_name	String	The name of the cluster the workload is deployed to
status	String	The status of the deployment of the Cluster
cpus	number	The number of CPUs of the Application
memory	number	The RAM of the Application in MB
storage	number	The space of the volume in GB
timestamp	String	
balance	number	The NEMO tokens of the workload
user	Integer	The id of the Workload User

Table 17: MOCA Workload Data Model

Attribute	Data type	Description
cluster	String	The id of the ClusterState
link_cid	String	The CID of the Cluster config stored in IPFS
ipfs_link	String	The link to retrieve the Cluster config
timestamp	String	

Table 18: MOCA IPFS Handler Data Model

Attribute	Data type	Description
id	String	
user_id	String	The id of the user
workload_id	String	The id of the Workload the computation took place for
cluster_id	String	The id of the cluster
cpu	String	The cpu used by the workload
ram	Array	The ram used by the workload
tokens	Array	The NEMO tokens that were charged to the workload
timestamp	String	

Table 19: MOCA WorkloadComputeTokensEvents Data Model

Attribute	Data type	Description
username	String	The user's username
balance	number	The balance of the user
smart_contracts	Array	The names of the smart contracts related to a user

Table 20: MOCA UserSmartContracts Data Model

Attribute	Data type	Description
region	String	
high_demand	Boolean	
high_demand_cost	number	
regional_cpu_limit	number	
regional_ram_limit	number	

Table 21: MOCA NemoTokenSetup Data Model

7.2 MOCA API endpoints

7.2.1 GET /api/v1/accounting_events

Returns all the accounting events related to a user (GET).

Responses			
HTTP Code	Description	Schema Type	Data Model
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

201	The accounting events of a user	Object	AccountingEvents

Table 22: GET Accounting Events responses

7.2.2 DELETE /cluster/delete/{id}

Delete a cluster based on its ID (DELETE).

Responses			
HTTP Code	Description	Schema Type	Data Model
200	The cluster has been deleted		
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 23: DELETE Cluster responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	The cluster id	True	string

Table 24: DELETE Cluster parameters

7.2.3 POST /cluster/register

Register a cluster in NEMO meta-OS (POST)

Parameters				
Attribute	Parameter Type	Description	Required	Data type
cluster_name	body	The cluster name	True	string
cpus	body	The cluster # of cpus	True	integer
memory	body	The cluster # of memory	True	float
storage	body	The cluster total storage capacity	True	float
availability	body	The cluster availability %	True	string
green_energy	body	The cluster RES powered %	True	string
cost	body	The cost category of the cluster	True	string
cpu_base	body	The cpu base cost for the cluster	True	float

memory_base_rate	body	The memory base cost for the cluster	True	float

Table 25: REGISTER cluster parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		
406	The smart contract rolled back (declined) the transaction		
201	The Cluster ID		

Table 26: POST Cluster responses

7.2.4 GET /cluster/retrieve

Retrieve all the clusters' details related to a user (GET).

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	body	The id of the cluster	False	string
cluster_name	body	The cluster name	True	string
cpus	body	The cluster # of cpus	True	integer
memory	body	The cluster # of memory	True	float
storage	body	The cluster total storage capacity	True	float
availability	body	The cluster availability %	True	string
green_energy	body	The cluster RES powered %	True	string
cost	body	The cost category of the cluster	True	string
cpu_base	body	The cpu base cost for the cluster	True	float
memory_base_rate	body	The memory base cost for the cluster	True	float
timestamp	body	Timestamp	False	string

Table 27: GET cluster parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The details of all records	Object	ClusterResources
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 28: GET Clusters responses

7.2.5 GET /cluster/retrieve/{id}

Retrieve a cluster's details related to a user (GET).

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	The cluster id	True	string

Table 29: GET Cluster with ID parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The details of selected records	Object	ClusterResources
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 30: GET Cluster with ID responses

7.2.6 PUT, PATCH /cluster/update/{id}

Update a cluster's attributes (PUT, PATCH).

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	The cluster id	True	
data	body	The cluster attributes	True	UpdateClusterResources

Table 31: PUT, PATCH Cluster parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object	UpdateClusterResources
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		
406	The smart contract rolled back (declined) the transaction		

Table 32: PUT, PATCH Cluster responses

7.2.7 POST /nemo_token_estimation_setup

Setup the region costs for that will be used in the workload usage calculation (POST).

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The generated transaction hash	Object	TransactionHash
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		
406	The smart contract rolled back (declined) the transaction		

Table 33: POST Region Costs responses

7.2.8 GET /nemo_token_setup_retrieve/{region}

Retrieve the information on the region costs (GET).

Parameters				
Attribute	Parameter Type	Description	Required	Data type
region	path	The region name	True	string

Table 34: GET Region Costs parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The region info	Object	NemoTokenSetupRetrieveRegion
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		
406	The smart contract rolled back (declined) the transaction		

Table 35: GET Region Costs responses

7.2.9 GET /nemo_user_info

Retrieve the information of a logged in user (GET).

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The information of the user	Object	NemoUserInfo
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 36: GET user information responses

7.2.10 GET /workload/retrieve

Retrieve all the details of the workloads related to a user (GET).

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The details of all records	Object	Workload
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 37: GET workoads' details responses

7.2.11 GET /workload/retrieve/{id}

Retrieve the details of a workload based on its ID (GET).

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	The workload id	True	string

Table 38: GET workoad's details parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The details of all records	Object	Workload
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 39: GET workload’s details responses

7.2.12 GET /workload_computations/{id}

Retrieve all the events of a workload that show its resource usage details (GET).

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	The workload id	True	string

Table 40: GET workload computation details parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
201	The details of all records	Object	WorkloadComputeTokensEvents
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Invalid permissions		

Table 41: GET workload computation details responses

8 Annex B – Intent-based API & data models

8.1 NEMO Intent-based API

The Intent-based API Server allows for exposure of NEMO functionalities, as well as management of intents and workloads. The API is available online at https://intent-api.nemo.onelab.eu/api/v1/swagger/.

8.2 Intent-API data models

Attribute	Data type	Description	Comments
username	String		
password	String		
token	String		

Table 42: Data model description: AuthToken

Attribute	Data type	Description	Comments
cluster_name	String	The name of the cluster resource	
cpus	Integer	The number of the CPUs	
memory	Integer	The RAM of the cluster in GB	
storage	Integer	The storage of the cluster in GB	

Table 43: Data model description: ClusterRegister

Attribute	Data type	Description
link_id	String	The IPFS link id
ipfs_link	String	The IPFS link to retrieve the cluster config

Table 44: Data model description: ClusterIpfs

Attribute	Data type	Description
id	String	The ID of the cluster
vm_name	String	The name of the cluster resource
cpus	Integer	The number of the CPUs
memory	Integer	The RAM of the cluster in GB
storage	Integer	The storage of the cluster in GB
endpoint	String	The endpoint of the Cluster
ipfs	ClusterIpfs	

Table 45: Data model description: Cluster

Attribute	Data type	Description
id	Integer	
context_attribute	String	
context_condition	String	
context_value_range	String	

Table 46: Data model description: Context

Attribute	Data type	Description
id	Integer	
object_type	String	It describes the expectation object type which can be supported by a specific intent handling function of MnS producer.
object_instance	String	
context_selectivity	String	How to select among the stated expectationContexts
object_contexts	Array	

Table 47: Data model description: ExpectationObject

Attribute	Data type	Description
id	Integer	
target_name	String	
target_condition	String	
target_value_range	String	
target_contexts	Array	

Table 48: Data model description: ExpectationTarget

Attribute	Data type	Description
id	Integer	
expectation_id	String	A unique identifier of the intentExpectation within the intent
expectation_verb	String	
expectation_object	ExpectationObject	
expectation_targets	Array	
expectation_contexts	Array	

Table 49: Data model description: IntentExpectation

Attribute	Data type	Description
id	Integer	
user_label	String	A user-friendly (and user assignable) name of the intent.
intent_preemption_capability	String	
observation_period	Integer	In seconds
intent_expectations	Array	
intent_report_reference	String	
intent_contexts	Array	

Table 50: Data model description: Intent

Attribute	Data type	Description
context_attribute	String	
context_condition	String	
context_value_range	String	

Table 51: Data model description: ContextInput

Attribute	Data type	Description
object_type	String	It describes the expectation object type which can be supported by a specific intent handling function of MnS producer.
object_instance	String	
context_selectivity	String	How to select among the stated expectationContexts
object_contexts	Array	

Table 52: Data model description: ExpectationObjectInput

Attribute	Data type	Description
target_name	String	
target_condition	String	
target_value_range	String	
target_contexts	Array	

Table 53: Data model description: ExpectationTargetInput

Attribute	Data type	Description
expectation_id	String	A unique identifier of the intentExpectation within the intent
expectation_verb	String	

expectation_object	ExpectationObjectInput	
expectation_targets	Array	
expectation_contexts	Array	

Table 54: Data model description: IntentExpectationInput

Attribute	Data type	Description
user_label	String	A user-friendly (and user assignable) name of the intent.
context_selectivity	String	How to select among the stated intentContexts
intent_preemption_capability	String	
observation_period	Integer	In seconds
intent_expectations	Array	
intent_contexts	Array	

Table 55: Data model description: IntentInput

Attribute	Data type	Description
intent	IntentInput	

Table 56: Data model description: IntentInputAttribute

Attribute	Data type	Description
intent	Integer	Intent ID (PK)

Table 57: Data model description: IntentOutput

Attribute	Data type	Description
target_name	String	
target_condition	String	
target_value_range	String	

Table 58: Data model description: TargetTemplate

Attribute	Data type	Description	Comments
instance_id	String	NEMO Workload instance ID	'uuid'
intent_type	String	Intent userLabels	
service_start_time	String	Optional	'date-time'
service_end_time	String	Optional	'date-time'
targets	Array	Expectation targets	
instance_id	String	NEMO Workload instance ID	'uuid'

Table 59: Data model description: IntentTemplate

Attribute	Data type	Description
action	String	Action to perform

Table 60: Data model description: IntentActionInput

Attribute	Data type	Description
target_id	Integer	
target_value_range	String	

Table 61: Data model description: IntentTargetUpdate

Attribute	Data type	Description	Comments
id	Integer		
username	String	Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.	
email	String		'email'
first_name	String		
last_name	String		

Table 62: Data model description: User

Attribute	Data type	Description	Comments
id	Integer		
name	String	The maintainers name (required)	
email	String	The maintainers email (optional)	'email'
url	String	A url for the maintainer (optional)	'uri'
chart	Integer		

Table 63: Data model description: WorkloadDocumentChartMaintainer

Attribute	Data type	Description
id	Integer	
name	String	The name of the chart
version	String	A SemVer 2 version string
repository	String	The repository URL or alias ("repo-name") (optional)
condition	String	A yaml path that resolves to a boolean, used for enabling/disabling charts (e.g. subchart1.enabled) (optional)

tags	Array	Tags can be used to group charts for enabling/disabling together
import_values	Array	ImportValues holds the mapping of source values to parent key to be imported.
alias	String	Alias to be used for the chart. Useful when you have to add the same chart multiple times
chart	Integer	

Table 64: Data model description: WorkloadDocumentChartDependency

Attribute	Data type	Description
id	Integer	
container_registries	Object	The mapped container registries
manifests	Array	The default generated manifests
values	Object	The chart default values
resource_mappings	Object	The container resource mappings
memory_requests	Integer	Total memory requests (in bytes)
memory_limits	Integer	Total memory limits (in bytes)
cpu_requests	Integer	Total CPU requests (in milli cpus
cpu_limits	Integer	Total CPU limits (in milli cpus)

Table 65: Data model description: WorkloadDocumentChartMetadata

Attribute	Data type	Description	Comments
id	Integer		
maintainers	Array		
dependencies	Array		
metadata	WorkloadDocumentChartMetadata		
api_version	String	The chart API version (required)	
name	String	The name of the chart (required)	
version	String	A SemVer 2 version string	
kube_version	String	A SemVer range of compatible Kubernetes versions (optional)	
description	String	A single-sentence description of this project (optional)	
type	String	The type of the chart (optional)	
keywords	Array	A list of keywords about this project (optional)	
home	String	The URL of this projects home page (optional)	'uri'

sources	Array	A list of URLs to source code for this project (optional)	
icon	String	A URL to an SVG or PNG image to be used as an icon (optional)	'uri'
app_version	String	The version of the app that this contains (optional). Needn't be SemVer. Quotes recommended.	
deprecated	Boolean	Whether this chart is deprecated (optional, boolean)	
annotations	Object	A list of annotations keyed by name (optional)	

Table 66: Data model description: WorkloadDocumentChart

Attribute	Data type	Description	Comments
id	Integer		
user	User	The NEMO user	
chart	WorkloadDocumentChart	The associated helm chart	
created	String		'date-time'
modified	String		'date-time'
name	String	The workload document name	
version	String	A SemVer 2 version string	
schema	Object	The document schema	
intents	Array	List of supported intents	
type	String	The workload document type	
status	String	The workload document status	
ingress_support	Boolean	Whether the workload document can be exposed via NEMO	
enabled	Boolean	If the workload document is enabled	
rejection_reason	String	Rejection reason	

Table 67: Data model description: WorkloadDocumentList

Attribute	Data type	Description
id	Integer	
status	String	The workload document status
user	Integer	The NEMO user
name	String	The workload document name
version	String	A SemVer 2 version string

schema	Object	The document schema
type	String	The workload document type
intents	Array	List of supported intents
ingress_support	Boolean	Whether the workload document can be exposed via NEMO

Table 68: Data model description: WorkloadDocumentCreate

Attribute	Data type	Description	Comments
id	Integer		
type	String	Lifecycle event type	
deployment_cluster	String	The NEMO deployment cluster	
migration_from_cluster	String	The NEMO migration from cluster	
migration_to_cluster	String	The NEMO migration to cluster	
timestamp	String	The timestamp of the lifecycle event	'date-time'

Table 69: Data model description: WorkloadDocumentLifecycleEvent

Attribute	Data type	Description	Comments
id	Integer		
lifecycle_events	Array		
created	String		'date-time'
modified	String		'date-time'
instance_id	String	NEMO unique workload document instance identifier	'uuid'
release_name	String	The workload document instance release name	
status	String	The workload document instance status	
lifecycle_metadata	Array	The lifecycle metadata associated with the instance	
cluster_name	String	The NEMO cluster name that the instance resides in	
ingress_enabled	Boolean	Whether the instance should be exposed via NEMO	
ingress_metadata	Object	Ingress metadata	
workload_document	Integer	The workload document	

Table 70: Data model description: WorkloadDocumentInstance

Attribute	Data type	Description
name	String	The workload document name
version	String	A SemVer 2 version string
schema	Object	The document schema
type	String	The workload document type
status	String	The workload document status
user	Integer	The NEMO user

intents	Array	List of supported intents
ingress_support	Boolean	Whether the workload document can be exposed via NEMO

Table 71: Data model description: WorkloadDocumentUpdate

Attribute	Data type	Description
release_name	String	The release name
values_override	Object	values.yaml override
include_crds	Boolean	Include CRDS
is_upgrade	Boolean	If its upgrade
namespace	String	Namespace to associate with
no_hooks	Boolean	No hooks flag
ingress_enabled	Boolean	Expose workload instance via NEMO
cluster_name	String	Target cluster override

Table 72: Data model description: WorkloadDocumentTemplateInput

8.3 Intent-based API endpoints

8.3.1 POST /api/v1/auth/login/

Create a new auth token for the user (POST)

Parameters				
Attribute	Parameter Type	Description	Required	Data type
data	body		True	AuthToken

Table 73: POST authorization token parameters

8.3.2 POST /api/v1/auth/logout/

Clears the token associated with the user. (POST)

Responses			
HTTP Code	Description	Schema Type	Data Model
201		Object	AuthToken

Table 74: POST authorization logout responses

8.3.3 POST /api/v1/cluster/register/

This endpoint writes a message to the rabbitmq topic that MOCA component listens to. This is performed in asynchronous manner. (POST)

Responses			
HTTP Code	Description	Schema Type	Data Model
204	No Content		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 75: POST cluster registration responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
data	body		True	ClusterRegister

Table 76: POST cluster registration parameters

8.3.4 GET /api/v1/cluster/retrieve/

Retrieve cluster details from MOCA component (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model
201	Created		
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 77: GET cluster retrieve responses

8.3.5 GET /api/v1/cluster/retrieve/{id}/

Retrieve a single cluster details from MOCA component (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object list	Cluster
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 78: GET cluster retrieve (id) responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path		True	

Table 79: GET cluster retrieve (id) parameter

8.3.6 GET, POST /api/v1/intent/

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object list	Cluster
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		

404	Resource not found		

Table 80: GET/POST intent responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
user_label	query	user_label	False	String
object_instance	query	object_instance	False	String
fulfilment_status	query	fulfilment_status	False	String
not_fulfilled_state	query	not_fulfilled_state	False	String
intent_id	query	intent_id	False	String
data	body		True	IntentInputAttribute

Table 81: GET/POST intent parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object list	Intent

Table 82: GET/POST intent responses

8.3.7 POST /api/v1/intent/template/

Creates an Intent with the given template (POST)

Responses			
HTTP Code	Description	Schema Type	Data Model
201		Object	IntentOutput
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 83: POST create intent responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
data	body		True	IntentTemplate

Table 84: POST create intent parameters

8.3.8 GET /api/v1/intent/types/

Lists the valid intent types (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model
201		Object	IntentOutput

400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 85: GET intent types responses

8.3.9 PUT /api/v1/intent/{id}/action/

Perform an action to a given Intent (PUT)

Responses			
HTTP Code	Description	Schema Type	Data Model
200			

Table 86: PUT intent action responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path		True	
data	body		True	IntentActionInput

Table 87: PUT intent action request

8.3.10 PUT /api/v1/intent/{id}/target/

Intent has to be in a valid state. It is best to use this in a rest api tool, e.g. postman and send data via ``application/yaml`` in order to derive data types better. (PUT)

Responses			
HTTP Code	Description	Schema Type	Data Model
200	Ok		
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 88: PUT intent's target (id) action responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path		True	
data	body		True	IntentTargetUpdate

Table 89: PUT intent's target (id) action parameters

8.3.11 GET, POST /api/v1/workload/

List or Create a new workload document(s) (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model

200	Ok		
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 90: GET workload documents list responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
name	query	name	False	String
version	query	version	False	String
data	body		True	WorkloadDocumentCreate

Table 91: GET workload documents list parameters

8.3.12 List or Create a new workload document(s) (POST)

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object list	WorkloadDocumentList

Table 92: POST workload document responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
name	query	name	False	String
version	query	version	False	String
data	body		True	WorkloadDocumentCreate

Table 93: POST workload document parameters

8.3.13 GET /api/v1/workload/instance/

Lists all the workload documents instances (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model
201		Object	WorkloadDocumentCreate

Table 94: GET workload instances responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
release_name	query	release_name	False	String
cluster_name	query	cluster_name	False	String
workload_document	query	workload_document	False	String
status	query	status	False	String

Table 95: GET workload instances parameters

8.3.14 PUT /api/v1/workload/instance/{instance_id}/delete/

Propagate a workload document instance deletion request for a deployed workload instance to the MO. (PUT)

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object list	WorkloadDocumentInstance

Table 96: PUT workload instance delete responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
instance_id	path		True	

Table 97: PUT workload instance delete parameters

8.3.15 GET /api/v1/workload/instance/{instance_id}/manifests/

Fetch workload instance manifests in a single ``.yaml`` format. (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model
200			

Table 98: GET workload instance manifests responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
instance_id	path		True	
instance_id	path	The Workload Document instance uuid	True	String

Table 99: GET workload instance manifests parameters

8.3.16 POST /api/v1/workload/upload/

The following must apply:

- The helm chart must be packed as ```.tgz``` (by running helm package).
- The helm chart must have a matching (name, version) pair with the associated workload document. - The helm chart must have a valid structure, files ```Chart.yaml```, ```values.yaml``` and folder ```templates``` are mandatory.
- The helm chart must be able to render (via helm template) without any errors.
- The helm chart underlying containers images must exist and be reachable by NEMO Intent API (either public or private registries with appropriate imagePullSecrets).

If everything is OK, the helm chart is uploaded to the NEMO S3 Helm Repository. After successful upload, the Workload Document is set to ```status=onboarding``` for further validation. After successful validation, the Workload Document is set to ```status=accepted``` or ```status=rejected``` if validation has failed. RabbitMQ is notified as per README.md (POST)

Responses			
HTTP Code	Description	Schema Type	Data Model
200	Kubernetes Manifests	Object list	
400	Provided data is invalid or malformed		

401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 100: POST workload upload request responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
file	formData	Packaged helm chart in ```.tgz``` extension	True	file
name	formData	Workload Name to associate with	True	String
version	formData	Workload Version to associate with	True	String

Table 101: POST workload upload request parameters

8.3.16.1 GET, PUT, PATCH, DELETE /api/v1/workload/{id}/

Update & Delete operations are only allowed when a workload document is in ```status=pending``` and the same user if performing the operation. (GET)

Responses			
HTTP Code	Description	Schema Type	Data Model
201	Created		
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 102: GET workload responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	A unique integer value identifying this Workload Document.	True	
data	body		True	WorkloadDocumentUpdate
data	body		True	WorkloadDocumentUpdate

Table 103: GET workload parameters

Update & Delete operations are only allowed when a workload document is in ```status=pending``` and the same user if performing the operation. (PUT)

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object	WorkloadDocumentList

Table 104: PUT workload responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type

id	path	A unique integer value identifying this Workload Document.	True	
data	body		True	WorkloadDocumentUpdate
data	body		True	WorkloadDocumentUpdate

Table 105: PUT workload parameters

Update & Delete methods are only allowed when a workload document is in ```status=pending``` and the same user if performing the operation. (PATCH)

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object	WorkloadDocumentUpdate

Table 106: PATCH workload responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	A unique integer value identifying this Workload Document.	True	
data	body		True	WorkloadDocumentUpdate
data	body		True	WorkloadDocumentUpdate

Table 107: PATCH workload parameters

Update & Delete operations are only allowed when a workload document is in ```status=pending``` and the same user if performing the operation. (DELETE)

Responses			
HTTP Code	Description	Schema Type	Data Model
200		Object	WorkloadDocumentUpdate

Table 108: DELETE workload responses

Parameters				
Attribute	Parameter Type	Description	Required	Data type
id	path	A unique integer value identifying this Workload Document.	True	
data	body		True	WorkloadDocumentUpdate
data	body		True	WorkloadDocumentUpdate

Table 109: DELETE workload parameters

8.3.16.2 POST /api/v1/workload/{id}/template/

Set header ```Accept``` to ```application/json``` or ```application/yaml``` (default). This action creates a workload document instance with a unique NEMO workload identifier (```instance_id```). RabbitMQ is notified as per README.md (POST)

Parameters

Attribute	Parameter Type	Description	Required	Data type
id	path		True	
data	body		True	WorkloadDocumentTemplateInput

Table 110: POST workload document instance parameters

Responses			
HTTP Code	Description	Schema Type	Data Model
201	Created		
400	Provided data is invalid or malformed		
401	Invalid credentials		
403	Forbidden from performing this action		
404	Resource not found		

Table 111: POST workload document instance responses