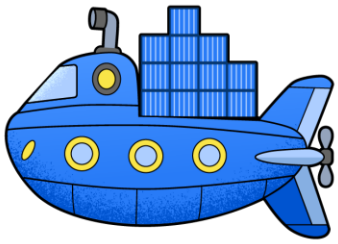


NEMO

Next Generation Meta Operating System

D4.3 Advanced NEMO platform & laboratory testing results. Final version

Document Identification			
Status	Final	Due Date	30/05/2025
Version	1.0	Submission Date	17/06/2025

Related WP	WP4	Document Reference	D4.3
Related Deliverable(s)	D1.1, D1.2, D1.3, D2.3, D3.3, D4.1, D4.2	Dissemination Level (*)	PU
Lead Participant	INTRA	Lead Author	Dimitrios Skias
Contributors	SYN, INTRA, AEGIS, SPACE, ATOS, MAG, ENG, ESOFT, SU, COMS	Reviewers	Enric Pages-Montanera (ATOS) Jonathan Klimt (RWTH)

Keywords:
Integration, validation, API, SDK, Lifecycle Management, Migration Controller, automation

Document Information

List of Contributors	
Name	**Partner**
Rubén Ramiro	ATOS
Enric Pages-Montanera	ATOS
Antonis Gonos	ESOFT
Dimitrios Skias	INTRA
Kostas Vrioni	MAG
Astik Samal	MAG
Nikos Drosos	SPACE
Emmanouil Bakiris	SPACE
Theodore Zahariadis	SYN
Ilias Seitanidis	SYN
Victor Gabillon	TSG
Spyros Vantolas	AEGIS
Hassane Rahich	SU
Matija Cankar	COMS

Document History			
Version	**Date**	**Change editors**	**Changes**
0.1	19/02/2025	Dimitrios Skias (INTRA)	ToC 1st version
0.2	20/04/2025	Dimitrios Skias (INTRA)	Final ToC
0.3	13/05/2025	INTRA, SYN, ATOS, UC3M, TID, TSG, AEGIS, SU, MAG, COMS	Initial contributions
0.4	20/05/2025	Dimitrios Skias (INTRA)	First round of contributions
0.5	23/5/2025	TSG, MAG, TID, SYN	Second round of contributions
0.6	30/5/2025	INTRA, ATOS, UC3M, SYN, COMS	Final round of Contributions
0.7	2/6/2025	RWTH, ATOS	Peer-Review ready version
0.8	6/6/2025	INTRA, MAG, TID, SYN, ATOS	Reviewers' comments consolidation
0.9	13/6/2025	INTRA, SYN	Document finalization
1.0	17/06/2025	ATOS	Quality Control & Submission

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Dimitrios Skias (INTRA)	17/06/2025
Quality manager	R. Valle Soriano (ATOS)	17/06/2025
Project Coordinator	E. Pages (ATOS)	17/06/2025

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
CIDR	Classless Inter-Domain Routing
CLI	Command Line Interface
CMDT	Cybersecure Microservices' Digital Twin
CI	Continuous Integration
CLI	Command-line Interface
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRD	Custom Resource Definition
DApps	Distributed Applications
DLT	Distributed Ledger Technology
DNS	Domain Name System
Dx.y	Deliverable number y belonging to WP x
E2E	End-to-End
EC	European Commission
EV	Electric Vehicles
FL	Federated Learning
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
gRPC	Generic Remote Procedure Call
GUI	Graphical User Interface
IBA	Intent-Based API
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
LL	Living Lab
meta-OS	Meta-Operating System
ML	Machine Learning

Abbreviation / acronym	Description
MLaaS	ML-as-a-Service
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
OSS	One-stop-shop
PAYG	Pay-As-You-Go
PV	Photovoltaic
PPEF	PRESS & Policy Enforcement Framework
RaaS	Resource-as-a-Service
REST	Representational State Transfer
SDN	Software Defined Network
SEE	Secure Execution Environment
SME	Small-Medium Enterprise
SMP	Slice Manager Plugin
SOM	System-on-Module
UI	User Interface
UPS	Uninterruptible Power Supply
UX	User Experience
YAML	yet another markup language

Executive Summary

The present deliverable, D4.3, entitled "Advanced NEMO platform & laboratory testing results-Final version" aims to provide a complete and detailed overview of the integration process carried out within the framework of the Next Generation Meta Operating System project towards its realization. Moreover, this document sheds light on the final technical specifications of the components that were developed in the context of the Work Package 4, namely the Intent-based API, MOCA, LCM and the IBMC.

Within the scope of this document several other aspects that enhance the understanding of the integration and validation process followed in NEMO are discussed. To this end, the infrastructure related aspects, such as the hardware specifications of OneLab facilities that host the NEMO meta-OS framework are presented. The ZeroOps configuration and deployment procedure followed, as described, can be utilized as a paradigm for future deployments of the NEMO meta-OS in 3rd party infrastructures. In addition, comprehensive documentation tailored for 3rd parties that aim to utilize NEMO meta-OS is included, presenting the steps that are necessary to access, register and deploy workloads in the NEMO meta-OS.

The present document intents to provide insights on the final integrated version on the NEMO meta-OS which is the result of a Validation and Verification methodology that is established and adopted. The associated work concerns also the scenario-driven integration tests that were conducted in laboratory setting including the supporting CI/CD environment and tools.

Finally, the document is foreseen to be used as the best practices handbook for other projects in terms of integration and validation procedures. NEMO since its beginning provided state-of-the-art architectural patterns that made the process of integration and validation easier, in terms of interconnecting components and debugging issues in the End-to-End system workflows.

1 Introduction

1.1 Purpose of the document

Deliverable D4.3 “Advanced NEMO platform & laboratory testing results-Final version” is the second and final version of the NEMO platform-laboratory results reporting. This document provides the updates and optimizations made since the delivery of deliverable D4.2 “Advanced NEMO platform & laboratory testing results-Initial version” [1] that aimed to provide the main guidelines for the testing and integrating steps to be followed towards NEMO’s validation. Moreover, the updated workflows used for these components’ interactions are reported in the following sections. As D4.3 is considered the final version of the integration deliverables of NEMO, an overview of the workflows used by the components of other WPs are presented towards an End-to-End workflow presentation.

1.2 Relation to other project work

Deliverable D4.3 is tightly connected to D4.2, as it is the second version of it and reports the updates of the work initially described in D4.2. In addition, D4.3 reports the issues that the component owners met during the finalization of their components. D4.3 is also strongly coupled with D4.1 [2] on which the integration methodology to be followed by the project components was defined as well as some preliminary integration activities and testing results that took place among the more mature NEMO components. In D4.3, the final version of the NEMO meta-OS was validated over as well as the updated integration plan. At Work Package (WP) level, D4.3 is firmly attached with the WP4 tasks. Tasks T4.1-T4.3 contribute to this deliverable by providing the updates carried out within the individual tools implemented by each task. Furthermore, task T4.4, responsible for the integration of the NEMO components vastly committed to the design of the individual integration pipelines as well as monitoring the procedure of integrating submodules towards the final integration.

Moreover, this deliverable has strong relation with other WPs and the technical work conducted. The components’ specifications and meta-architecture as well as the functional and non-functional requirements defined within WP1 [3] and reported on the related deliverables are validated in the current deliverable and any change of them is reported. The benchmarking definition provided by task T1.3 is demonstrated through the integration procedures reported in the following sections. While WP4 contributed to WP2 and WP3 in terms of providing the best practices related to the integration procedure, the integration steps and semantics of these two WPs are being reported in this deliverable.

1.3 Relation between D4.3 and D4.2

In D4.2, the initial outcomes as well as the guidelines to be followed in the WP4 components integration were presented, in D4.3 the final updates and optimizations carried out by the components’ owners are described. In addition, the final workflows of the WP4 and other WPs are presented to provide a complete picture of the End-to-End workflow of NEMO. While the two deliverables share some common chapters, they have several differences, for the ease of the reader these changes are described in Table 1. The structure of D4.3 has been changed compared to D4.2 to provide a better understanding of the NEMO integration process and the updates of the WP4 tools to the reader. New subsections were added to incorporate the integration activities carried out by the Open Call projects as well as some of the subsections of D4.2 were converted to standalone sections in D4.3 to provide more details about the progress of the Project and the integration pipelines.

Table 1. Differences between D4.3 and D4.2

Section in D4.3	Section in D4.2	Differences
1. Introduction	1. Introduction	Several updates in this section
1.1 Purpose of the document	1.1 Purpose of the document	Minor update
1.2 Relation to other project work	1.2 Relation to other project work	The section has been updated
1.3 Relation between D4.3 and D4.2	-	New section in D4.3
1.4 Structure of the document	1.4 Structure of the document	The section has been updated
2. NEMO Integration, Validation & Verification approach and tools	2. Methodology	Several updates in this section
2.1 NEMO CI/CD Environment & Tools	2.1 NEMO CI/CD Environment & Tools	Several updates in this Section.
2.2 3rd Party documentation of NEMO meta-OS	-	New Section in D4.3
2.3 Cloud/Edge/IoT Integration and Validation Infrastructure	2.2 Cloud/Edge/IoT Integration and Validation Infrastructure	The section has been updated
2.4 Integration & V&V Methodology & Plan	2.3 Integration & V&V Methodology & Plan	Minor Updates
3 NEMO Integrated Platform (Final Version)	2.5 NEMO Integrated Platform	New Section in D4.3, Updated
3.1 Meta-OS functionality in NEMO v2	2.5.1 Meta-OS functionality in NEMO v1	The section has been updated
3.2 NEMO Open Call integration activities	-	New Section in D4.3
4. NEMO Service Management Layer	3. NEMO Service Management Layer	Major Updates
4.1 Intent-based Migration Controller	3.1 Intent-based Migration Controller	Updated
4.2 Plugin & Applications Lifecycle Manager	3.2 Plugin & Applications Lifecycle Manager	Updated
4.3 Monetization and Consensus-based Accountability	3.3 Monetization and Consensus-based Accountability	Updated
4.4 Intent-based SDK/API	3.4 Intent-based SDK/API	Updated
5. NEMO scenario-driven verification & results	4. NEMO scenario-driven verification & results	Major Updates
5.1 NEMO Cluster registration – resource provisioning	4.1 NEMO Cluster registration	Major Updates
5.2 NEMO workload registration & provisioning	4.2 NEMO workload registration, deployment & provisioning	Major Updates
5.3 NEMO workload scheduling & orchestration	4.3 NEMO workload migration	Major Updates
5.4 NEMO workload lifecycle management	4.4 NEMO workload lifecycle management	Major Updates
5.5 NEMO WP3 Integration Activities	-	New section
6. Conclusions	5. Conclusions	The section has been updated

1.4 Structure of the document

The remainder of this report is organized as follows:

- **Section 2** contains 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** presents the high-level NEMO Integrated Platform along with the integration activities of the Open Call Projects with NEMO.
- **Section 4** 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 5** 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 6** provides conclusions and insights of the final version of the NEMO meta-OS.

2 NEMO Integration, Validation & Verification approach and tools

2.1 NEMO CI/CD Environment & Tools

2.1.1 Open-Source repository

NEMO is making use of a GitLab CI/CD framework [4] deployed at the Eclipse Research Labs to house and keep track of the code stack developed within the project. Moreover, within NEMO a docker registry repository is being used for storing and deploying the generated containers. The pipeline adopted by NEMO is integrated with the GitLab repository and utilizes an automated build functionality that generates the docker container images [5] based on the architectural tool chains of the project. The final step of the pipeline involves the automated deployment of the latest image generated to the appropriate server. The use of a centralized versioning system such as GitLab provides several benefits:

- Automates repetitive tasks like running tests and building code.
- Integrates code frequently, allowing teams to catch issues sooner.
- Run automated tests and static analysis tools with every commit.
- Automatically deploy to staging or production environments.
- Track the state of builds, tests, and deployments in real time.
- Keep a record of who deployed what, when, and why.
- Link code changes directly to issues or feature requests.
- Allow for code review and discussion before merging.
- Detect vulnerabilities in dependencies or container images.

On the other hand, apart from the technical benefits of using a version control repository, there are other benefits as well. NEMO since its conception aimed to provide an opensource meta-OS framework that will enable local organizations to deploy and use a load management system in the IoT-Edge-Cloud continuum. For this reason, the NEMO's repository[1] is public and accessible along with the Open Call projects[2,3] that aim at validating and enhancing NEMO. The repository follows a sub-project organization, Figure 1, where the components are grouped based on the implemented outcomes of the relevant tasks, while two major subprojects, Open Call 1 and Open Call 2, house the services enhancing and validating NEMO.

[1] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project
[2] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/opencall-1
[3] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/opencall-2

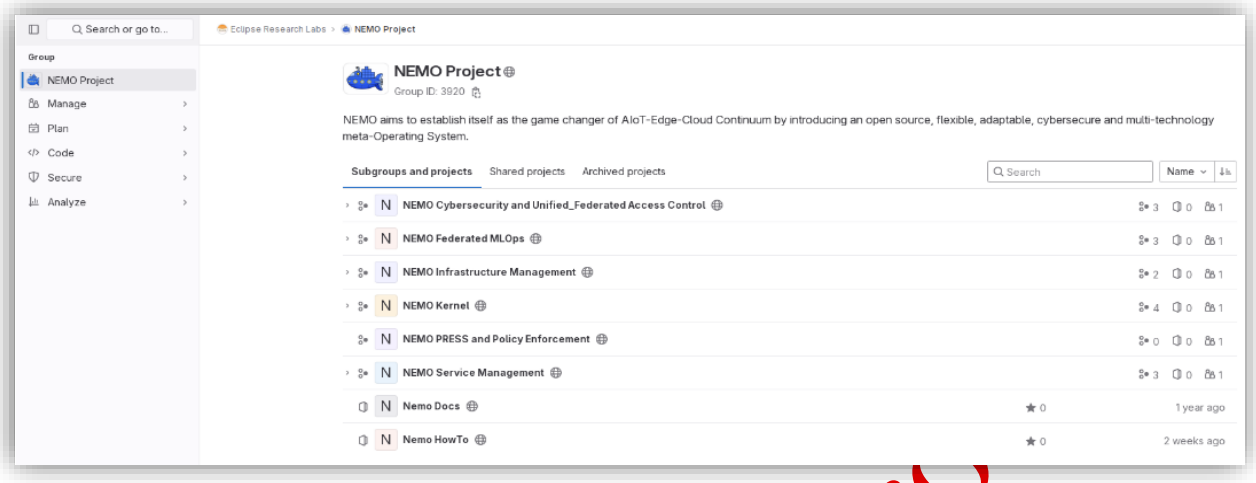


Figure 1. NEMO code repository in Eclipse Research labs

2.1.2 NEMO Automated Deployment and Configuration

In Software development one of the fundamental steps in the lifecycle of a system is the CI/CD development methodology. In this context, within NEMO a CI/CD pipeline was set up to automate the deployment and configuration of the NEMO components in various environments. In D4.2 the basic functionality of the NEMO CI/CD was described along with a short guide[4] for the developers.

The NEMO components are organized in a cluster-oriented hierarchy within the Flux CD [6] [7] continuous delivery solution as depicted in Figure 2. Each directory contains the configuration files of the components deployed at that particular cluster. In the final version of the NEMO meta-OS, apart from the component integration, the integration of 3rd party infrastructures took place. Figure 2 illustrates both the OneLab clusters and the pilot environments. The NEMO CI/CD [6] played a pivotal role as it was possible to deploy all the NEMO core components in the Pilots' clusters without the need for extensive configuration and involving the components' owners achieving a high impact with low effort curve.

[4] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo/-/blob/main/CI-CD%20Integration.md?ref_type=heads

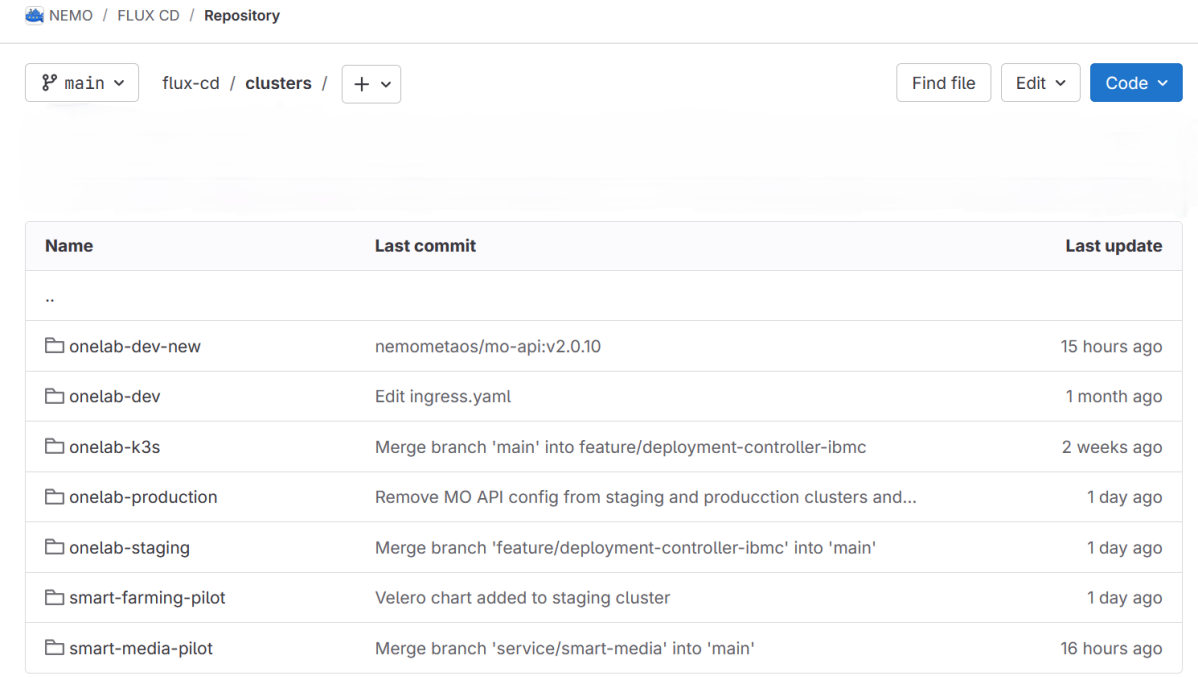


Figure 2. NEMO CI/CD organization

2.2 3rd Party documentation of NEMO meta-OS

Within the NEMO Meta-OS project, a framework that will enable the effortless deployment of services through Zero DevOps Tools was designed and implemented. The derived framework based on the best practices of microservices [8] architecture deployment in a Kubernetes [9] [10] [11] cluster environment provides to the 3rd party user two interfaces to interact with. A user-friendly graphical user interface based on the best practices of UI/UX and a REST API for service-to-service interaction with NEMO, allowing the 3rd party user to develop tailored interactions with NEMO.

The User is initially landed on the Login page of NEMO, Figure 3, to authenticate and be categorized based on the user categories described in deliverable D1.3 [12]. The authentication mechanism used is based on the Keycloak [13] framework which is a well-known and established identity management system widely used in modern critical systems. After the user is successfully authenticated, he is redirected to the main NEMO management page, Figure 4. In this view the main action categories for using NEMO are displayed:

- Workload Monitoring
- Workload Instances
- Intent Management[5,6]
- Resource Provisioning

More details about the functionalities of each category will be provided later in this section. Finally, the option log out on the top right corner of the view concludes the list of actions for this view.

[5] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-service-management/intent-based-sdk_api/intent-api/-/blob/main/intent-examples.yml?ref_type=heads

[6] https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-service-management/intent-based-sdk_api/intent-api/-/blob/main/network-intents.yaml?ref_type=heads

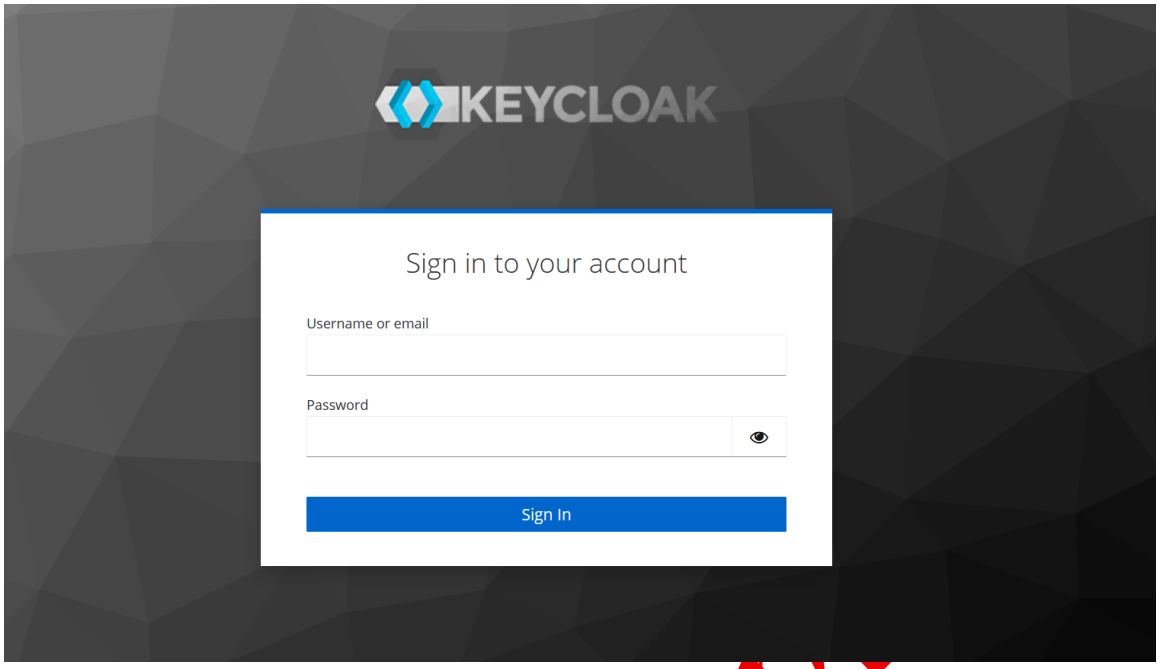


Figure 3. NEMO Login page

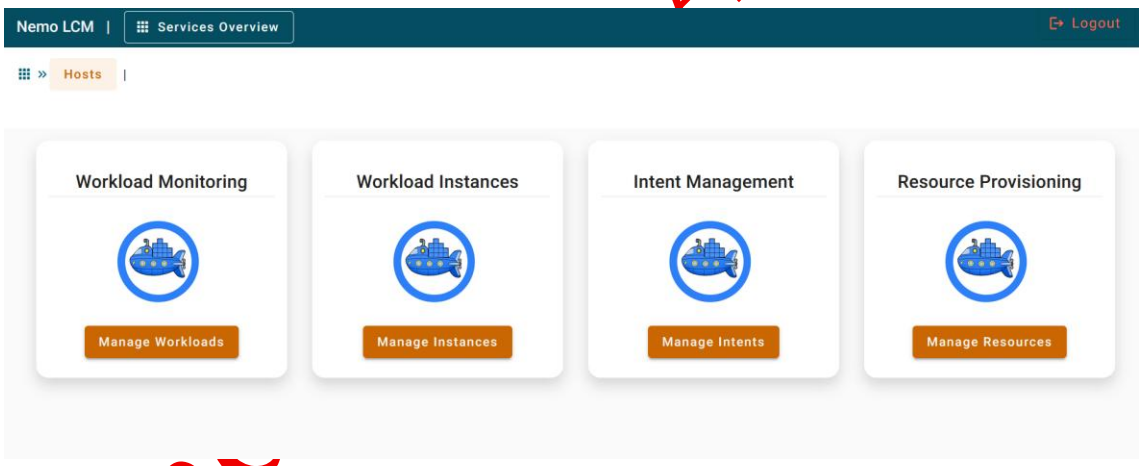


Figure 4. NEMO main management page

The Registered Workloads View, Figure 5, provides a complete monitoring interface for the already submitted workloads. Each workload record has its unique identification number sorted in a descending order. The Workload Name and Version attributes are the unique identifiers upon the creation of each Workload record, meaning that for a given workload name there cannot be two records with the same version. In the example displayed in Figure 5, the ml-retrain workload task is displayed multiple times, however the version differs each time. Then, there is the Status attribute, the values of this attribute are Pending and Accepted. When a workload is created it gets in the pending status until it is evaluated and validated by the NEMO framework. After this procedure is finished the workload gets in the accepted status. Next there is the Ingress Support field that denotes if the workload supports or not network communication with services outside of the NEMO environment. The User field displays the user that created, and it is used for trackability of user actions within the NEMO framework. Finally, the Actions column houses the functionalities related to the management of each workload. The first item is a file

uploader, and it is used to upload the HELM [14] file of the workload. The file can be uploaded after the creation of the workload and the workload cannot transit to the accepted status before this action is performed. The second item is the instance creation button, it is used to create a workload instance based on the workload's configuration, both from the workload creation page and the uploaded HELM file. The third item in the row is the deletion action in case the user wants to delete the workload. Finally, the associated instances button is a shortcut for the Workload instances view for this specific workload. In Figure 5 within the Actions column the HELM file uploader and the deletion buttons are faded, until a workload gets to the accepted status, the user is able to upload a new HELM file and/or delete the workload. After the workload is marked as accepted, the user cannot modify the existing workload, any change required on the existing workload requires the creation of a new workload.

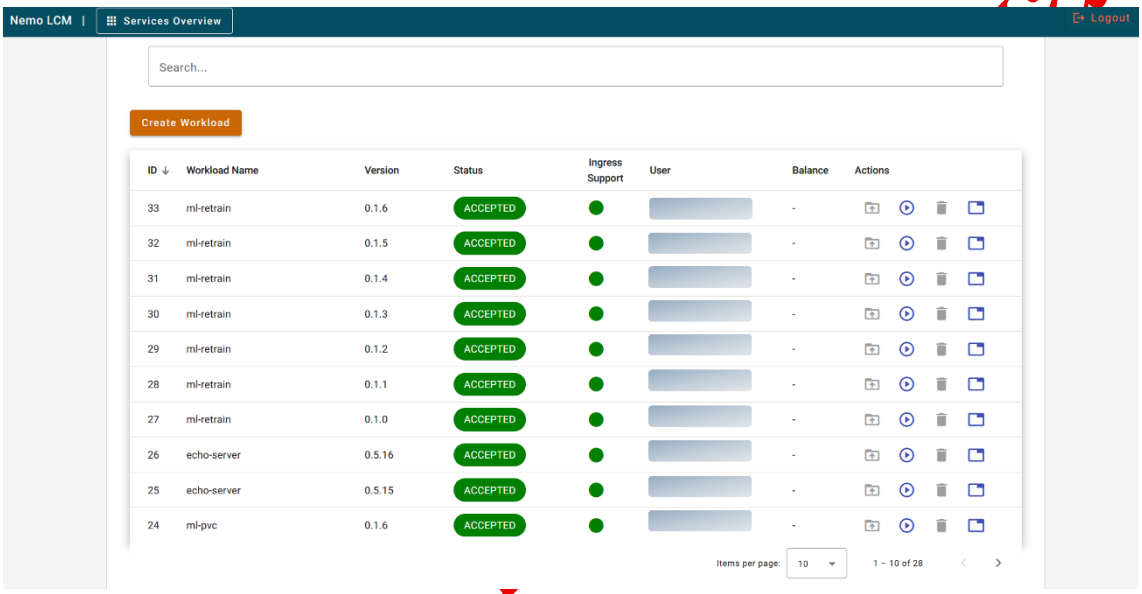


Figure 5. Registered Workloads View

In Figure 6 the workload creation form is displayed. In this view the user enters the basic information of the workload to be used, as we explained earlier the combination of the name and version attributes need to be unique and these fields are mandatory. The field Supported Intents contains a dropdown list of the available intent categories that the user can select for deploying the workload in a cluster based on the needs of the service, e.g., availability, existence of dedicated GPU, energy, etc. The user can modify the intents to be used on the cluster selection by the Meta-Orchestrator also after the workload's creation through the Intent Management view. Finally, there is the Ingress Support attribute discussed before. The values assigned are True or False, with False being the default value for security purposes.

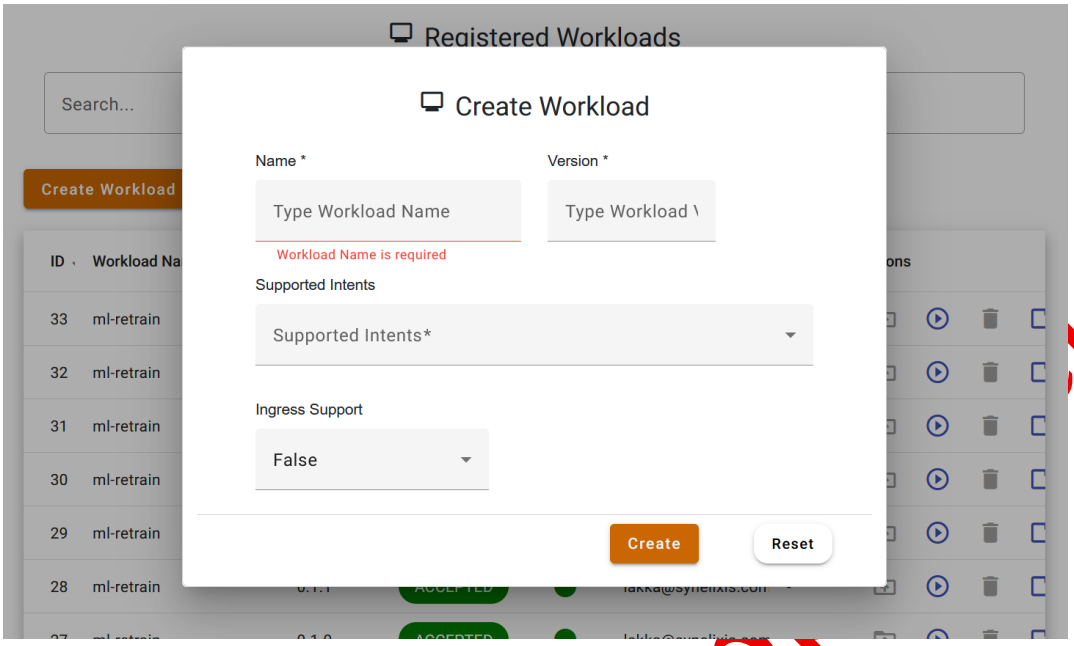


Figure 6. Create Workload View

The last step in the pipeline defined for executing a service in NEMO is the workload instance creation, Figure 7. After a workload is created and gets on the accepted status, an instance of it can be created. The user has to fill the name of the instance (Release Name) and if applicable a proposed cluster to be executed. The attributes: Values Override, Include CRDs, Is Upgrade and NoHooks are out of scope of NEMO as these are Kubernetes oriented parameters. The workload instance can be created without any further action, however, by clicking the Create Intent Template button a submenu opens. In this submenu the user is able to select one or more requirements that will enhance the execution of his service. In the sample displayed in Figure 7 shows that the desired execution environment for the service is a cluster able to provide an availability higher than 90%. Moreover, the user is able to set a specific time range that wants this migration to take place, e.g., high intense processing periods, etc.

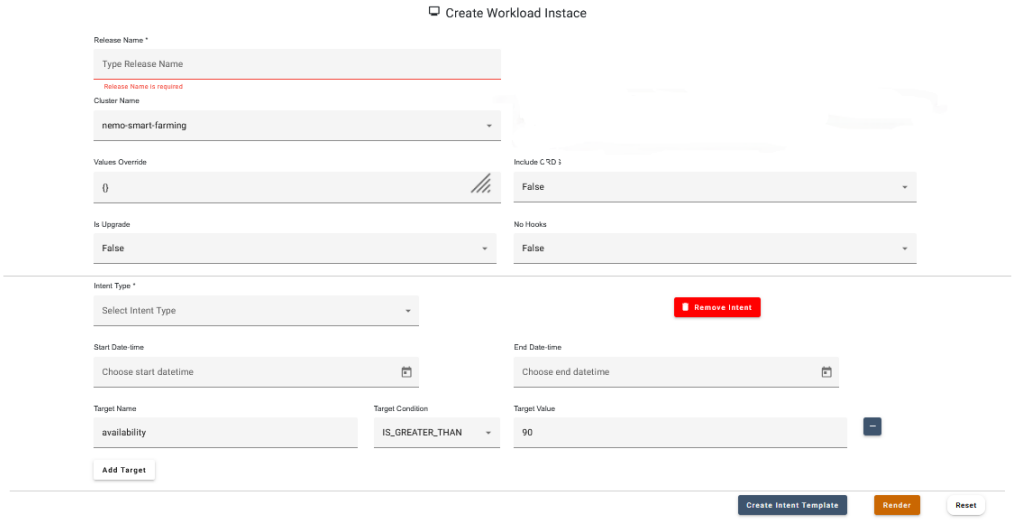


Figure 7. Create Workload Instance View

Another high-level category in the NEMO platform is Workload Instance View, Figure 8. In this view the user can see the status of the workload instances previously deployed and manage them. The Release name refers to the name provided in the previous form while the Workload ID refers to the parent workload created at the first step. The status denotes whether the instance is Rendered, Deployed or deleted. The Cluster field shows the cluster that the current instance is deployed based on the Meta-Orchestrator's decision-making algorithm and the requirements (Intents) that the user set up. The instance ID is the unique identifier of each instance. The combination of Release name and Instance ID is unique identifier of each deployed workload instance across all clusters connected to NEMO. Moreover, a set of useful interactions in terms of status monitoring and management are provided in the action column. First, a historic representation of the various statuses of the workload instance, e.g., created, deployed, suspended, deleted along with the timestamp of each status update. Then there is the manifest viewer, which can be used for examining the configuration parameters in case an error occurs and finally the deletion button can be used to permanently terminate the service instance.

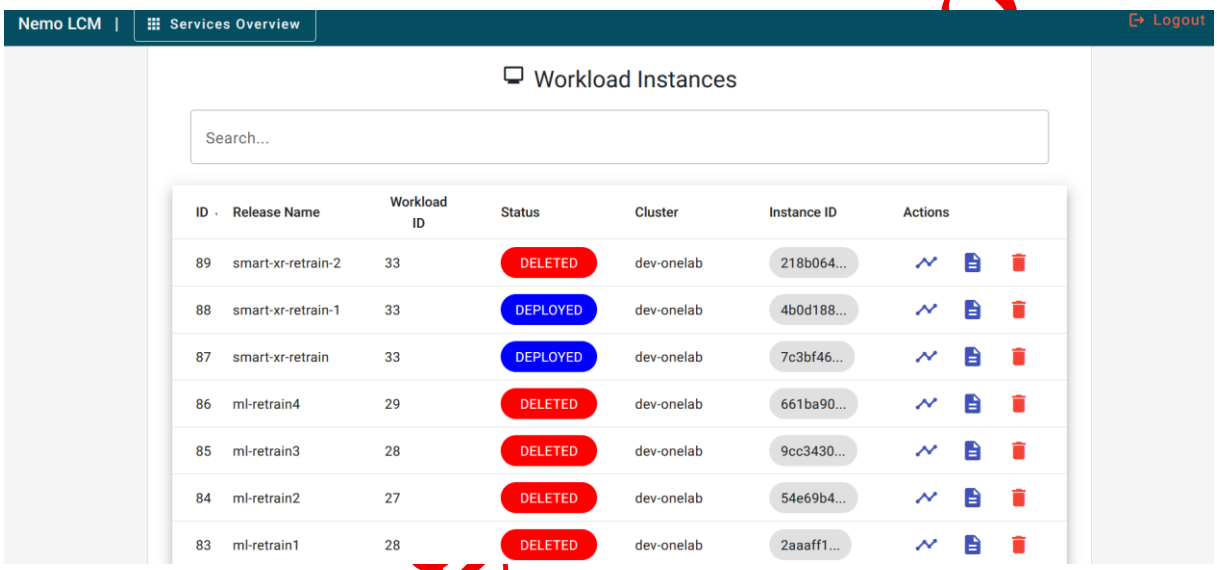


Figure 8 Workload Instances main View

NEMO was designed to migrate workloads based on specific requirements (Intents), for this reason the migrations take place based on tailored and sticked requests. NEMO provides to the user the possibility to modify on the fly these requirements for any given running workload instance. In this context, the Intents management View was created to house this functionality, Figure 9. As with the previously described views, the status field denotes the current condition of a specific Intent type on a specific workload instance. The status can be *Fulfilled* or *not fulfilled*, *compliant*, *degraded* or *fulfilmentfailed*. In the Intent Type column, the specific requirement (Intent) is displayed while in the next column the specific Workload instance is given. It is worth noting that one workload instance can have multiple intent types, even of different types. In the actions column there is the Intent Report option which provides helpful insights for understanding why an Intent failed by displaying detailed information of the error occurred. In addition, the user can modify the existing Intent by altering the desired target value. Finally, a simple control panel offers to the user the possibility to start and suspend the execution of an Intent with the related buttons as well as to delete it.

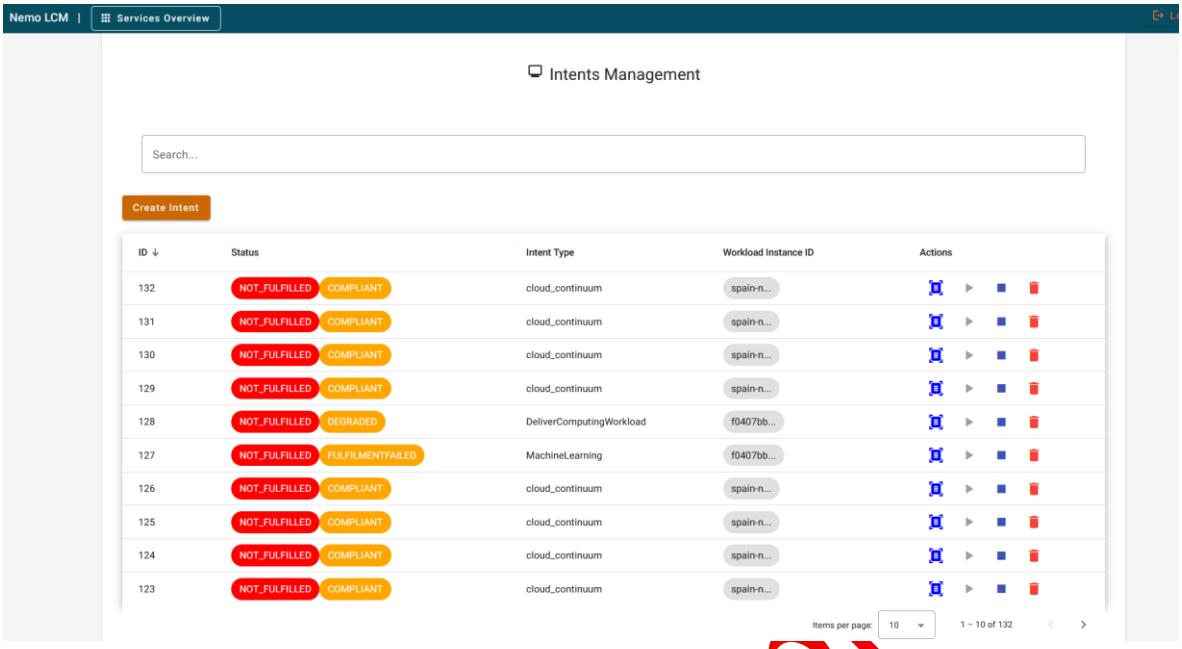


Figure 9. Intent Management View

From the Intents Management main View, the user is able to see the existing Intents but also to create new Intents, in Figure 10 the form of adding new Intents to an existing workload is displayed. The advantage of this view is that the user can add intents after the creation of a workload instance as well as for advanced users to create an intent using YAML files. The use of YAML files is used within NEMO for the connection and orchestration of complex components such as the mNCC for network creation for pods across the clusters.

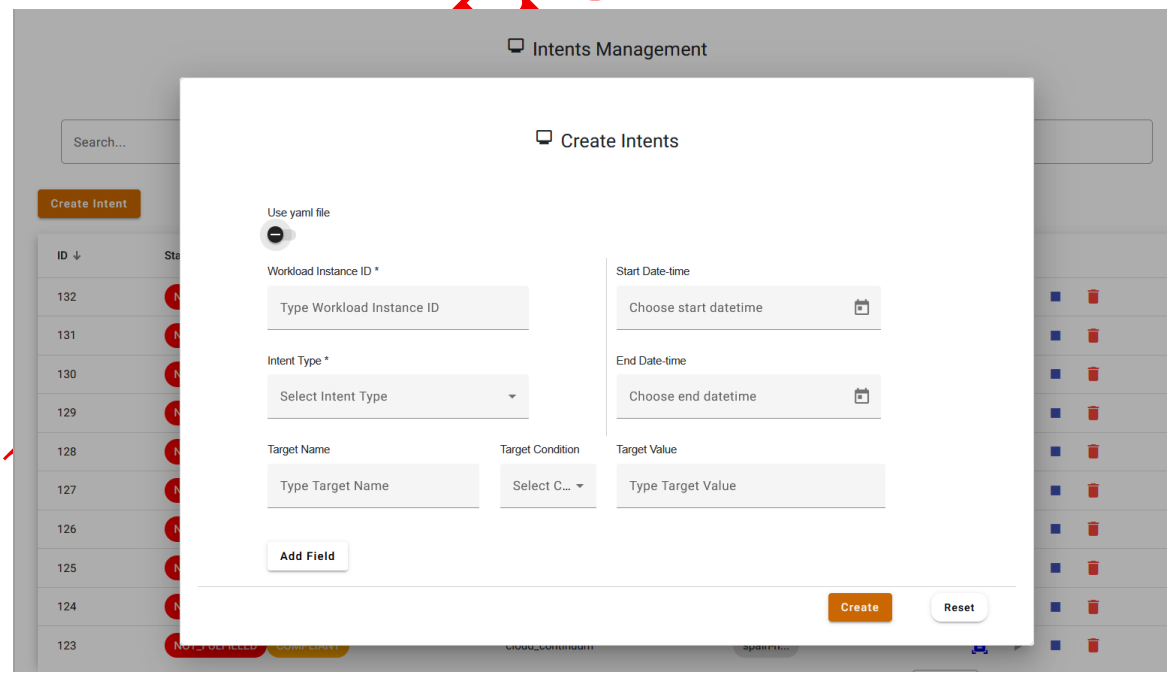


Figure 10. Create Intent View

The last menu provided by the NEMO Management User Interface is the Resource Provisioning View, Figure 11. In this View the user can see the available clusters connected to the NEMO platform along with some useful information such as their processing capabilities and real-time performance metrics.

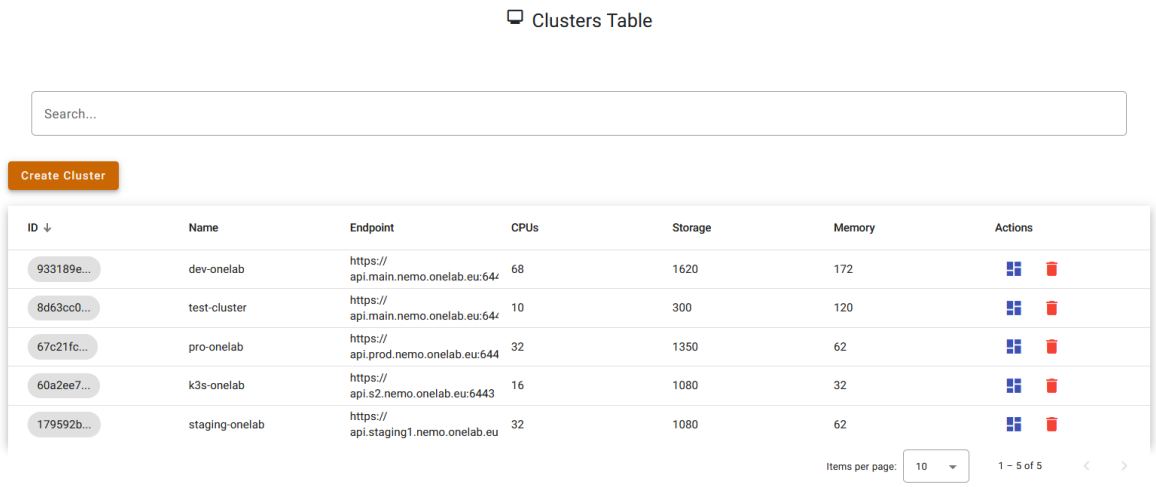


Figure 11. Resource Provisioning View

Moreover, through this menu option, the users providing their own cluster infrastructure can register them through this view, Figure 12. The infrastructure providers need to fill the information related to the processing, environmental and financial aspects of their machines. The cluster registration process is divided into two steps, a) provide the basic information used by the NEMO components for the decision-making process of pods' deployment and b) the underlying inter-cluster connectivity. The registration form presented is responsible for tackling the first part of the registration, while the second part of the registration is achieved by uploading the kubeConfig file of the Kubernetes cluster to be registered. The Graphical User Interface previously described is one of the ways to interact with the NEMO platform. The GUI is indented to be used from non-technical personnel that need to effortlessly deploy a workload.

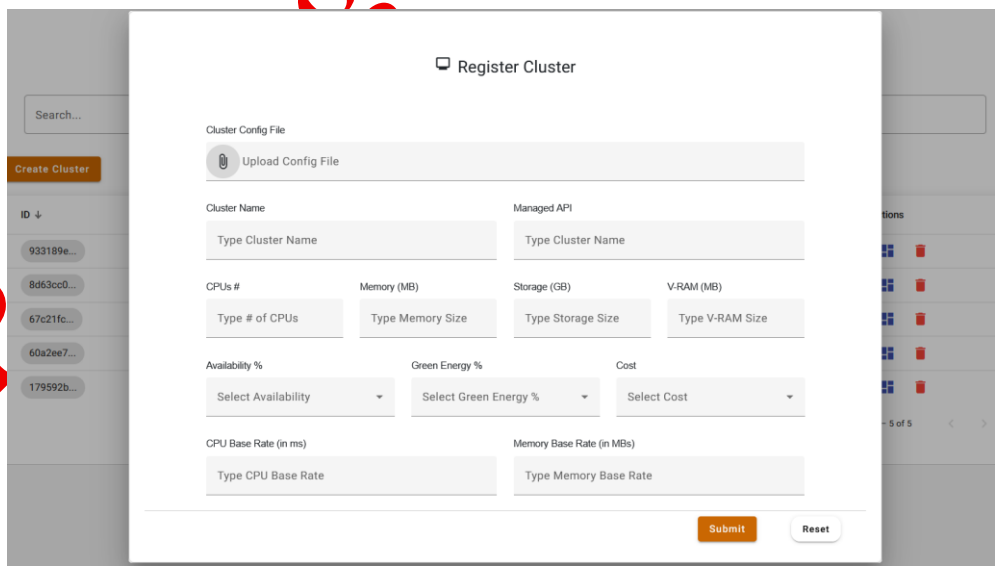


Figure 12. Cluster Registration Form

Another way of interacting with NEMO is through a REST API [15] , Figure 13. Intent-Based API endpoints. Through this solution a power user can make a set of API calls in order to create tailored pipelines in terms of deploying, configuring and monitoring applications (workloads). A documentation page[7] was created to enable 3rd party interactions by providing both a complete set of instructions and the data types that are being used as well as a sandbox environment where the user can interact without coding or an external GUI for REST calls.

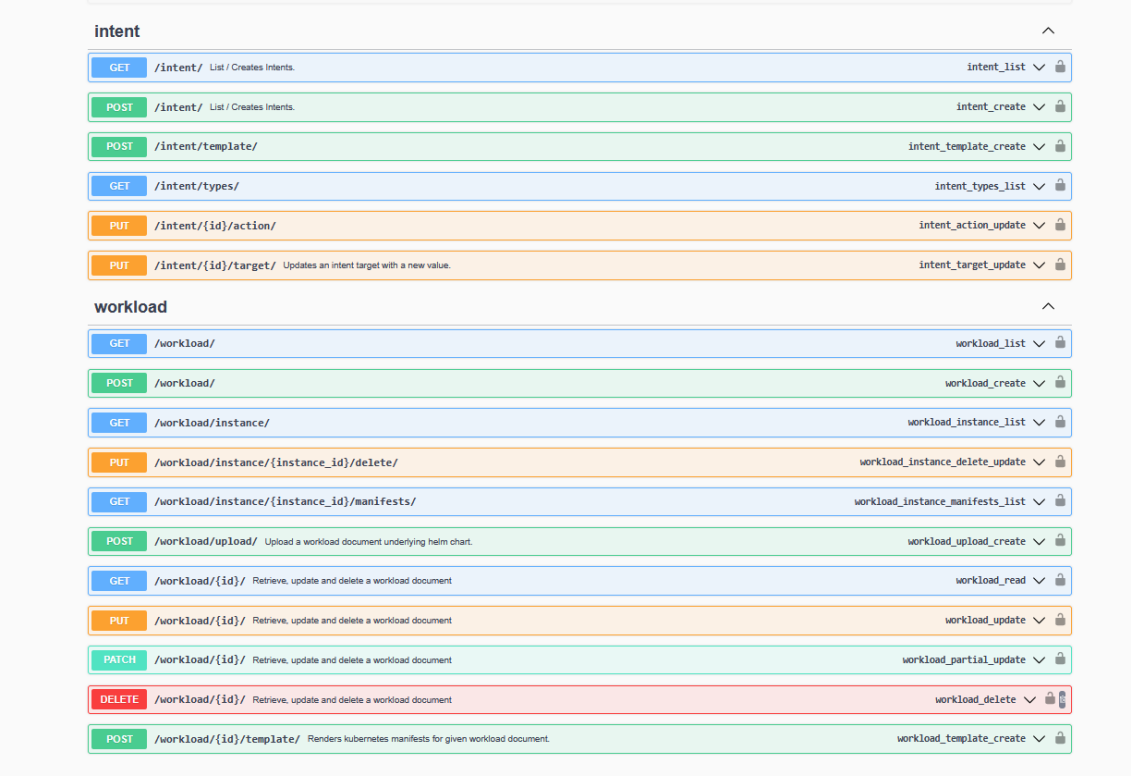


Figure 13. Intent-Based API endpoints

Figure 14, presents a sample of the expected parameters' body of the workload POST endpoint.

[7] https://intent-api.nemo.onelab.eu/api/v1/swagger/

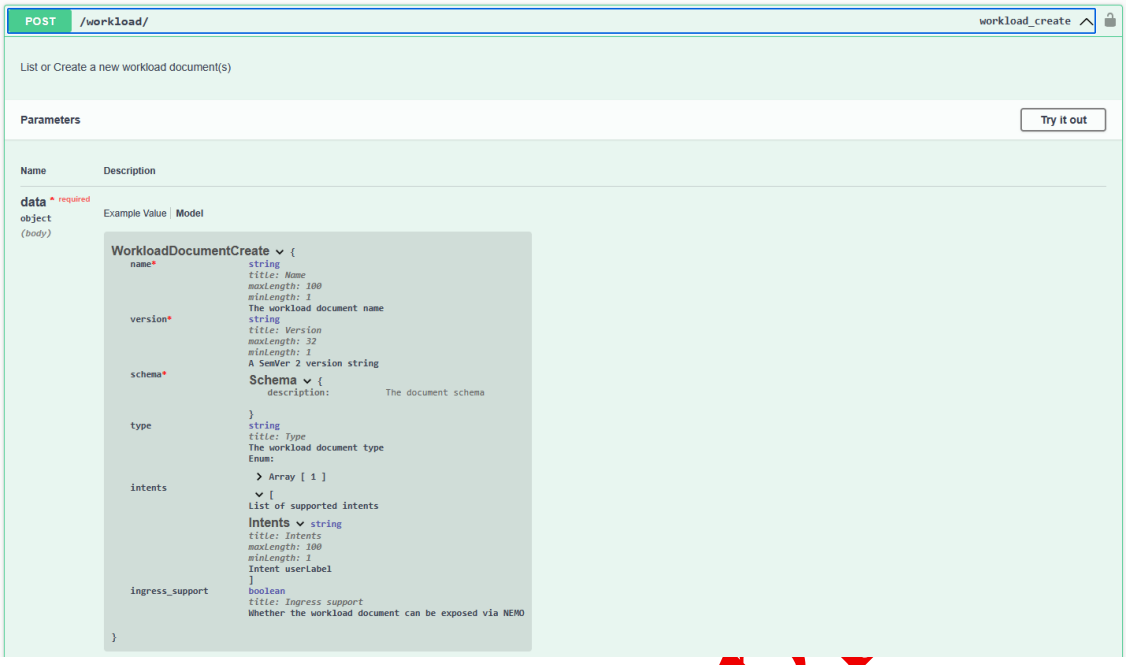


Figure 14. Sample of the expected parameters' body of the workload POST endpoint

2.3 Cloud/Edge/IoT Integration and Validation Infrastructure

The complete integration of different systems in the Cloud-Edge-IoT continuum into a single framework is a very complex task and requires a lot of effort and precision. The individual systems in each layer of the continuum Cloud, Edge and IoT are designed to provide max throughput, low latency and increased processing capabilities to meet the requirements of a modern system. While these systems can flawlessly work individually, several issues may arise when they need to be interconnected. NEMO is a complex framework that aims to provide severe automations at a low level across the continuum in terms of deployment and device management. In a typical Cloud-Edge-IoT continuum, the cloud computing layer provides a centralized environment where the main processing power and storage space exist. On the contrary, Edge serves, as the frontier of processing as they are located close to the IoT devices. These are responsible for processing the incoming data and taking decisions in real-time.

Within the NEMO framework an operational management of the Cloud-Edge-IoT continuum takes place. The infrastructure where the NEMO meta-OS platform is being deployed varies based on the equipment present on the Pilots' premises. To guarantee a complete and flawless integration of the platform, a set of minimum requirements was distributed among the partners. This guaranteed that the underlying technologies, such as Kubernetes can work without issues. Typically, the main issue within the Cloud-Edge-IoT continuum interactions is the connectivity due to the communications taking place over different technologies with varying capabilities, e.g., wireless channels, low bitrate channels, etc. In NEMO, this was considered since the design phase where lightweight communication protocols and messaging mechanisms, such as RabbitMQ [16] that are resilient to packet loses and require a small set of resources. These channels within the proposed meta-OS are used to exchange real-time information about the resource availability on the connected clusters which is crucial for the NEMO users as it can be used for taking strategical decisions on where a workload should be deployed. In addition, the channels are able to handle high volume data images for migrating the workloads along with the related data volumes.

2.4 Integration & V&V Methodology & Plan

Since the NEMO conceptualization phase, a detailed plan for the integration and validation phases was created. This plan was initially presented in deliverables D1.3 and D4.1 while an updated version reporting the alignment with that plan as well as the updates of the tools was presented in deliverable D4.2. The plan was aiming to create a universal cross component and pilot template for monitoring the various phases of development to keep track of the updates and to identify any risks arose. The plan adopted is based on the best modern practices of software development, NEMO followed an agile and incremental approach of iteration cycles, grouped in 3 Phases, as depicted in Figure 15.

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.

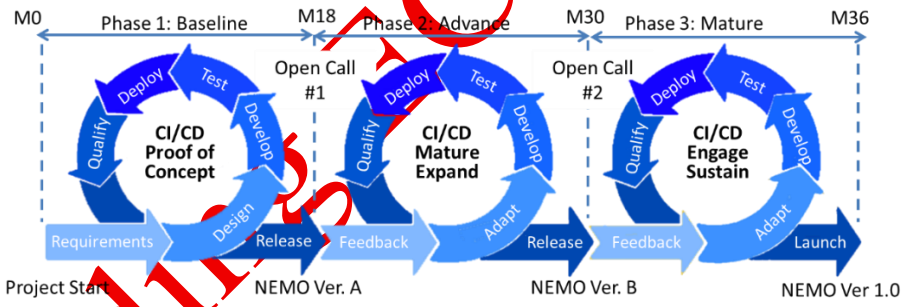


Figure 15: NEMO project phases and main meta-OS version releases

At the competition of each phase a retrospective analysis along with a risk management assessment took place. This guaranteed that the project would proceed based on the plan conducted at the beginning of the project and any deviation or risk appeared during the development phase would be immediately identified and resolved without jeopardizing the progress of the project. The template presented in the following subsection was used to document and monitor any issues occurred.

2.4.1 NEMO V&V documentation

The NEMO End-to-End integration pipeline is a complex procedure; in order to provide high-quality tests, the End-to-End pipeline was divided in smaller and simpler test pipelines. Associated technical details and stemming results are provided in Section 5 of this document. As mentioned earlier, a common template applicable across the different functional scenarios present in NEMO was defined. This played a pivotal role as it allowed to properly document all the possible steps and interaction took place in an organized and well-written manner. At each iteration phase presented before, the following template was evaluated, and new features were introduced in case a feature was missing or modified in case it

was not adding value to the project. To better understand what needs to be done as well as to verify the goals' success, three main attributes were defined:

- Scenario
- Outcome
- Checkpoints

In Table 2 the template for the integration tests explaining the scenarios and what needs to be tested along with the functional and non-functional requirements for each scenario are documented. Moreover, an overview of the test plan is present to assist the engineers to perform the tests. After the completion of a test procedure, the involved partners were filling in the integration scenario checkpoint template, Table 3, which was later used in the retrospective analysis to provide information about the open issues.

Table 2. Integration testing - scenario template

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

Table 3. NEMO integration testing - Checkpoints 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?	✓		

3 NEMO Integrated Platform (Final Version)

The final version of the NEMO meta-OS was established following a comprehensive integration and validation procedure. The conducted integration tests concerned both the NEMO developed technical solutions and the 3rd party ones introduced by the 2 Open Calls.

3.1 Meta-OS functionality in NEMO ver. 1.0

3.1.1 NEMO Infrastructure Management

The infrastructure management across the various Kubernetes clusters has been designed for scalability, flexibility, and high availability, utilizing a multi-cluster approach. Several clusters have been deployed, each tailored for specific roles such as control planes, worker nodes, and specialized hardware, including GPU-enabled nodes.

- **Nemo main cluster**: A control-plane node (nemo-dev-master) and five worker nodes (nemo-dev-worker1 to nemo-dev-worker5) have been configured within this development cluster, running Kubernetes version 1.29.13. This setup provides a robust environment for development and testing purposes. Table 4 provides a detailed view of the technical specifications of the NEMO main cluster.

Table 4. NEMO dev cluster (K8S)

Node name	Node Type	Specifications
k8smaster.onelab.eu	Master	• **CPU:** 8 CPU Cores • **RAM:** 16GB • **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:** 32GB • **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
k8sworker2.onelab.eu	Worker and Storage	• **CPU:** 16 CPU Cores • **RAM:** 32GB • **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

Node name	Node Type	Specifications
k8sworker3.onelab.eu	Worker and Storage	• **CPU:** 16 CPU Cores • **RAM:** 32GB • **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
k8sworker4.onelab.eu	Worker and Storage	• **CPU:** 16 CPU Cores • **RAM:** 32GB • **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
k8sworker5.onelab.eu	Worker and Storage	• **CPU:** 16 CPU Cores • **RAM:** 32GB 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

- **Nemo Staging 1 Cluster**: Focused on scaling operations, this cluster comprises a control-plane node (nemo-s1-master) and several worker nodes (nemo-s1-worker1 to nemo-s1-worker3), running Kubernetes versions 1.30.7 and 1.31.3, thus ensuring reliability for both development and production workloads. Table 5 presents the technical details of the NEMO Staging 1 cluster.

Table 5. Staging 1 cluster (K8S)

Node Name	Node Type	Specifications
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
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
nemo-s1-worker2	Worker and Storage	• **CPU:** 16 CPU Cores • **RAM:** 16GB

Node Name	Node Type	Specifications
		• **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
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

- **Nemo Staging 2 Cluster**: The lightweight k3s-based cluster consists of a control-plane node (nemo-k3s-master) and three worker nodes (nemo-k3s-node-1 to nemo-k3s-node-3), operating on version 1.30.6+k3s1. A specialized AMD-based worker node (nemo-k3s-node-3) was added to support the integration of the NEMO component, ensuring optimal performance for resource-intensive tasks. More details about the Staging 2 cluster are presented on Table 6.

Table 6. Staging 2 Cluster (K3S)

Node name	Node Type	Specifications
nemo-k3s-master	Master	• **CPU:** 4 CPU Cores • **RAM:** 8 GB • **Storage:** 64 GB External SSD • **OS Image:** Ubuntu 24.10 • **Kernel Version:** 6.11.0-1004-raspi • **Container Runtime:** containerd://1.7.22-k3s1
nemo-k3s-node-1	Worker and Storage	• **CPU:** 4 CPU Cores • **RAM:** 8 GB • **Storage:** 1TB External SSD • **OS Image:** Ubuntu 24.10 • **Kernel Version:** 6.11.0-1004-raspi • **Container Runtime:** containerd://1.7.22-k3s1
nemo-k3s-node-2	Worker and Storage	• **CPU:** 4 CPU Cores • **RAM:** 8 GB • **Storage:** 64 GB External SSD • **OS Image:** Ubuntu 24.10 • **Kernel Version:** 6.11.0-1004-raspi • **Container Runtime:** containerd://1.7.22-k3s1
nemo-k3s-node-3		• **CPU:** 16 CPU Cores • **RAM:** 16 GB • **Storage:** 120GB Ephemeral + 150GB Ceph • **OS Image:** Ubuntu 24.10 • **Kernel Version:** 6.11.0-19-generic • **Container Runtime:** containerd://1.7.12

- **Nemo Production Cluster**: Dedicated to production workloads, this cluster includes a GPU-enabled worker node (nemo-prod-gpu-worker) and several worker nodes (nemo-prod-worker1 to nemo-prod-worker3), along with a control-plane node (nemo-prod-master), all operating on Kubernetes version 1.30.7. This configuration ensures high-performance processing for tasks such as machine learning. The NEMO Production cluster details are demonstrated on Table 7.

Table 7. Production Cluster (k8S)

Node Name	Node Type	Specifications
nemo-prod-master	Master	• **CPU:** 4 CPU Cores • **RAM:** 8 GB • **Storage:** 80 GB Ephemeral • **OS Image:** Ubuntu 22.04.3 LTS • **Kernel Version:** 5.15.0-78-generic • **Container Runtime:** containerd://1.7.12
nemo-prod-worker1	Worker and Storage	• **CPU:** 8 CPU Cores • **RAM:** 16 GB • **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
nemo-prod-worker2	Worker and Storage	• **CPU:** 8 CPU Cores • **RAM:** 16 GB • **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
nemo-prod-worker3	Worker and Storage	• **CPU:** 8 CPU Cores • **RAM:** 16 GB • **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
nemo-prod-gpu-worker	Worker and Storage	• **CPU:** 4 CPU Cores • **RAM:** 8 GB • **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

To mitigate the impact of power outages, dedicated UPS systems have been implemented for each host running virtual machines that support the cluster nodes. This solution ensures uninterrupted operation during power failures, effectively reducing downtime. These infrastructure improvements, along with the addition of specialized hardware for NEMO integration, have significantly enhanced the resilience, performance, and reliability of the entire system.

3.2 NEMO Open Call integration activities

In order to provide homogenised, flexible orchestration of varied workloads across heterogeneous and scattered devices, the concept of the meta-Operating System (meta-OS) refers to the efficient integration of highly diversified hardware and software resources. Because of NEMO's expandable architecture, additional features can be added as NEMO plugins. Since the NEMO Kernel serves as the core system, plugins are designed to add capabilities to the core, such as flexibility, extensibility, and isolation of bespoke meta-OS logic or applications. Horizontal or domain-independent services that seek to offer some fundamental and standard features that expand the NEMO capabilities are referred to as plugins.

NEMO has already realized extensions through additions developed by the 1st Open Call projects. NEMO 1st Open Call invited SMEs active as edge computing, edge and/or native cloud software development, operating systems, IoT/5G networks and IoT manufacturing entities to join the NEMO ecosystem by offering: a) NEMO meta-architecture extensions b) software components/plugins not covered by current NEMO implementation plan c) new network or service/resources metering/automated control components or d) porting NEMO on new, highly heterogeneous IoT devices.

Six winning proposals have been identified, specifically MetaFOX, by Vodéna, Serbia working on Machine Learning and AI technologies; CorMOS, by Business & IoT Integrated Solutions, Cyprus focusing on cross-domain orchestration of Meta-OS edge resources; ARGO, by Intellia ICT, Greece, investigating porting AR/XR technologies in the meta-OS; Eros4NRG, by Martel Innovate BV, Netherlands offering ZeroTrust IoT Analytics with focus on Smart Energy applications; GENESYS, by SWHARD srl, Italy providing an edge gateway for the NEMO Meta-OS; and MARINEMO, by BEAM Innovation, Romania working on an efficient resource utilization and maritime network slicing plugin for the NEMO Meta-OS.

In addition, NEMO introduced Open Call #2 in the framework of which the finalized NEMO integrated framework will be further validated, as the NEMO Open Call #2 projects will utilize the NEMO meta-OS functionality supporting the various use cases realized by the 3rd parties.

The rest of the chapter provides a brief description of the NEMO Open Call #1 projects' technical concept and describes the integration details of each technical solution developed as a NEMO meta-OS architecture extension.

3.2.1 ARGO

The ARGO project, standing for "Augmented Reality next Generation Operational Systems in NEMO", is a pioneering initiative designed to align seamlessly with the objectives and scope of the Next Generation Meta Operating System (NEMO) program. The motivation behind ARGO's proposition lies in its integration with the NEMO architecture, enhancing the existing framework with innovative AR solutions. At the core of ARGO is the integration of NEMO's flexible meta-architecture with specialized AR devices, such as the Vuzix M4000 and Vuzix Shield, and the backend support system of the AR devices. These lightweight, monocular, and binocular AR glasses are designed for prolonged industrial use, offering immersive experience without compromising user comfort during full-shift operations. ARGO's target is to enhance industrial efficiency, safety, and training using advanced AR technology. The ARGO system has been already deployed on the NEMO provided development environment in OneLab and preliminary tests of the system on the premises of the Foundation of the Hellenic World (FHW) in the context of Smart Media/ City & XR Living Lab were conducted.

3.2.2 CorMOS

The CorMOS Orchestration Engine is a specialized software component developed to facilitate the deployment of applications to appropriate cluster nodes based on user-defined requirements, goals, and intents. The functionality of this software component relies on data stored in a JSON file that captures the application's or users' goals/intents in a platform-independent way, along with a YAML file that specifies the internal components of the application, their interactions, and additional resource constraints. The CorMOS Orchestration Engine further analyses the data in the JSON file and updates the YAML file with the appropriate instructions (e.g., node selectors and/or affinity rules) to ensure that the components of the application are deployed to the correct nodes of the cluster, thus achieving user goals such as performance optimization and efficient resource utilization. Moreover, the CorMOS developed Telemetry Data Management system focuses on a few parameters of particular interest that directly affect the behavior of the CorMOS Orchestration Engine.

The CorMOS orchestration engine will be integrated with the NEMO architecture and its logical placement would be within or alongside the Meta-Orchestrator (MO). The Meta-Orchestrator plays a

critical role in coordinating workload execution, deployment, and migration across the IoT-Edge-Cloud continuum. Given its pivotal function, the CorMOS Orchestration Engine could interact with key NEMO components to ensure seamless integration and to leverage the platform's existing capabilities effectively. From an architectural standpoint, the CorMOS Orchestration Engine should be positioned as a high-level component within the orchestration layer. This placement would allow it to either extend the Meta-Orchestrator (MO) functionality while serving as the decision-making hub for workload execution in context-specific infrastructures and environments. To achieve optimal orchestration, the CorMOS Orchestration Engine should retrieve information from various monitoring, networking, security, and policy enforcement components, ensuring that workload placement and execution are conducted in an intelligent, secure, and resource-efficient manner.

3.2.3 Eros4NRG

Eros4NRG is a platform designed to monitor and analyse energy production and consumption, enabling more efficient decision-making through predictive analytics. It acts as a one-stop-shop (OSS) for energy stakeholders by integrating data from electric vehicles (EVs), EV charging stations, photovoltaic (PV) plants, and smart headquarters. The platform enhances trust and transparency in energy data, while addressing key challenges such as: (i) Unreliable data from IoT sources impacting machine learning operations; (ii) poor organization of static energy data leading to information loss; and (iii) the need for user-friendly AI/ML insights for non-technical stakeholders.

Eros4NRG's final integration is guided by a set of clear functional and non-functional requirements, ensuring that the system operates efficiently across modern cloud-native infrastructures (such as MinIO, PostgreSQL, and Docker). Designed with a service-oriented architecture, the platform enables streamlined data collection, processing, visualization, and security. The Eros4NRG project relies on extracting raw data from EMOTION and ASM Terni systems via API calls. This continuous stream of data enables the platform to monitor and analyse energy consumption and production, as well as EV battery performance in (near) real time. The module starts by sending API requests to the EMOTION and ASM Terni systems. Building on the final integration, Eros4NRG will undergo several enhancements and additions to maximize its potential and improve user experience.

3.2.4 Genesys

GENESYS project aims to extend NEMO's scope and technology by porting the meta-operating system on a new IoT Edge device, fully customisable and industry-ready, developed by SWHARD. Targeting the Industry 5.0 and G-IoT paradigms, aligned with the NEMO *Smart Manufacturing & Industry 4.0 Use Cases & Living Lab*, the project's primary goal is to install NEMO on an industrial-grade System-on-Module (SOM), delivering a functional product while providing extensive documentation for seamless installation on Edge platforms.

FLEX is categorised as a micro-edge system. Conventionally, the purpose of this class of edges is to collect data from field sensors, do some light pre-processing and transfer aggregated information to the cloud. Despite the low resources available (by definition), the expected average workload for the CPU is usually very low in a common edge-cloud design.

In NEMO's perspective the micro-edge is a cluster node which still serves the two purposes above. Being part of the cluster, it will also provide processing power not only for the elaboration of its "own" sensors, but for other tasks. The Meta-Orchestrator is responsible to decide, deploy and monitor the tasks to be run in every node, including the micro-edges. This concept leads to a greater optimization of all the available resources in the cluster.

Following the MVP implementation of the Genesys framework, the successful porting of the Software components required to let the FLEX enter the NEMO META-OS ecosystem, is the keystone of GENESYS. Although the work has not yet demonstrated with a real application, the FLEX nodes have

been successfully joined the test cluster, ran a simple application and a set of benchmarks. A side activity that has been done was the test of the integration of Coral NPU into the cluster.

3.2.5 MARINEMO

The MARINEMO Slice Manager Plugin (SMP) is designed as an add-on for 5G-enabled cellular networks, its main purpose being to efficiently optimize the communication resources of the targeted systems by applying AI/ML-powered intents for slicing and user profile reconfigurations.

The API Engine of the Slice Manager Plugin facilitates external interactions, providing a crucial interface for third-party systems to influence and monitor the network's behaviour. It offers APIs for defining testbed-specific operational parameters, alerting mechanisms, and performance analysis. This component not only enhances the plugin's adaptability to different environments but also ensures that it can serve a broad range of use cases from operational adjustments to detailed performance analytics.

Regarding the Slice Manager Plugin integration points, the plugin needs to be connected to testbed-specific agents for network API exposure and performance monitoring. Specifically, the MARINEMO Slice Manager Plugin will be integrated within META-OS ecosystem through the Telefonica Private 5G testbed and its Cumucore 5G Core Network APIs. Furthermore, the SMP will be instantiated on top of a container orchestration platform, more specifically, the OneLab Kubernetes cluster.

The MARINEMO SMP will be deployed on the META-OS Kubernetes-based virtualization platform by following the next steps.

- Access to the Kubernetes cluster for testing the plugin - for testing and validating the SMP interactions with the META-OS APIs the plugin docker-based container will be directly deployed on the NEMO Kubernetes cluster. Access to the META-OS Docker Hub repository - for testing and validating the deployment capabilities of the NEMO META-OS platform, access to the META-OS Docker Hub repository will be used for uploading the SMP docker image and instantiating the container on the Kubernetes cluster by running the META-OS Gitlab CD scripts.
- Access to the Gitlab META-OS repository for complete CI/CD - for testing and validating the complete CI/CD capabilities of the META-OS platform, access to the project-related Gitlab repository will be used for uploading all the SMP source code and running the CI scripts for creating the plugin docker image and the CD scripts for instantiating the container on the Kubernetes cluster.

3.2.6 MetaFOX

The MetaFOX project[8] is an advanced automated machine learning (AutoML) component, which significantly simplifies the initial model creation within NEMO META-OS by automating the process of model selection, feature engineering, and hyperparameter tuning. AutoML is a transformative approach that simplifies complex machine learning processes, enabling both experts and non-specialists to design and deploy models efficiently. MetaFOX uses the power of automation in machine learning to streamline model development, enhance the quality of models, and democratize AI accessibility. It significantly simplifies the initial model creation for federated learning (FL) and transfer learning (TL) by automating the process of model selection, feature engineering, and hyperparameter tuning.

The MetaFOX solution, when deployed on Kubernetes, consists of a collection of components running in separate Pods, each governed by Deployments that handle the creation and scaling of these Pods. Services provide stable network endpoints for communication within the cluster, while Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) ensure the durability of data for stateful components. In the context of the deployment, Minikube was used as the Kubernetes environment on a

[8] https://metafox.readthedocs.io/en/latest/

dedicated Linux server, providing a local cluster suitable for both development and production-like evaluations.

Preliminary integration tests were conducted achieving a successful deployment providing for early validation of the component's compatibility with OneLab's infrastructure. The successful deployment carried out has provided sufficient evidence of the MetaFOX component's compatibility with the OneLab environment. These results support the feasibility of full-scale integration, and they lay the groundwork for further testing and optimization. Future work will focus on consolidating these findings into a robust deployment procedure tailored for OneLab.

4 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 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.

4.1 Intent-based Migration Controller

4.1.1 Overview

The Intent-based Migration Controller (IBMC) is responsible for orchestrating the seamless transfer of workloads across the IoT, Edge, and Cloud Continuum. Rather than relying on manual configurations, it uses intent-based networking to automatically interpret high-level objectives and translate them into concrete migration actions. This allows the system to adapt to changing conditions, ensure efficient use of resources, scale effectively, and uphold service reliability. The IBMC's role is essential in managing the highly dynamic and distributed nature of the meta-OS environment.

4.1.2 Architecture

The IBMC architecture is represented in Figure 16, which follows the hierarchical structure of the Open Cluster Management (OCM) HUB-managed cluster[9]. The key sub-components that make up this architecture are described below:

- **Ibmc-controller**: This component serves as the entry point for initiating the workload migration process. Upon receiving a new intent from the Intent-Based API, the ibmc-controller checks the current workload status and determines whether it complies with the specified intent parameters. If any of these parameters are not satisfied, the controller selects a suitable target cluster from the list of available managed clusters and sends a migration trigger message to the ibmc-agent deployed in the selected cluster.
- **Ibmc-agent**: The ibmc-agent functions as a black-box component responsible for executing the actual migration process. Each cluster runs its own ibmc-agent, which listens to a dedicated RabbitMQ queue for incoming migration messages. When a message arrives at the source cluster, the corresponding ibmc-agent initiates a backup of the specified workload using Velero[10]. Upon successful completion of the backup, a message is sent to the target cluster, where the local ibmc-agent receives the message and proceeds to restore the workload accordingly.
- **Velero:** Velero is a key technology in the workload migration process, as it is deployed across all clusters and ensures that workloads can be efficiently backed up and restored, enabling seamless transitions between clusters. Its integration into the migration workflow allows for consistent data protection and minimizes downtime during migrations.
- **Rook Ceph:** Rook Ceph[11] is used as an S3-compatible object store for Velero, providing reliable and scalable storage for backup data. By integrating with Velero, Rook Ceph ensures that

[9] https://open-cluster-management.io/docs/concepts/architecture/
[10] https://velero.io/
[11] https://rook.io/docs/rook/latest/Storage-Configuration/Object-Storage-RGW/object-storage/

workload snapshots and metadata are securely stored and readily available for restoration during migration processes.

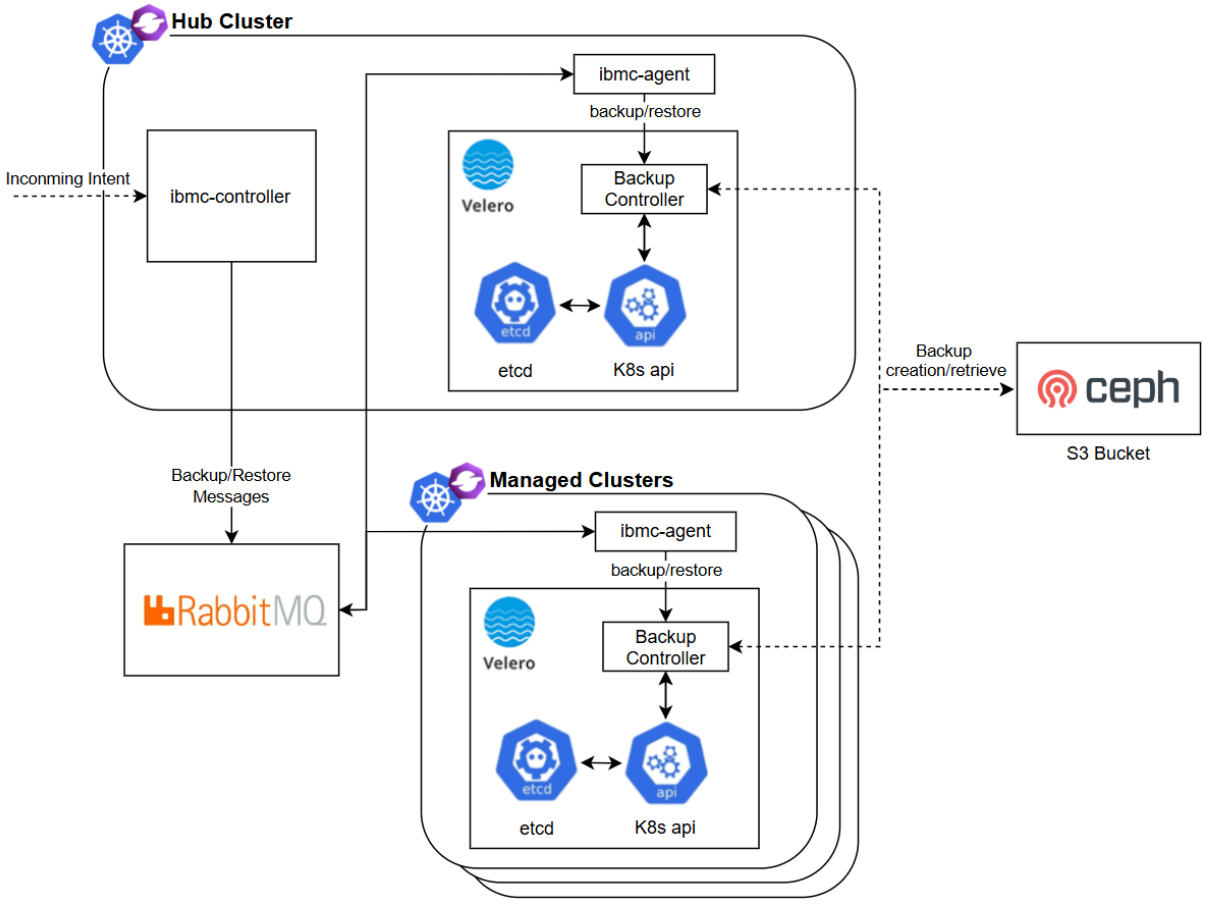


Figure 16. IBMC Architecture

Figure 17, depicts the sequence diagram that concerns the NEMO workload migration process. The corresponding integration and validation test are presented in section 5 of the document.

4.1.3 Interaction with other NEMO components

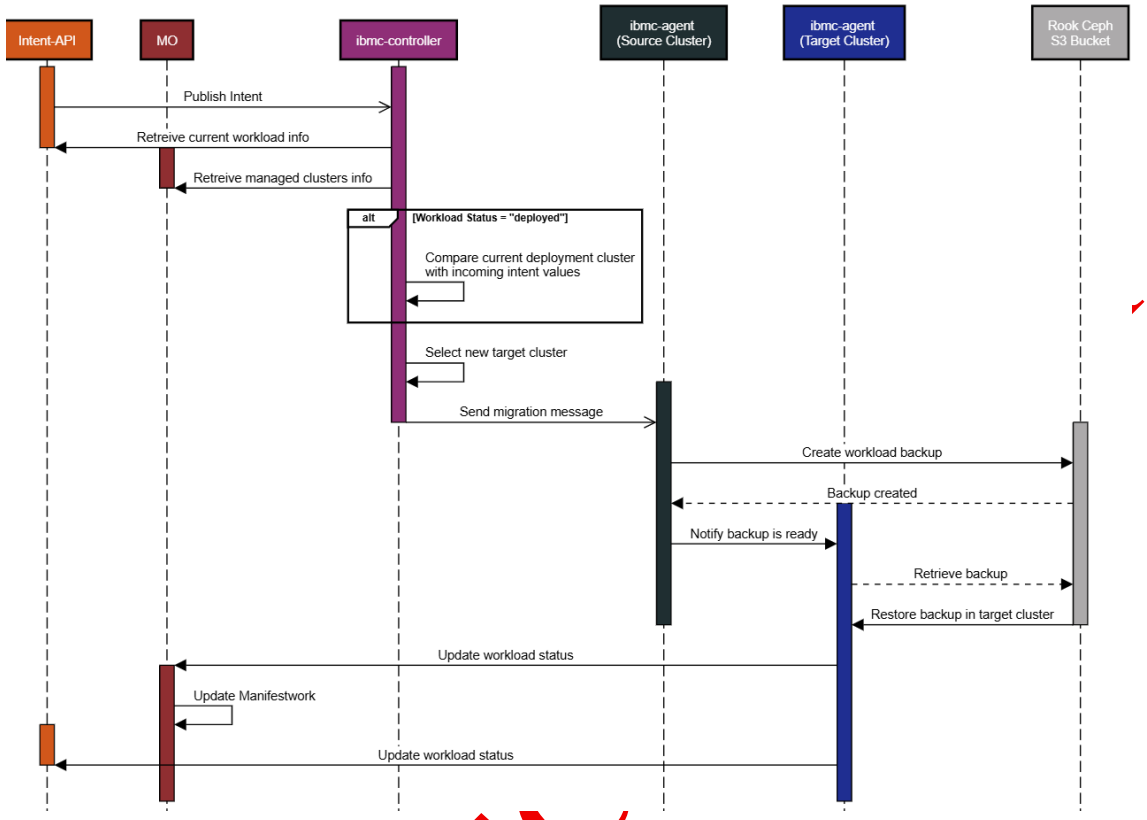


Figure 17. Migration Sequence Diagram

1. The Intent-API publishes in the 'notify' Rabbitmq queue an Intent with one or more requirements (Availability, GreenEnergy, CPU, Memory etc).
2. The ibmc-controller retrieves workload status from the Intent-API.
3. The ibmc-controller retrieves from the MO the information related to each of the managed clusters.
4. If the workload is already deployed in any cluster and the intent requirements are not met, then a migration action is triggered.
5. A new target cluster that complies with the intent requirements is selected from the available Managed Clusters list.
6. The ibmc-controller sends a message via RabbitMQ queue to the ibmc-agent containing the workload ID and the selected target cluster.
7. A backup is created with all of the workload associated resources and it is uploaded to the Rook Ceph S3 Bucket located in the HUB cluster.
8. When the backup is completed, a message is sent to the target cluster's ibmc-agent to continue with the migration process.
9. The backup is restored in the target cluster, creating all the workload resources.
10. A message is sent to the MO notifying about the workload migration completion.
11. The MO updates the corresponding Manifestwork so that it matches the new workload deployment status.
12. A similar message is sent to the Intent-API updating the workload status, specifying the cluster where it has been deployed.

4.1.3.1 Meta-Orchestrator (MO)

The IBMC integrates with the MO primarily to gather cluster-related data and coordinate migration execution. The ibmc-controller queries the MO to retrieve up-to-date information about all managed clusters, which it uses to evaluate compliance with intent requirements. After a migration is completed, the ibmc-agent notifies the MO, which then updates the ManifestWork to reflect the workload's new deployment state.

4.1.3.2 Intent-Based API

The Intent-Based API maintains all information related to deployed workloads and is responsible for creating and managing the intents that initiate workload migrations. When a migration is triggered, the intent's requirements are used to identify the new target cluster, which must meet all specified criteria. Upon completion of the migration, the ibmc-agent sends a notification to the Intent-API, updating the workload's status, including the new cluster where it has been deployed. This ensures that the Intent-API remains synchronized with the latest workload deployment state.

4.1.3.3 MNCC

The notify queue to which the ibmc-controller listens, is also used for the L2S-M network creation process which also relies on Intents. The creation of these networks, like workload migrations, is triggered by specific intent requirements. For more details, refer to Section 5.2.

4.1.3.4 CFDRL

The CFDRL uses machine learning and relies on trained models to detect the sub-optimal workload operation, leading to the triggering of the workload migration. When this occurs, the CFDRL sends the required information to the IBMC, in order perform the migration.

4.1.4 Conclusion

The IBMC, has completed its functionality implementation and has been successfully integrated with the other components, including MO, Intent-Based API, mNCC, and CFDRL. This integration enables the IBMC to support seamless workload deployment and migration across the IoT-to-Edge-to-Cloud continuum, maintaining operational efficiency within the dynamic meta-OS environment.

The functionality of the Intent-Based Migration Controller (IBMC) has been significantly extended to support intelligent redeployment decisions driven by dynamic, user-defined intents. This enhancement enables the IBMC to continuously monitor the operational environment and autonomously evaluate incoming intents, which encapsulate high-level objectives such as performance optimization, energy efficiency or latency reduction requirements.

The addition of this intent-driven redeployment logic marks a shift toward a more proactive and adaptive control mechanism within the meta-OS framework, ensuring that service delivery remains aligned with evolving user needs and operational constraints.

The IBMC is deployed across the NEMO pilots' infrastructures, where it will be validated through pilot-specific use cases to demonstrate the IBMC's potential to enhance service continuity and resource efficiency in complex, distributed computing environments.

4.2 Plugin & Applications Lifecycle Manager

The Plugin & Applications Lifecycle Manager (LCM) is a flexible and unified solution for managing plugins and applications across the NEMO ecosystem. Acting as the interface between NEMO users and the system, the LCM enables on-demand deployment of workloads (services, applications, and plugins) within the NEMO environment.

While workloads operate on the NEMO meta-OS, an event-driven system continuously monitors key performance metrics. Simultaneously, a dedicated security controller tracks security-related events, alerts users to anomalies, and takes proactive steps to mitigate potential cyber threats.

The LCM's user interface is designed to integrate seamlessly with other core NEMO components - such as the Intent-based API, PPEF, MOCA, and CMDT - delivering unified and intuitive user experience. It provides access to features including user profile management, workload deployment and monitoring, security oversight, and historical data analysis, all tailored to specific user roles of NEMO ecosystem.

4.2.1 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 final LCM high-level architecture of NEMO meta-OS is depicted in the development view diagram in Figure 18. In brief, the software components that make up the LCM module are the following.

LCM CD is based on ARGO CD framework [17] [18] [19] 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.

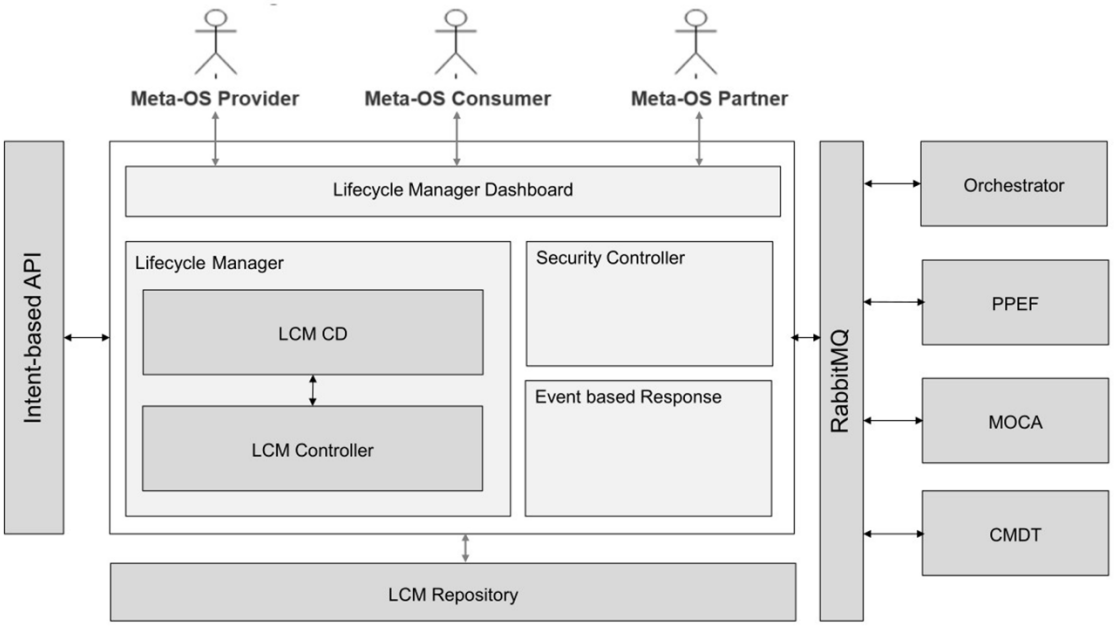


Figure 18. LCM high-level architecture

4.2.1.1 LCM Repository

LCM Repository uses Elasticsearch[12] 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.

4.2.1.2 LCM Visualization

LCM Visualization serves as the primary interface of the NEMO ecosystem, offering tailored tools for each user role to manage workloads and resources within the NEMO meta-OS and monitor activities. Its goal is to deliver an intuitive and seamless experience across the NEMO ecosystem, catering to both advanced users and those with less technical expertise. The interface presents relevant information about the workload lifecycle, usage, and security of workloads and resources in a clear, concise format, with multiple levels of detail (e.g., workload, resource, user, and system views). Detailed explanation of the LCM views were provided in section 2.2, as part of the 3rd party documentation and are also described extensively in the context of section 5.

4.2.2 Conclusion

The LCM component has been successfully integrated into the NEMO ecosystem, deployed in OneLab environment, providing features that include:

- Workload lifecycle management and monitoring
- Resources provisioning and usage monitoring
- Security/vulnerability scanning and monitoring
- Visual interface for experienced and non-experienced users

The final result offers seamless workload deployment and monitoring through an intuitive interface.

4.3 Monetization and Consensus-based Accountability

4.3.1 Overview

The Monetization and Consensus-based Accountability (MOCA) component lies at the heart of NEMO's pre-commercial ecosystem, providing a secure, transparent, and equitable framework that enables both resource providers (offering compute, storage, network, or data) and service consumers to seamlessly earn and spend "credits" across the AIoT–Edge–Cloud continuum. As the backbone of NEMO's Resource-as-a-Service (RaaS) marketplace, MOCA guarantees timely, accurate reward settlement for providers and precise billing for consumers, all underpinned by a tamper-proof audit trail maintained on a distributed ledger.

Since the previous deliverable, D4.2, significant progress has been made to expand MOCA's capabilities and refine its functionalities. Key advancements include:

- **Smart Contract Deployment Tool**: Implementation of a dedicated tool to streamline the deployment and management of smart contracts utilized by MOCA.
- **ML-as-a-Service (MLaaS) Smart Contract**: Development and deployment of a specific smart contract tailored for the accounting and monetization of Machine Learning services offered and consumed via the NEMO platform.
- **Enhanced Reward Mechanisms**: Introduction of more sophisticated smart contract functionalities to accurately reward service providers based on various contribution factors.

[12] https://www.elastic.co/elasticsearch

- **Network Resource and Data Monetization**: Extension of the MOCA framework to encompass the accounting and potential monetization of network resources and data assets within the NEMO ecosystem.

These enhancements aim to increase MOCA's flexibility, broaden the scope of monetizable resources and services, and support more complex business models emerging within the NEMO continuum, further solidifying its role in facilitating a trustworthy Resource-as-a-Service (RaaS) marketplace.

4.3.2 Architecture

MOCA's architecture has been described in detail in D4.2. In the current document, there is one notable addition to the component's architecture, namely the Smart Contract Deployment Tool. In Figure 19 the updated architecture diagram for MOCA is depicted. The Smart Contract Deployment Tool is responsible for allowing the users to upload their own contracts to the Smart Contracts component and allow them to enhance the tool with new accounting logic.

Regarding the workflow presented in the deliverable D4.2, it remains the same without any major changes. The component can be accessed through the Event Server's REST API, to comply with the original component architecture, where the Event Server acts as the main communication interface with the rest of the MOCA sub-components.

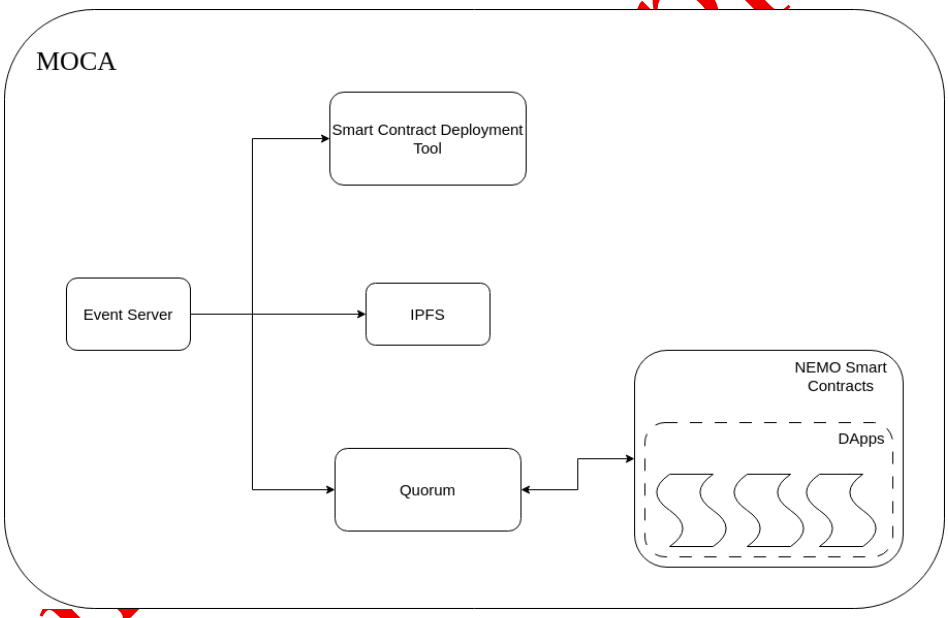


Figure 19. MOCA architecture

4.3.3 Interaction with other NEMO components

The interactions of the MOCA component with other NEMO components have remained the same as presented in deliverable D4.2. However, for consistency, in Figure 20 an updated version of the integration diagram is provided including the Smart Contract Deployment Tool subcomponent.

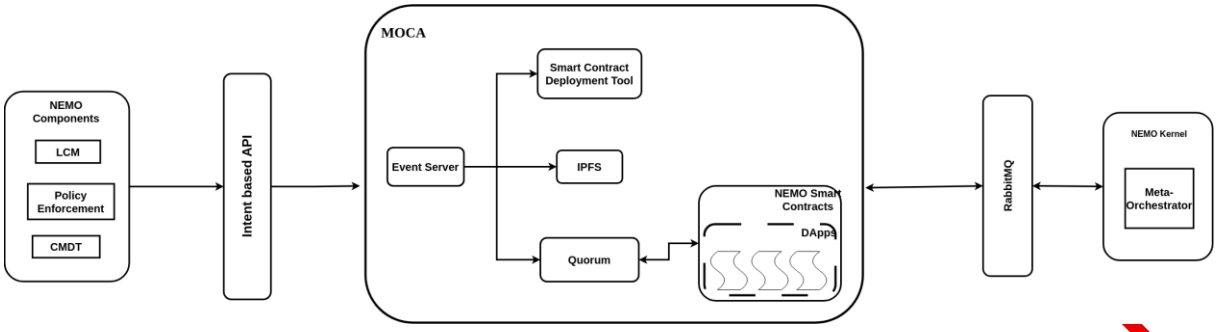


Figure 20. MOCA integration diagram

4.3.4 Results

In deliverable D4.2 the initial set of MOCA's features and capabilities were described; in this document the vastly expanded accounting capabilities of the component are presented. Initially, MOCA utilized a set of smart contracts to perform the necessary calculations of the rewards and costs in the NEMO continuum, namely (a) NemoTokenEstimation, (b) NemoFunds, (c) InfrastructureOwnerModel and (d) ServiceProviderModel. In addition to these smart contracts, MOCA was enriched with supplementary calculation methods that enhanced its versatility, such as:

- Calculate the costs of network usage of a workload
- Calculate the costs of the Machine Learning workloads
- Allow the users to upload a dataset and reward them
- Apply cost deductions for workloads that run on energy efficient clusters

4.3.4.1 Network Resource Monetization

To calculate the network usage costs, a new smart contract, the NetworkUsageModel, Listing 1, has been created to make the necessary cost calculations for a workload. It is worth noting that only the workloads that are exposed through NEMO, with the usage of the ingress attribute, are subject to this cost deduction. As an incentive, when the workloads with the ingress attribute enabled are registered, MOCA rewards them with 5 additional tokens, apart from the initial 5 tokens for the registration. Every workload which is deployed as an external service is given 5GBs of free network usage before they start getting charged. Above this threshold, the workload is charged per GB consumed. The network usage of a workload is calculated periodically by MOCA and it is defined as the sum of the transmitted and received data of the workload.

NetworkUsageModel.sol

```
pragma solidity ^0.8.0;
import "./NemoFunds.sol";
contract NetworkUsageModel {
    NemoFunds public nemoFunds;
    uint256 constant MAX_FREE_NETWORK_USAGE = 500000000;
    uint256 constant CHARGE_PER_GB = 10000000;
    struct NetworkMetrics {
        string serviceId;
        string clusterId;
        uint256 networkUsage; // in GB
    }
    event NetworkComputeTokens(
        string serviceId,
        string clusterId,
        uint256 networkUsage,
```

```
        uint256 tokens
    );
    constructor(address _nemoFundsAddress) {
        nemoFunds = NemoFunds(_nemoFundsAddress);
    }
    function computeCredits(NetworkMetrics memory _metrics) public {
        require(
            nemoFunds.isCustomerRegistered(_metrics.serviceId),
            "The service is not registered!"
        );
        uint256 _networkUsage = _metrics.networkUsage;
        uint256 _tokens = 0;
        if (_networkUsage > MAX_FREE_NETWORK_USAGE) {
            _tokens =
                ((_networkUsage - MAX_FREE_NETWORK_USAGE) * CHARGE_PER_GB) /
                10 ** 8;
        }
        nemoFunds.makeTransaction(
            _metrics.serviceId,
            _metrics.clusterId,
            _tokens
        );
        emit NetworkComputeTokens(
            _metrics.serviceId,
            _metrics.clusterId,
            _networkUsage,
            _tokens
        );
    }
}
```

Listing 1. The NetworkUsageModel smart contract details

4.3.4.2 ML-As-A-Service (MLaaS)

NEMO supports the deployment of Machine Learning models, which usually require GPU power to perform efficiently processing intensive processes such as the training of a large model. In this context, it was vital to offer a smart contract dedicated to the use of GPU resources, MLModel, Listing 2. The NEMO continuum offers GPUs with time slicing enabled, which allows for multiple jobs to be executed in parallel simultaneously. Therefore, to accurately calculate the usage cost for a Machine Learning workload, it was important to consider, except for the GPU percentage that has utilized, also the time it had reserved that particular resource. Another crucial aspect that was used for the MLaaS was the deployment cost of such infrastructure based on socioeconomic factors.

MachineLearningModel.sol

```
pragma solidity ^0.8.0;
import "./NemoTokenEstimationSetupContract.sol";
import "./NemoFunds.sol";
contract MachineLearningModel {
    NemoTokenEstimationSetupContract public nemoTokenEstimationSetup;
    NemoFunds public nemoFunds;
    address public owner;
    uint256 public gpuRate;
    // Number of seconds per hour
    uint256 constant SECONDS_PER_HOUR = 360000000000;
    struct MachineLearningMetrics {
        string serviceId;
        string clusterId;
```

```
    string region;
    uint256 gpuUtil;
    uint256 allocatedTime;
    uint256 usageFraction;
  }
  event MachineLearningComputeTokens(
    string serviceId,
    string clusterId,
    string region,
    uint256 gpuUtil,
    uint256 tokens
  );
  constructor(
    address _nemoTokenEstimationSetupContractAddress,
    address _nemoFundsAddress
  ) {
    nemoTokenEstimationSetup = NemoTokenEstimationSetupContract(
      _nemoTokenEstimationSetupContractAddress
    );
    nemoFunds = NemoFunds(_nemoFundsAddress);
  }
  modifier checkRegionData(string memory region) {
    require(
      nemoTokenEstimationSetup.isRegionSet(region),
      "Data for region must be set before calling this function."
    );
    _;
  }
  function computeCredits(MachineLearningMetrics memory _metrics) public {
    (uint256 _regionalGpuCosts) = nemoTokenEstimationSetup.getRegionGpuInfo(
      _metrics.region
    );
    string memory _serviceId = _metrics.serviceId;
    string memory _clusterId = _metrics.clusterId;
    uint256 _tokens = 0;
    _tokens =
      ((_metrics.allocatedTime * _metrics.usageFraction) /
        SECONDS_PER_HOUR) +
      _regionalGpuCosts;
    nemoFunds.makeTransaction(_serviceId, _clusterId, _tokens);
    emit MachineLearningComputeTokens(
      _metrics.serviceId,
      _metrics.clusterId,
      _metrics.region,
      _metrics.gpuUtil,
      _tokens
    );
  }
```

```
}
```

Listing 2. The MachineLearningModel smart contract details

4.3.4.3 Data Monetization

One of the rapidly increasing aspects of the AI/ML domain is dataset sharing. In this context, MOCA provides a monetization mechanism of data for NEMO users. MOCA's Event Server allows the upload of datasets for 3rd party users to utilize them on their own applications. In return, the dataset provider is rewarded with 10 tokens. The datasets are stored in an internal S3 bucket that is accessible through MOCA's API. In Figure 21 the endpoint related to the dataset's uploading is presented along with the payload fields required, while on Listing 3 the details of the related smart contract are shown.

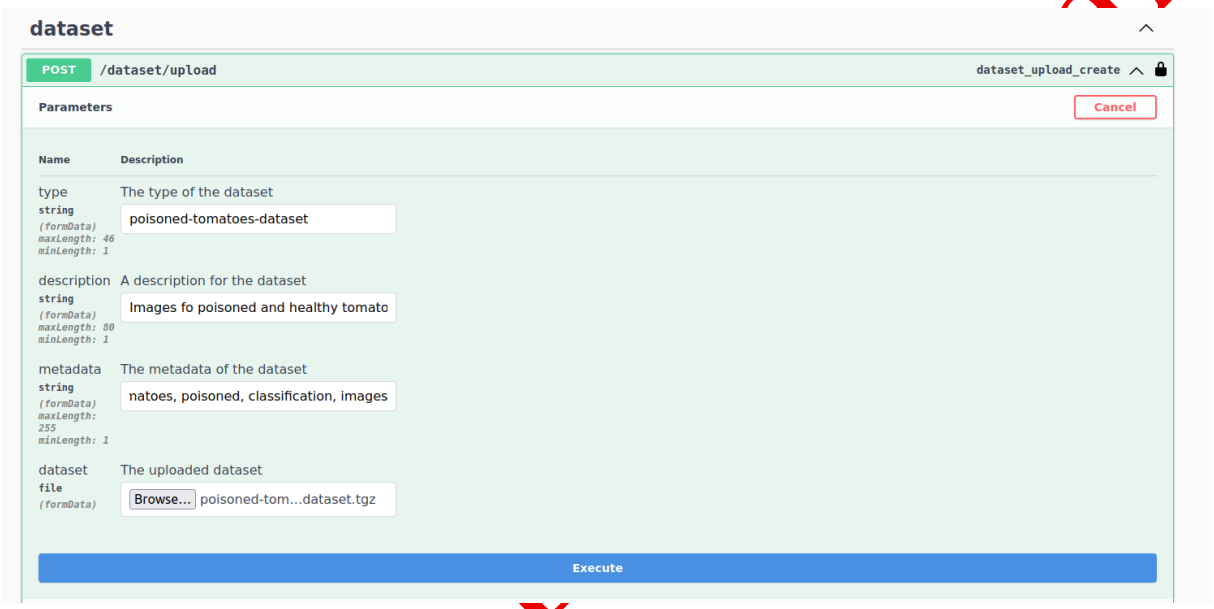


Figure 21. Example of deploying dataset

DataModel.sol

```
pragma solidity ^0.8.0;

contract DataModel {
    struct DataInfo {
        string dataId;
        string dataType;
        string description;
        string metadata;
        string endpoint;
    }

    event DataComputeTokens(string dataId, uint256 tokens);

    function computeCredits(DataInfo memory _info) public {
        uint256 _tokens = 0;

        _tokens = 1000000000;

        emit DataComputeTokens(_info.dataId, _tokens);
    }
}
```

Listing 3. The DataModel smart contract details

4.3.4.4 Green Energy rewards

In this version of MOCA, we have also incorporated a rewarding mechanism for the workloads that run on "green" clusters. A workload that runs on a green cluster will receive a reduction to their resource usage costs, depending on the percentage of green energy supported by the infrastructure. This method aims to give incentive to the workload owners to prefer green deployment clusters, so that:

1. They reduce their workload costs and
2. Indirectly, they help with making the NEMO continuum more energy efficient.

The *ServiceProviderModel* smart contract was modified appropriately to consider the green energy percentage of the deployment platform. Annex 3 – ServiceProviderModel provides the details of the contract.

4.3.4.5 Smart Contracts Deployment Tool

Another major addition to the MOCA component is the **Smart Contracts Deployment Tool**. This sub-component allows the users to upload their own smart contracts in the private Quorum blockchain used for the contract deployment. In this way, the users can interact easily with the blockchain, customize and test MOCA with their own smart contracts, implementing different approaches in the accounting logic. Figure 22 demonstrates a simple example of the body of the request to upload the contract. The user only needs to provide their smart contract.

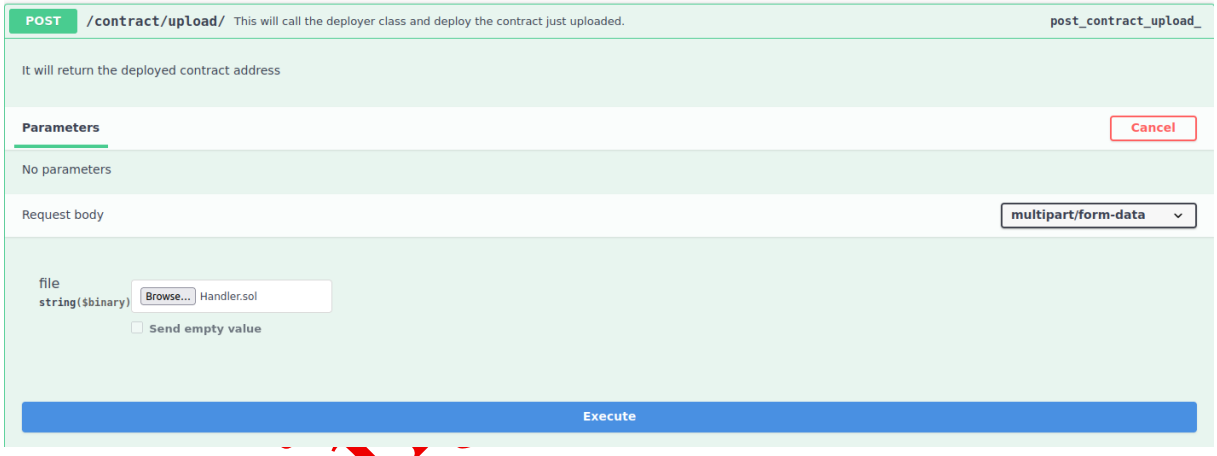


Figure 22. Example of contract deployment

4.3.5 MOCA Business Models

The MOCA component, as mentioned earlier in this document, is not a business model by itself but rather a foundational enabler for a variety of business models within the NEMO ecosystem. It establishes a transparent, automated, and fair marketplace for resources and services across the AIoT-Edge-Cloud continuum. The core of MOCA's design facilitates a "credits" or "tokens" based economy, where contributions are rewarded and consumption is billed, all managed via smart contracts on a distributed ledger.

The primary business models MOCA enables can be viewed from the perspective of different stakeholders:

4.3.5.1 Resource-as-a-Service (RaaS) for Providers:

- **Computing Resource Monetization**: Infrastructure owners (as per *InfrastructureOwnerModel*) can offer their computing resources (CPU, RAM) and get rewarded based on usage by service providers (as per *ServiceProviderModel*). MOCA's smart contracts ensure accurate accounting and reward distribution.

- **Network Resource Monetization:** Providers offering network access can monetize this, especially for workloads requiring external exposure (as per *NetworkUsageModel*). The model incentivizes initial free usage, then charges per GB, creating a clear value proposition.
- **Specialized Hardware Monetization (e.g., GPUs for MLaaS):** Owners of specialized hardware like GPUs can offer these for Machine Learning (ML) workloads (as per *MachineLearningModel*). MOCA allows for nuanced billing based on GPU utilization, allocated time, and even regional cost differences, enabling a viable ML-as-a-Service (MLaaS) model.
- **Data Monetization:** Entities possessing valuable datasets can make them available on the NEMO platform (as per *DataModel*). They are rewarded with tokens when their data is accessed or utilized, creating a marketplace for data itself.
- **Incentivizing Green Energy**: Resource providers utilizing green energy sources can offer their infrastructure at potentially premium rates or benefit from NEMO's reward mechanisms (as seen in the *ServiceProviderModel* modifications for green energy rewards), encouraging sustainable practices and creating a niche market.

4.3.5.2 Pay-As-You-Go (PAYG) & On-Demand Services for Consumers:

- **Flexible Resource Consumption:** Service developers and end-users can consume various resources (compute, network, ML processing, data) on-demand without significant upfront investment. They are billed transparently through MOCA based on their actual usage.
- **Access to Specialized Services:** Consumers can readily access and pay for specialized services like MLaaS or specific datasets, fostering innovation and reducing barriers to entry for developing complex applications.
- **Cost Optimization:** Incentives for using green clusters, along with tiered pricing models (e.g., free initial network usage), enable consumers to reduce and optimize their operational costs.

4.3.5.3 Custom Business Logic via Smart Contract Deployment:

- **Extensible Monetization Schemes:** The Smart Contract Deployment Tool (Section 4.3.4.5) is a crucial enabler for novel business models. It allows NEMO users (providers or even third-party integrators) to deploy their own smart contracts with custom accounting and monetization logic. This opens the door for:
  - Subscription-based models.
  - Revenue-sharing agreements.
  - Usage-tier based pricing is not natively covered by default MOCA contracts.
  - Loyalty programs or bundled service offerings.
- **Fostering a Broader Ecosystem**: This flexibility allows the NEMO marketplace to adapt and evolve, accommodating new types of resources or service delivery models as they emerge, driven by the community or specific enterprise needs.

In essence, MOCA aims to create a vibrant, self-sustaining economic ecosystem. It empowers resource providers to generate value from their assets and service consumers to access these assets efficiently and transparently. The extensibility offered by custom smart contract deployment ensures that the business models operating within NEMO can evolve and diversify over time. The distributed ledger technology underpinnings provide the trust and auditability necessary for such an ecosystem to thrive.

4.4 Intent-based SDK/API

4.4.1 Overview

Intent-based SDK/API (IBA) is the interface of NEMO to third party interactions, with the finetuning of this component being continuous to incorporate new features as the project progresses. To this end, several activities towards the finalization of the Intent-based SDK/API were carried out since the delivery of deliverable D4.2. These activities can be divided into two categories, a) the integration related activities, and b) core optimization activities that enhance the existing functionalities of the tool.

4.4.2 Architecture

The Intent Based API was developed in a microservices oriented architecture aiming to a high-availability and highly scalable system. The current architecture, Figure 23, demonstrates the IBA subcomponents as well as their interaction with other NEMO components. This allows a fast and easy integration of new components without major modifications in the entire IBA system, just only in the specific subcomponent that interacts with the newly introduced NEMO component. An example of this architecture will be presented in the subsequent subsections with the integration of mNCC to IBA.

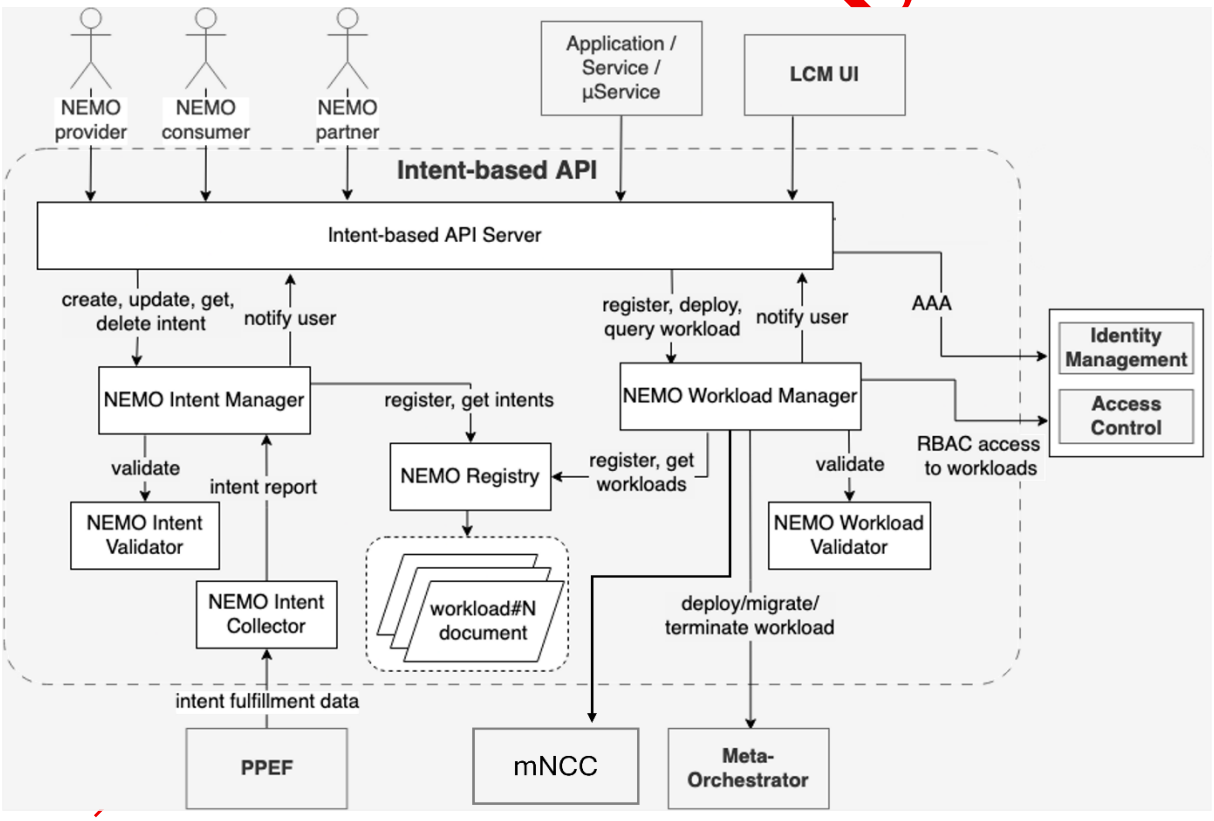


Figure 23. Final Architecture diagram of Intent-Based API

4.4.3 Interaction with other NEMO components

The activities related to the Intent-based SDK/API in the 2nd integration phase of the project are mainly focused on the integration between IBA and other NEMO components. As described in the Sub-section 4.4.2, Intent-based SDK/API is located at the front-frontier of the NEMO framework, meaning that it is the main input of the information used by the NEMO components. In this context, IBA is participating

in many integration pipelines which are described in Section 5. The related work carried out involved the integration with Meta-Orchestrator's components, such as the IBMC and the deployment/undeployment components. In this integration pipeline, new message exchanging queues were created, while the existing ones were updated to house the needs of the new features presented by the Meta-Orchestrator as well as to provide the maximum optimization for the deployment of workloads. In addition, several integration actions were performed to incorporate IBA with NEMO Access Control component. This was of paramount importance as it allows the use of IBA from external parties that are located outside of the NEMO ecosystem, e.g., a service in an external cluster. In terms of workload management, several modifications were made regarding the IBA-CMDT interactions. More specifically, the work carried out was focused on the integration of Linkerd[13] into the features of IBA to be able to handle all the states of a workload, such as creation, deployment, running status, and termination by adding the necessary annotations to the generated manifest file, Listing 4. Finally, is the integration of IBA with the networking components (mNCC & L2S-M) of NEMO. For this integration pipeline, severe modifications were required at the IBA. A new endpoint was created to handle the new Intent request along with the related Intent, Annex 1, and its validator. As for the communication pipelines for inter component communication, several RabbitMQ channels were created to allow the communication and state management of the network slices from the IBA, Listing 5.

```
def set_linkerd_injection(manifests: List[Dict[str, Any]]):
    for manifest in manifests:
        if manifest['kind'] in ['Deployment', 'StatefulSet', 'DaemonSet']:
            annotations = manifest.get('spec', {}).get('template', {}).get('metadata', {}).get('annotations', {})
            annotations['linkerd.io/inject'] = 'enabled'
        elif manifest['kind'] in ['Pod']:
            if 'metadata' not in manifest.keys():
                manifest['metadata'] = {}
            if 'annotations' not in manifest['metadata'].keys():
                manifest['metadata']['annotations'] = {}
            manifest['metadata']['annotations']['linkerd.io/inject'] = 'enabled'

    return manifests
```

Listing 4. Function used for annotating Manifests

```
nemo.rabbitmq INFO 2025-03-27 14:53:59,245 rabbitmq [x] Sent to mncc, routing_key=mncc.ibs, message:
{"workload": "61edb162-1d52-4e4d-8b55-275187a780ba", "pod": {"labels": {"l2sm": true, "l2sm/app": "[parameter]"},
"annotations": {"l2sm/networks": "spain-network"}, "env": {"DNS_NAME": "[parameter].spain-network.inter.l2sm"}}}
```

Listing 5. RabbitMQ communication channel with mNCC and sample payload

On the other hand, apart from the integration related development of IBA, several developments for enhancing both the User and the intra-service interfaces were done. Firstly, a lifecycle event monitoring subcomponent was created to better handle the state of the workloads, Annex 2. This was a crucial feature as it allows the end-user to have knowledge of each state of a workload that is passed through the Graphic User Interface. Then, the user can understand if his workload has been initiated, running or terminated. In case an error occurs, the user can retrieve the information related to that failure in a user-friendly way. Another enhancement that took place in the 2nd integration phase was related to the MOCA component. As the service's needs changed some of the MOCA's endpoints got deprecated and later

[13] https://linkerd.io/

removed from the IBA-MOCA integration pipeline as this was reported in deliverable D4.2. In addition, as the project progresses and based on the iterative Quality Assurance process employed in NEMO as well as the integration process a minor number of bugs along with service improvements were identified. To tackle these newly identified needs, IBA enhanced its existing toolset by presenting new endpoints for housing the needs of a modern microservice. Moreover, the need for specific information in Intents needed for the evaluation and deployment of a workload, lead to the creation of use case specific Intents, such as the Machine Learning Intent, Listing 6, that forces the workload to be deployed in a GPU compatible cluster node.

```
Intent:
  Id: 'Intent_6'
  userLabel: 'MachineLearning'
  intentExpectations:
   - expectationId: '4'
     expectationVerb: 'DELIVER'
     expectationObject:
        objectInstance: '781052a5-4270-4113-847a-8730cdf55ba7'
        objectType: 'NEMO_WORKLOAD'
     expectationTargets:
      # How much VRAM is needed for the workload to run (in GB) -> will be evaluated by checking the available free VRAM
for the GPU
       - targetName: 'vram'
         targetCondition: 'IS_GREATER_THAN'
         targetValueRange: '10'
  intentPriority: 1
  observationPeriod: 60
  intentAdminState: 'ACTIVATED'
```

Listing 6. MachineLearning Intent sample

4.4.4 Conclusion

The Intent Based API was designed to be the interface between the Graphical User Interface or the 3rd party User and the NEMO components. As mentioned in the previous subsections, IBA plays a pivotal role within the NEMO architecture as it orchestrates the communication channels and is the initial point for many crucial NEMO workload pipelines. Since the delivery of D4.2, IBA was enhanced with functionalities introduced by other components as well as internal optimizations that guarantee the efficient operation of NEMO. The workflow pipelines where the IBA participated in are described in Section 5 of this document.

5 NEMO scenario-driven verification & results

In this section the activities and tests performed towards a complete and integrated NEMO infrastructure are presented. Apart from the WP4 tools' integration activities, as presented in D4.2, for completeness in this document also the integration pipelines related to WP2 and WP3 tools are presented. This section aims to provide in a complete and coherent way the information regarding the tools' interactions within NEMO. As the NEMO framework consists of many tools one of the first critical issues was the testing procedure. To solve this issue, a categorization of the tools took place based on sub-system functionalities was selected. Through this system breakdown dedicated pipelines created focusing on small and dedicated functions of the System.

Each scenario, as outlined in the subsequent subsections, focuses on a distinct operational aspect of the NEMO meta-OS. The results presented herein provide empirical evidence of the platform's capabilities and its readiness to support the diverse use cases envisioned within the NEMO project, highlighting both successful integrations and areas for further refinement. The scenarios cover critical functionalities including cluster registration, workload registration and provisioning, workload scheduling and orchestration, workload lifecycle management, and integration activities stemming from WP3, such as the Secure Execution Environment.

These dedicated integration pipelines provided several benefits in the integration procedure as the integration, testing and optimization of the components based on specific functionalities made the entire process easier and more robust, as well as aligning with the best practices of Software development principles.

5.1 NEMO Cluster registration – resource provisioning

This section details the NEMO cluster registration scenario related activities undertaken. The integration results as reflected in the cluster registration sequence diagram described in Figure 24 are presented below.

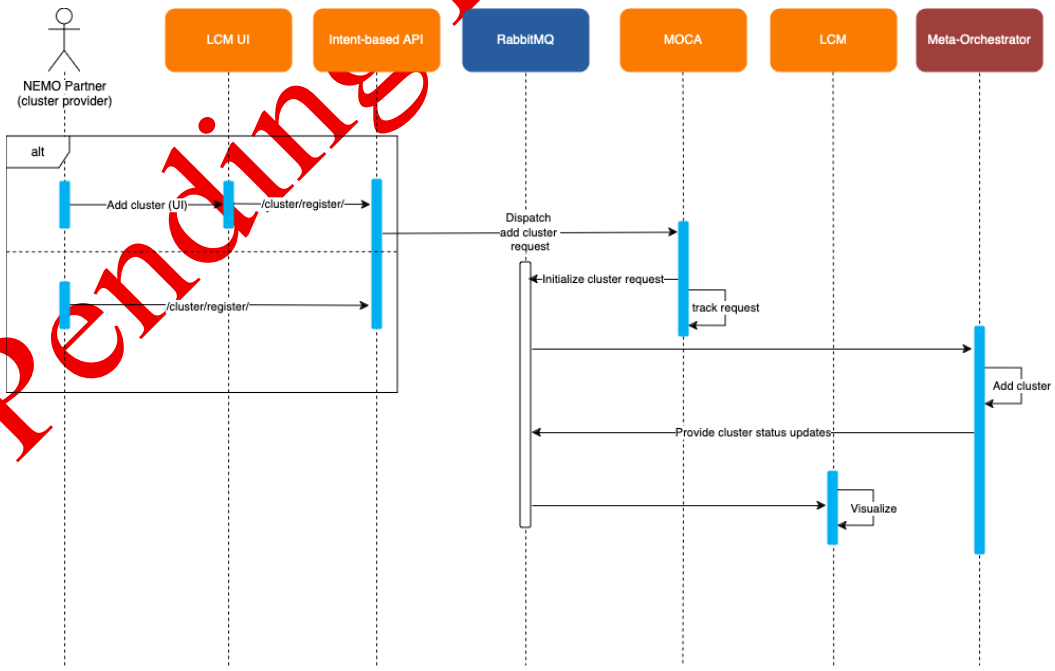


Figure 24. Cluster registration sequence diagram

5.1.1 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 message communication to MO 3. Cluster addition process by MO 4. Cluster status provisioning to RabbitMQ 5. Cluster status update visualization in LCM UI

5.1.2 Results

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

5.1.2.1 Cluster registration through the LCM UI

To register a cluster through the LCM UI, first, we go through the LCM UI home page to the "Resource Provisioning" tab, Figure 25.

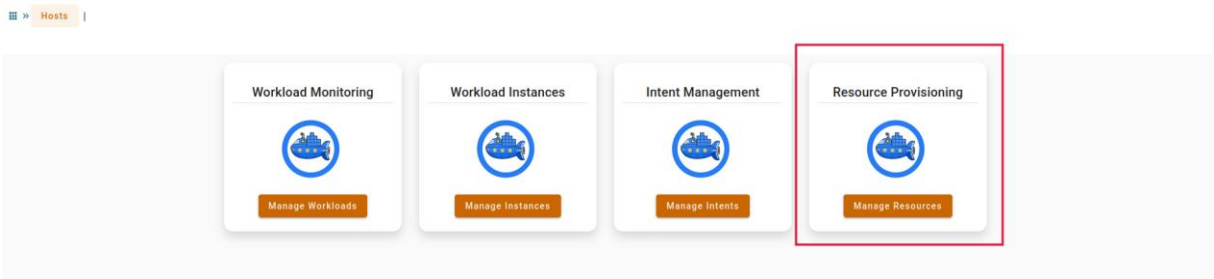


Figure 25. LCM home page

In the "Resource Provisioning" page, we have an overview of some basic details for the clusters, like their ID, name, resources (CPU, RAM, disk, GPU), their deployment status and the endpoint they are available at, as it is shown in Figure 26.

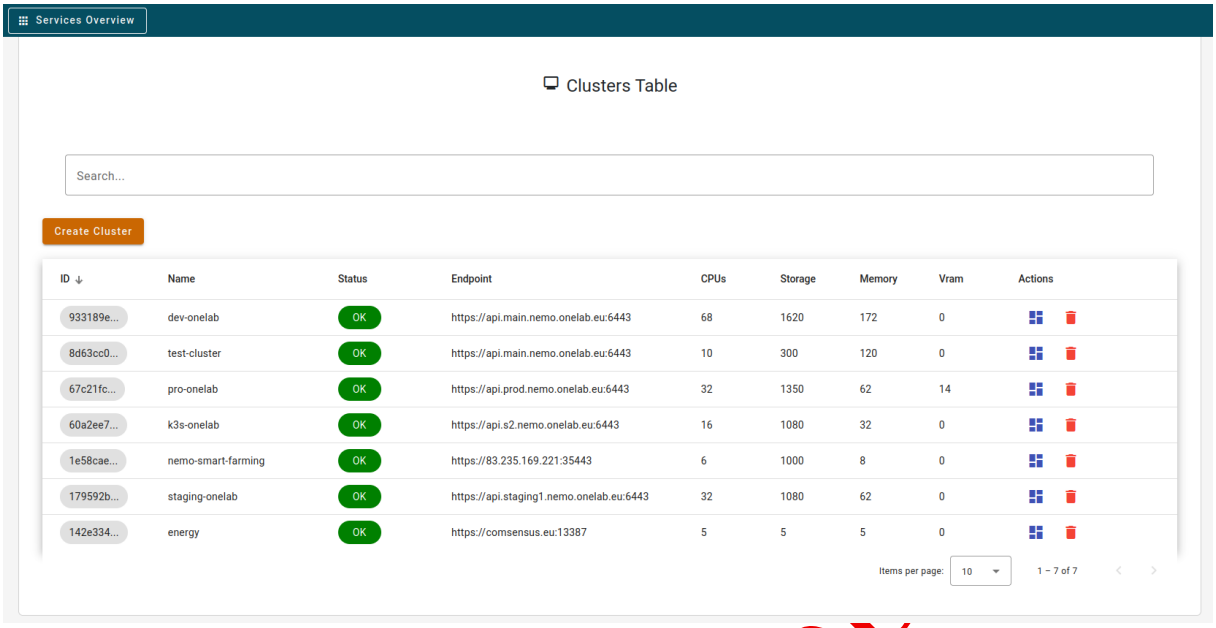


Figure 26. Clusters overview page

When a user registers a cluster, they are prompted to fill in the form shown in Figure 27. The form requires of the user to upload the configuration file of their cluster and fill in information like the name of the cluster, its endpoint, the resources of the cluster, the availability percentage, its green energy percentage and possible costs they want to apply to the workload that will be deployed on their premises.

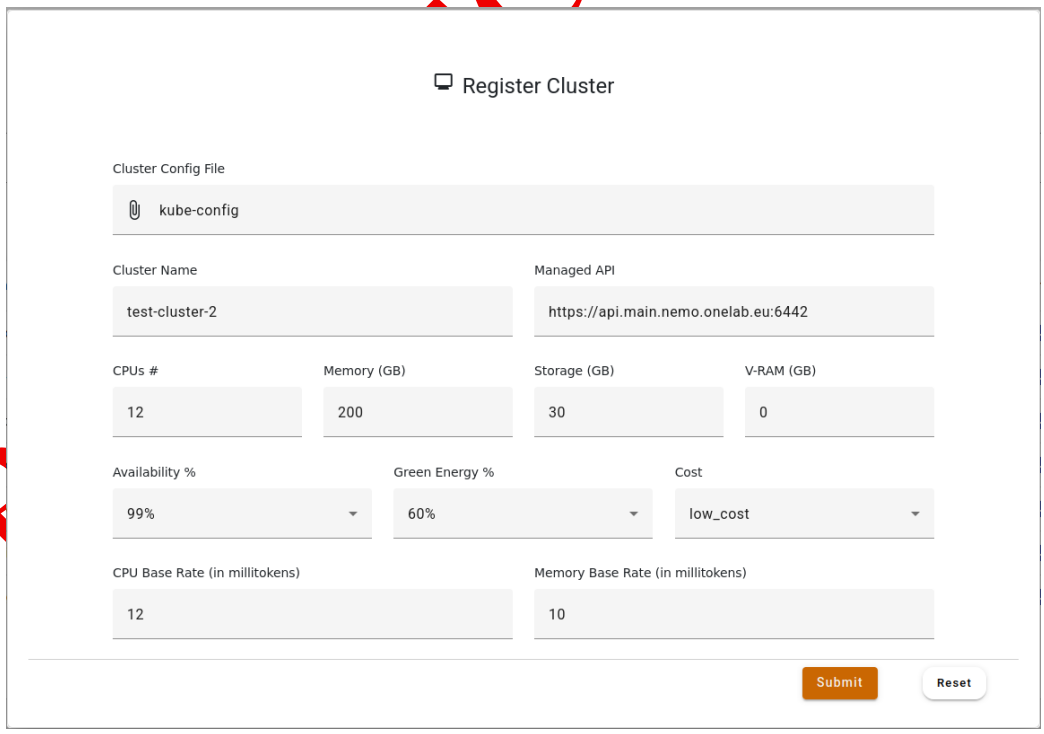
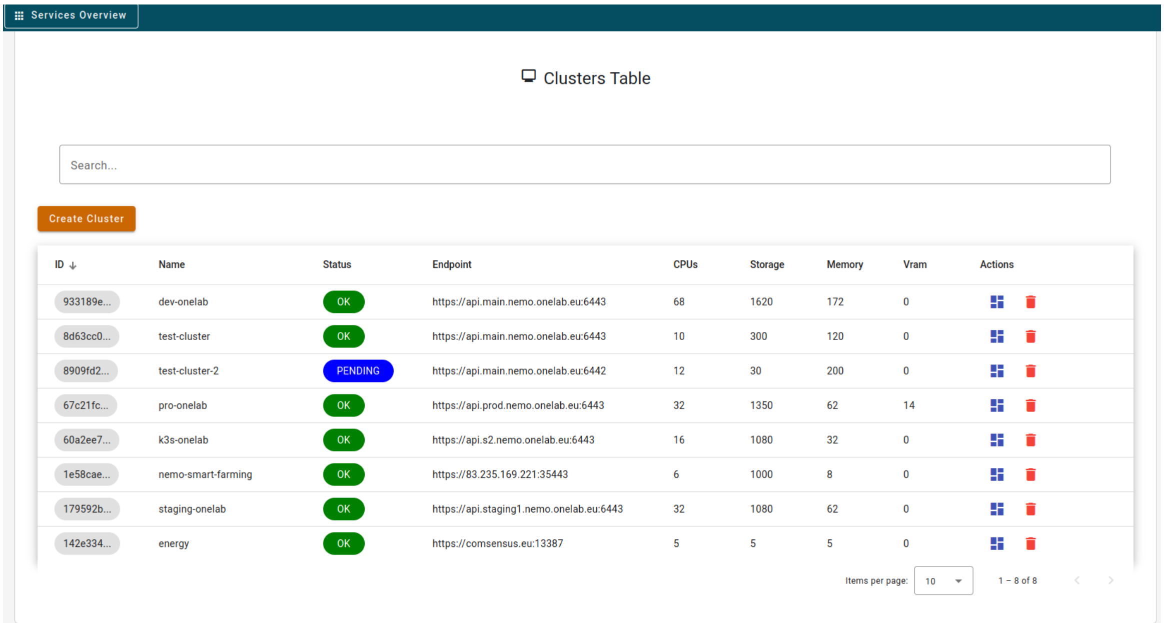


Figure 27. Cluster registration form

5.1.2.2 Cluster registration message communication to MO

Once the submission form is completed correctly, LCM sends to NEMO the registration request (the request is proxied through the Intent API). MOCA receives the request informs the Meta Orchestrator of request, in order to join the cluster with the NEMO continuum. Until the join process is completed, the cluster is saved with state "Pending", Figure 28.



Services Overview

Clusters Table

Search...

Create Cluster

ID	Name	Status	Endpoint	CPUs	Storage	Memory	Vram	Actions
933189e...	dev-onelab	OK	https://api.main.nemo.onelab.eu:6443	68	1620	172	0	
8d63cc0...	test-cluster	OK	https://api.main.nemo.onelab.eu:6443	10	300	120	0	
8909fd2...	test-cluster-2	PENDING	https://api.main.nemo.onelab.eu:6442	12	30	200	0	
67c21fc...	pro-onelab	OK	https://api.prod.nemo.onelab.eu:6443	32	1350	62	14	
60a2ee7...	k3s-onelab	OK	https://api.s2.nemo.onelab.eu:6443	16	1080	32	0	
1e58cae...	nemo-smart-farming	OK	https://83.235.169.221:35443	6	1000	8	0	
179592b...	staging-onelab	OK	https://api.staging1.nemo.onelab.eu:6443	32	1080	62	0	
142e334...	energy	OK	https://comsensus.eu:13387	5	5	5	0	

Items per page: 10 1 – 8 of 8

Figure 28. Cluster pending status

5.1.2.3 Cluster addition process by MO

Figure 29 shows the message from MOCA the Meta Orchestrator received. The message includes all the information the user filled in in the registration form.

Figure 29. MOCA cluster registration message through RabbitMQ

5.1.2.4 Cluster status provisioning to RabbitMQ

Once the Meta Orchestrator has joined the cluster, it sends through RabbitMQ a message with the status of the action. Since the process was completed successfully, the status returned is "ok", Figure 30.

```
→ kubectl logs moca-cluster-parser-5bbddfbd47-kxfzj -n nemo-svc
[12/May/2025 18:57:38] - [nemo.rabbitmq:124] - [INFO]  [*] Waiting for messages. To exit press CTRL+C
[20/May/2025 14:38:49] - [nemo.rabbitmq:63] - [INFO]  [x] Received {'action': 'join', 'managed_api': 'https://api.main.nemo.onelab.eu:6443', 'cluster_name': 'test-cluster-2', 'operation_id': '2603d351-d2cc-46a0-9f44-5b4147d88598', 'timestamp': '2025-05-20T16:38:49+02:00', 'status': 'ok'}
[20/May/2025 14:38:51] - [app.utils.utils:282] - [INFO] http://127.0.0.1:8080/ipfs/QmTtpAaYdtiWEs5UweDtRyCffW6uTdaMvuaHYBSNrPZKZc
[20/May/2025 14:38:56] - [nemo.rabbitmq:81] - [INFO] This is the CID: QmTtpAaYdtiWEs5UweDtRyCffW6uTdaMvuaHYBSNrPZKZc
[20/May/2025 14:39:02] - [nemo.rabbitmq:94] - [INFO] The cluster was registered with transaction hash 0x9c52620b258fbbe0a6130d6d0661b363bdccbc64ce7981053757f531eb1e5474
```

Figure 30. Meta-Orchestrator response

After receiving this message, MOCA continues to store the provided cluster configuration file and register the cluster to the blockchain. Figure 30 provides some logs that show that both actions were completed successfully. The file was stored in IPFS and can be retrieved, if necessary, with the produced

CID, a unique identifier of that file. IPFS here is used as an extra storage layer which guarantees that any tampering attempts to the cluster configuration file will be easily detectable, as any changes to the content will produce a new CID. Since the configuration file holds sensitive information, it is important to ensure the integrity of the data. After this is completed successfully, the cluster is registered in the blockchain. Figure 31 shows the details of the transaction.

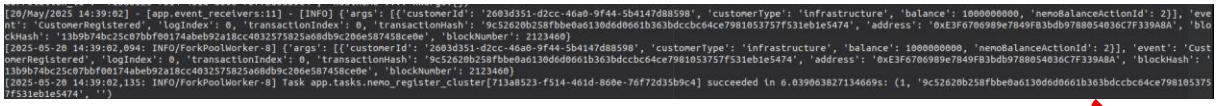


Figure 31. Register cluster to blockchain

5.1.2.5 Cluster status update visualization in LCM UI

At the end of the process, MOCA updates the status of the cluster to “OK”. This is visible through the LCM UI, as shown in Figure 32.

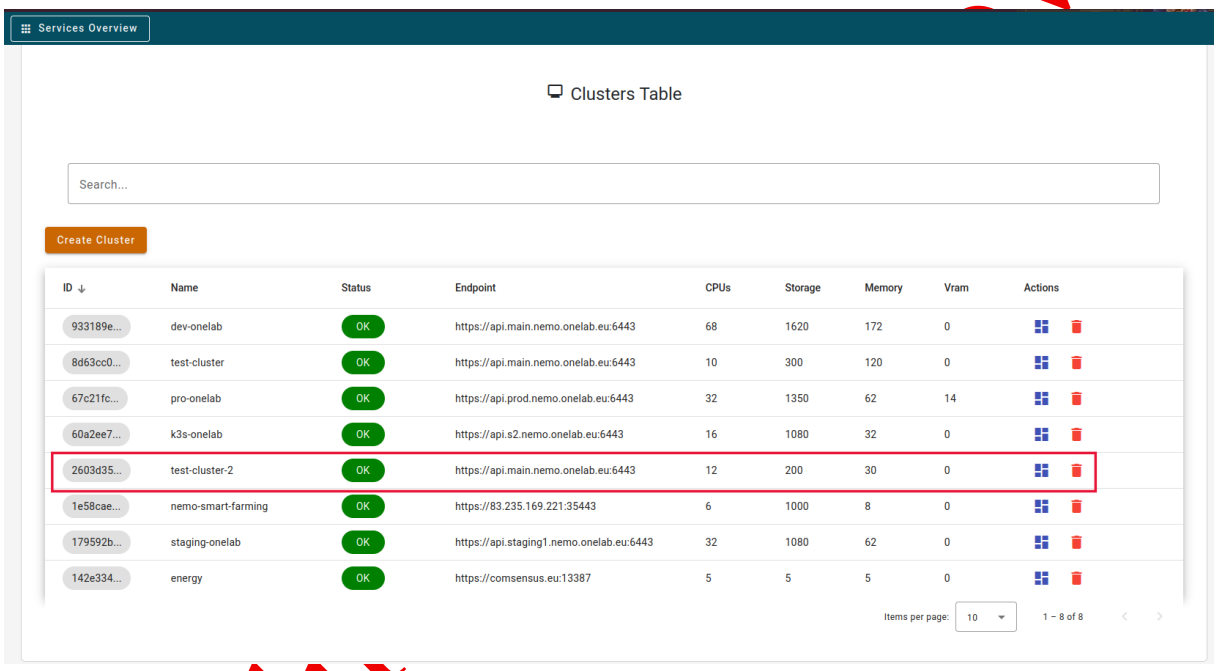


Figure 32. Updated cluster status

5.1.2.6 Results from meta-Orchestrator

In step 3, “Cluster addition process by MO,” MO receives the cluster joining request from the RabbitMQ queue. After several communications between MO’s subcomponents, MO API registers the cluster if the request is well formed and there is no error. Listing 7 presents an example of a joining request payload that the MO is expecting to receive. If there is no error, the MO will send an ok message, Listing 8. Otherwise, if an issue has occurred, either syntactic error or connectivity related, with the request, the MO returns an error, Listing 9. The error presented in Listing 9, error 409 is related to a joining request for an existing cluster.

```
{
    "availability": "99.9%",
    "cluster_name": "test",
    "cost": "low_cost",
    "cpu_base_rate": 10,
    "cpus": 10,
```

```
    "green_energy": "20%",
    "id": "28aa341f-246d-4890-886b-79529d8e8b7e",
    "managed_api": "https://api.s2.nemo.onelab.eu:6443",
    "memory": 200,
    "memory_base_rate": 10,
    "storage": 300,
    "timestamp": "2025-02-04T10:37:55.640728Z",
    "vram": 1024
}
```


Listing 7. Joining cluster request payload


```
{
    "action": "join",
    "cluster_name": "test-cluster",
    "managed_api": "https://testing:6443",
    "operation_id": "4981421-test-join",
    "status": "ok",
    "timestamp": "2025-05-26T08:12:02+02:00"
}
```


Listing 8. Successfully joining request


```
{
    "action": "join",
    "cluster_name": "test_cluster",
    "error": "request failed: http request failed with status 409",
    "id": "4981421-test-join",
    "status": "error"
}
```


Listing 9. Error joining request

Finally, after the joining request succeeds, each cluster's metrics persist in the MO database. The metrics used for the Workload Placement were described in deliverable D3.3 [20], Table 8.

Table 8. Cluster Metrics

Field	Type	Title	Description
cluster_name	string	Cluster name	The name of the Cluster that will be deployed.
cpus	number	CPUs	The number of CPUs of the Cluster.
memory	number	Memory	The RAM of the Cluster in MB.
storage	number	Storage	The disk storage of the Cluster in MB.
availability	string	Availability	The percentage of time that the cluster is up (99.9%, 99%, 90%).

Field	Type	Title	Description
green_energy	string	Green energy	The percentage of RES powering the cluster (0%, 20%, 40%, 60%, 80%, 100%).
cost	string	Cost	The cost type of a cluster (low cost, high performance). Enum
cpu_base_rate	number	CPU base rate	The CPU cost of the cluster by the CPU capacity of the cluster (in milliseconds).
memory_base_rate	number	Memory base rate	The memory cost of the cluster by the memory capacity of the cluster (in MBs).
vram	number	Video Random Access Memory	The cluster VRAM available in MB

5.2 NEMO workload registration & provisioning

Once the cluster registration has been performed within the NEMO infrastructure, a NEMO user can proceed to register, deploy and provision the corresponding workflows in the infrastructure. Figure 33, Figure 34 and Figure 35 depict the whole process necessary for the proper deployment of a workload within NEMO.

Since both the workload provisioning, Figure 33, and deployment, Figure 34, were already described in deliverable D4.2. This section of this document focuses on the interactions that occur during the deployment process of a NEMO workload, especially detailing how NEMO workloads take advantage of the mNCC flexibility to support isolated communication between its functionalities. Supplementary to the workload provisioning, the workload deployment process showed in the Figure 34 is also a key part for the completion of this experiment.

Following the initial stages of the workload deployment process illustrated in Figure 33 and Figure 34, the workflow continues with the creation of the network elements necessary to establish isolated communication channels between NEMO Workloads deployed over the project's infrastructure. Figure 35 shows the creation process of a virtual network by the mNCC NEMO component.

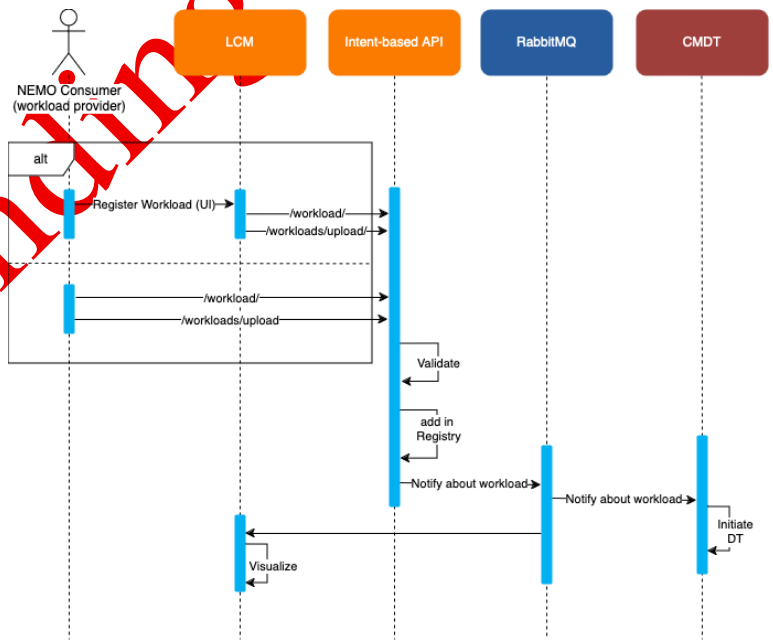


Figure 33. Workload provisioning workflow in NEMO

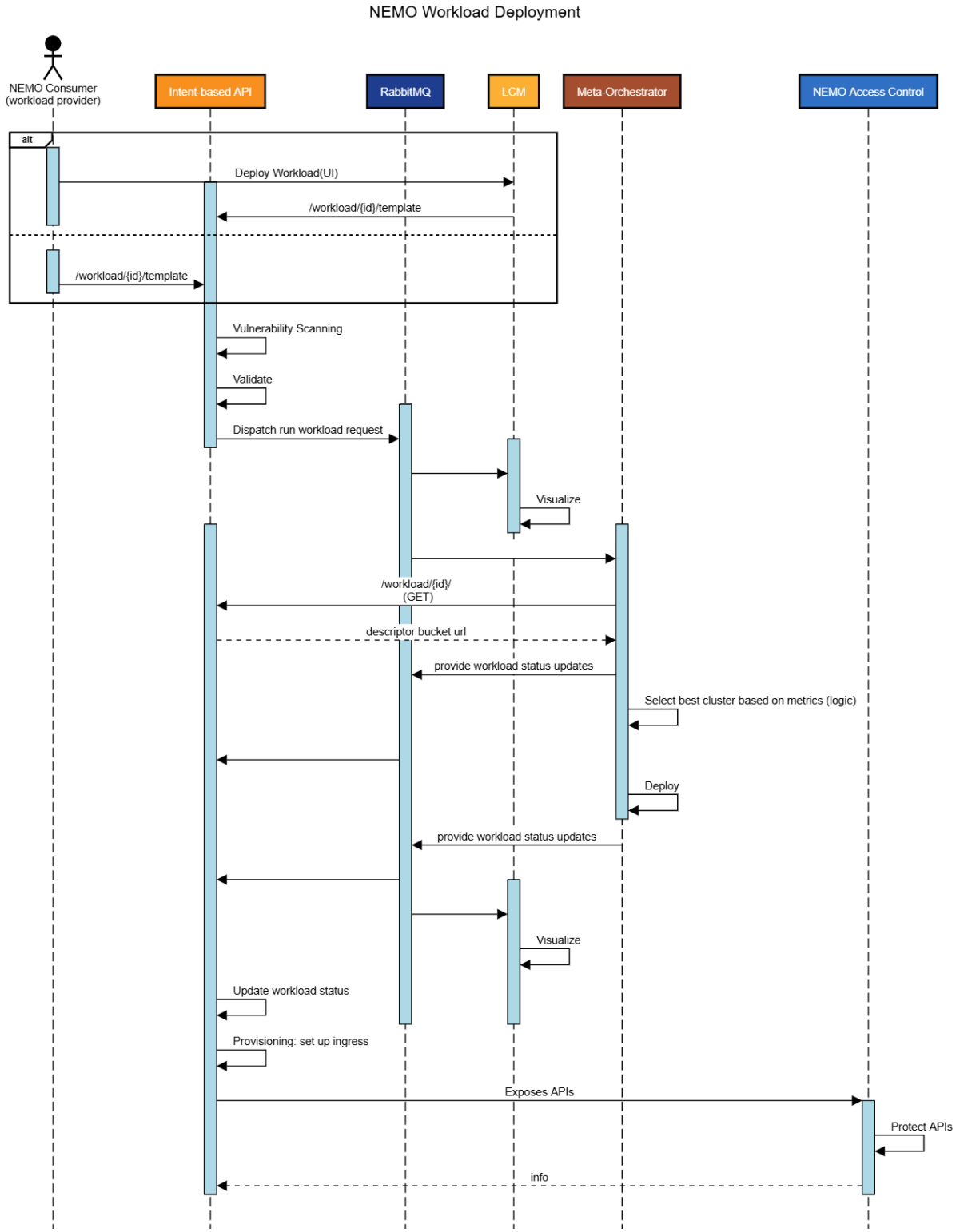


Figure 34. NEMO workload deployment workflow

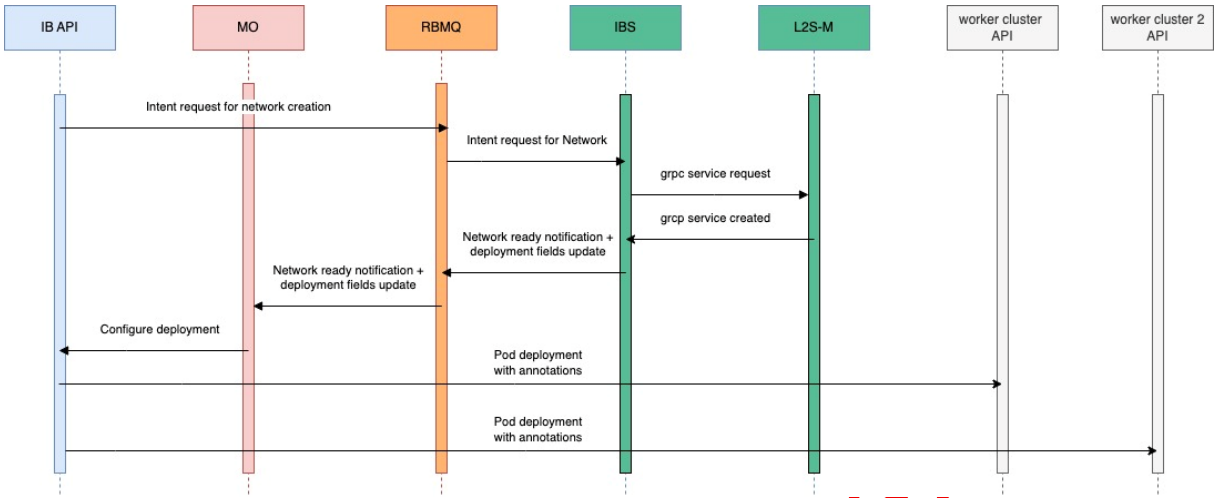


Figure 35. Network workload deployment workflow

The virtual networks created by the mNCC, enable isolated communications between two (or more) pods located in one, or multiple, Kubernetes (K8s) clusters in NEMO. This step must be performed before the deployment of the pods themselves within the clusters used for the deployments, since each virtual network is defined (and created) as a K8s resources that each pod must consume at its instantiation stage.

In this regard, when a new virtual network is created through the mNCC component, the pods that consume this K8s resource (i.e., pods attach to this virtual network) will be able to communicate between each other as if they were located in the same Local Area Network (LAN), regardless of their physical location. More details about these virtual network resources can be seen in Deliverables D2.2 and D2.3.

It is important to address that, although the original workflow included the creation of multiple network slices where various NEMO administrative domains could be split based on the needs of each platform at network level, dynamically modifying the network topology needed for each use case, the workflow presented in this integration document will feature a single network slice. The purpose of this decision is two-fold: on the one hand, due to the relatively low complexity of the current NEMO infrastructure, splitting the network into multiple slices will not provide significant improvements for the isolation of NEMO deployments, since it will be done using the virtual networks created in the default slice. On the other hand, it simplifies the NEMO workload deployment process, as the user will not need to decide in which slide should the NEMO deployment be located. Nevertheless, this slice creation and modification functionality is enabled within the mNCC to enhance the scalability of NEMO once more clusters are aggregated to the NEMO project infrastructure.

In order to start the deployment of each virtual network in the default slice present in NEMO, the Intent-based API component must create the associated network intents that represent the network interconnecting the pods from a NEMO workload.

In Table 9 an overview of how to construct an intent to request a network connection between different clusters is presented. Later in this section, each value will be explained. In Table 9, a complete example of a L2S-M Intent is described.

Table 9. Intent for network connectivity request. L2S-M network request attributes.

Key	Description	Value
Intent.id	Identifier for the intent	<string>
Intent.userLabel	Label to identify the intent among others posted by the intentAPI	cloud_continuum
Intent.intentExpectations	List of expectations for the network	<intentExpectations>
intentExpectation.expectationId	Identifier for the expectation	<string number>
intentExpectation.expectationVerb	Verb defining what type of action to apply in the object.	DELIVER***, ENSURE
intentExpectation.expectationObject	Type of object to be delivered by the network	<expectationObject>
expectationObject.objectType	Defines type of network intent	K8S_L2_NETWORK*, K8S_CLUSTER_CONFIG**
expectationObject.objectContexts	Set of attributes required to define a specific object	<objectContexts>
objectContext.contextAttribute **objectContext.contextCondition** **objectContext.contextValueRange**	String defining the attribute. The value or range is set on contextValueRange. And the condition restricting the value or range in contextCondition	name***, providerName*, domain*, pod_cidr* bearer_token**
intentExpectation.expectationTarget	Objective to fulfil by the expectation object	isolatedConnectivity*
intentExpectaion.expectationContext	Scope of the related expectation.	namespace**
Intent.intentContext	Scope of the entire intent. In the NEMO platform the main context is the related workload	NEMO_WORKLOAD

The network intent is read from the exchange/queue "nemo.api.workload/network-intent" and processed only by the IBS if the *userLabel* matches a network intent. The is classified and translated as described with more detail in D2.2 and D2.3 deliverables. Additionally, is doing some simple network management in order to fit L2S-M requirements and abstract simple networking behaviours from the higher levels in the architecture.

Once the intent has been processed in the Intent Based System (IBS), this component will generate a gRPC [21] [22] [23] service request towards L2S-M (particularly, the L2S-M MD component). This request must include the following fields:

- Network Network Class-less Inter Domain Routing (CIRD), which will be used to automatically assign IP addresses to pods using the virtual network.
- A Provider endpoint. This provider is a combination of a DNS Server and as SDN Controller which will act as a central point which manages the network.

- List of clusters where the network is going to be created. This list includes:
  - Kubernetes endpoint, including the API Key and Bearer Token of a virtual network user.
  - Namespace to be used inside the cluster.
  - The address pool, contained inside the CIDR for the virtual network. For instance, if the chosen CIDR in a new virtual network is 10.1.0.0/16, each cluster would use a subnet like 10.1.n.0/24. This subdivision is processed and calculated by the IBS.

Once this request is received in L2S-M, it will create a virtual network resource in each specified K8s cluster. This resource will be created at the same namespace as the NEMO workload deployment, since virtual networks are *namespaced* resources that must be present in the same namespace of the pods that will use them. Furthermore, once this resource is created in each cluster, the inter-domain L2S-M component will register this creation and enable the inter-domain connectivity between them, as well as generating the DNS domain record that must be added in the pod deployments to enable their connectivity within the virtual network (avoiding the need to manually configure the IP addressing of each pod).

To enable pods to utilize the virtual network resource, L2S-M includes the required configuration fields in the gRPC response sent to the IBS. Specifically, any NEMO deployment intending to use the newly provisioned network must specify the following fields in their deployment descriptors:

- L2S-M annotation label, to clarify that the networking resource is managed by this component.
- L2S-M name of the application label (defined by the deployment owner), which will be used to identify the application with DNS.
- The name of the L2S-M virtual network to which the pods will be attached
- DNS name for each pod (L2S-M provides the DNS domain; the deployment owner is responsible for assigning the specific name to the application)

L2S-M populates these values in the gRPC response to the IBS, effectively providing all the necessary information required for the deployment to leverage the virtual network within the Kubernetes cluster.

The Intent-based API has a RabbitMQ consumer listening to *exchange=mncc, (routing_key,queue)=(mncc.ibs, mncc.ibs)* for the values mentioned above. A sample message payload expected by the IBA is presented in Listing 10.

```
{
  "workload": "2b9c9ade-0296-4cf6-b0b9-6b83a069aef0",
  "pod": {
    "labels": {
      "l2sm": "true",
      "l2sm/app": "<workload-name>"
    },
    "annotations": {
      "l2sm/networks": "spain-network"
    },
    "env": {
      "DNS_NAME": "<workload-name>.spain-network.inter.l2sm"
    }
  }
}
```

Listing 10. Sample payload received by IBA from the RabbitMQ queue

The Intent-based API then updates the workload manifests with the above additions (pod labels, annotations and environment) while also performing substitution on the <workload-name> matching the name of the workload with the given NEMO Workload ID.

When the manifests are updated, the Intent-based API informs the MO by publishing a RabbitMQ message on the *exchange=nemo.api.workload*, *(routing_key, queue)=(run,run)* with the message payload containing "status" field set to "updating", signaling to the MO to update the NEMO Workload residing in the corresponding cluster.

5.2.1 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
	  ○ NEMO Access Control
	• NEMO Workload provisioning
	  ○ Intent-based API
	  ○ Access Control
	• NEMO Network Deployment
	  ○ Intent-based API
	  ○ meta-Orchestrator
	  ○ RabbitMQ
	  ○ IBS (mNCC)
	  ○ L2S-M (mNCC)

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 • Network Deployment 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 GUI & Intent-based API. Then, the newly registered workload is requested to be deployed into the NEMO meta-OS ecosystem by the MO. Following 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. The Intent-based API initiates a network creation intent, which is propagated through RBMQ to mNCC and L2S-M, triggering gRPC service setup. Once the network is ready, updated deployment fields are returned through the same path to the Intent-based API.

Test 2: NEMO workload registration and provisioning	
	The updated workload is then published and deployed to the managed clusters with appropriate network annotations by MO.
Test setup	The associated components are deployed in OneLab facilities
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. Workload placement using cluster metrics by MO g. Deployment operation process executed by MO h. Communication and update of the deployment operation status to the Intent-API i. Expose the services using NEMO access control j. 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[14] Plugin 4. NEMO Network Deployment a. Intent request for network creation by Intent-Base API b. Intent request for Network received mNCC c. gRPC service request from mNCC to L2SM d. gRPC service created by L2SM e. Network ready notification + deployment fields update from mNCC to Intent-based API. f. Publish updated workload into RabbitMQ g. Receive updated workload bt MO h. Updated workload by MO- i. MO deploys pod with network annotations and updates workload into Managed Clusters selected by the HUB

5.2.2 Results

The rest of the subsections present the corresponding results step by step as indicated by the verification scenario.

5.2.2.1 NEMO workload deployment

This section of the present document will be centred around the interactions between the NEMO components and the mNCC component to create network intents that workloads will use in order to communicate through the infrastructure.

[14] [46] [27]

When a new intent arrives with the intent-notify key to the Rabbit-MQ is then read by the IBS. The first step is to filter and detect the type of intent by the label as show in Figure 36. In case the label is not referring to cloud continuum connectivity, the intent is discarded.

```
[INFO - 07:51:28] intent_engine.executioners.rabbitMQ_recv: [Thread-1 (reciver)] Sending cloud_continuum message to queue
[INFO - 07:51:28] intent_engine.executioners.rabbitMQ_recv: [Thread-1 (reciver)] New message from RMQ: intent-notify
[DEBUG - 07:51:28] intent_engine.core.ib_model: [MainThread] Schema Type Expectation: <class 'intent_engine.core.IntentNrm.IntentExpectatio
n'>
{
    'id': '154',
    'userLabel': 'cloud_continuum',
    'intentExpectations': [
        {
            'expectationId': '1',
            'expectationVerb': 'DELIVER',
            'expectationObject': {
                'objectType': 'K8S_L2_NETWORK',
```

Figure 36. Arrival of intent and first filter.

Then, the intent is classified and processed in the IBS as explained in deliverable D2.3. Then, the intent triggers the execution to a call of L2S-M grpc server. This is shown in the last lines of Figure 37, where the grpc executioner is called and starts the exchange of the network request.

```
[DEBUG - 07:51:28] intent_engine.catalogue.l2sm: [MainThread] Workload id: b30c0b56-9eed-4118-ac6f-08ecef16
[DEBUG - 07:51:28] intent_engine.catalogue.l2sm: [MainThread] Compiling l2sm schema...
[DEBUG - 07:51:28] intent_engine.catalogue.l2sm: [MainThread] pod_cidr subneting...
[DEBUG - 07:51:28] intent_engine.catalogue.l2sm: [MainThread] cluster name: staging-1
[DEBUG - 07:51:28] intent_engine.catalogue.l2sm: [MainThread] existing net! subnetworks: 10.1.1.0
 ------ Execution platform -------
[INFO - 07:51:28] intent_engine.core.intent_core: [MainThread] Runnning sys_out
 -> -> Executing:
[
    {
        'network': {
            'name': 'spain-network-2',
            'provider': {
                'name': 'default-slice',
                'domain': 'api.main.nemo.onelab.eu',
            },
            'pod_cidr': '10.1.0.0/16',
            'clusters': [
                {
                    'name': 'staging-1',
[INFO - 07:51:28] intent_engine.core.intent_core: [MainThread] Runnning grpc_connector
[DEBUG - 07:51:28] intent_engine.executioners.grpc_connector: [MainThread] l2smmd grpc proto
[DEBUG - 07:51:28] intent_engine.executioners.grpc_libs.l2smmd: [MainThread] Reading enviroment GRPC_SERVICE_L2SM_ADDRESS vars: l2sm-grpc-s
ervice.nemo-net-svc.cluster.local
```

Figure 37. Grpc connector execution

```
[INFO - 07:51:29] intent_engine.executioners.grpc_libs.l2smmd: [MainThread] Sending CreateNetworkResponse to rabbitmq...
[INFO - 07:51:29] intent_engine.executioners.rabbitMQ_emit: [MainThread] Connecting to RabbitMQ at nemo-rabbitmq.nemo-sec.svc.cluster.local
:5672
[DEBUG - 07:51:29] intent_engine.executioners.rabbitMQ_emit: [MainThread] Sending message: {"workload": "b30c0b56-9eed-4118-ac6f-08ecef16",
 "pod": {"labels": {"l2sm": "true", "l2sm/app": "<workload-name>"}, "annotations": {"l2sm/networks": "spain-network-2"}, "env": {"DNS_NAME"
: "<workload-name>.spain-network-2.inter.l2sm"}}}
[INFO - 07:51:29] intent_engine.executioners.rabbitMQ_emit: [MainThread] Channel created
[DEBUG - 07:51:29] intent_engine.executioners.rabbitMQ_emit: [MainThread] Key: mncc.ibs, Routing Key: mncc.ibs, Message: {"workload": "b30c
0b56-9eed-4118-ac6f-08ecef16", "pod": {"labels": {"l2sm": "true", "l2sm/app": "<workload-name>"}, "annotations": {"l2sm/networks": "spain-n
etwork-2"}, "env": {"DNS_NAME": "<workload-name>.spain-network-2.inter.l2sm"}}}
[INFO - 07:51:29] intent_engine.executioners.rabbitMQ_emit: [MainThread]  [x] Sent mncc.ibs:{"workload": "b30c0b56-9eed-4118-ac6f-08ecef16"
, "pod": {"labels": {"l2sm": "true", "l2sm/app": "<workload-name>"}, "annotations": {"l2sm/networks": "spain-network-2"}, "env": {"DNS_NAME
": "<workload-name>.spain-network-2.inter.l2sm"}}}
 [x] Sent mncc.ibs:{"workload": "b30c0b56-9eed-4118-ac6f-08ecef16", "pod": {"labels": {"l2sm": "true", "l2sm/app": "<workload-name>"}, "ann
otations": {"l2sm/networks": "spain-network-2"}, "env": {"DNS_NAME": "<workload-name>.spain-network-2.inter.l2sm"}}}
-------- Intent procesed !!! ---------
```

Figure 38. L2S-M annotations to be used by the workload.

Once the intent has been received from the IBS component, L2S-M proceeds to create the virtual network in the relevant clusters, as shown in Figure 39 and Figure 40 and each network has been properly created in the K8s clusters as a custom resource. In consequence, its details can be obtained through the command line interface of the cluster, showcasing how the combination of the IBS and the L2S-M components enable the creation of virtual networks in the NEMO infrastructure (i.e., the mNCC NEMO component).

```
alex@alex-pc:~$ kubectl get l2network -n fcc6f34d-f33a-4bfe-a66b-764d9bedd113 --context nemo-s
1
NAME              AVAILABILITY   CONNECTED_PODS   AGE
spain-network-1   Available      1                15d
alex@alex-pc:~$ kubectl describe l2network spain-network-1 -n fcc6f34d-f33a-4bfe-a66b-764d9bed
d113 --context nemo-s1
Name:         spain-network-1
Namespace:    fcc6f34d-f33a-4bfe-a66b-764d9bedd113
Labels:       <none>
Annotations:  <none>
API Version:  l2sm.l2sm.k8s.local/v1
Kind:         L2Network
Metadata:
  Creation Timestamp:  2025-04-30T13:30:09Z
  Finalizers:
    l2sm.operator.io/finalizer
  Generation:        1
  Resource Version:  63712928
  UID:               478a8267-fee9-47a7-81e2-947114a90c34
Spec:
  Config:             10.1.0.0/16
  Network CIDR:       10.1.0.0/16
  Pod Address Range:  10.1.2.0/24
  Provider:
    Dns Grpc Port:  30818
    Dns Port:       30053
    Domain:         132.227.122.23
    Name:           default-slice
    Of Port:        6633
    Sdn Port:       30808
  Type:             vnet
Status:
  Assigned I Ps:
    10.1.2.1:
  Connected Pod Count:    1
  Internal Connectivity:  Available
  Last Assigned IP:       10.1.2.1
  Provider Connectivity:  Available
Events:                   <none>
```

Figure 39. Network resource created in S1 cluster

```
alex@alex-pc:~$ kubectl get l2networks -n 6512a712-ae0f-4839-9703-fcec4fb13828 --context nemo-
s2
NAME              AVAILABILITY   CONNECTED_PODS   AGE
spain-network-1   Available      2                15d
alex@alex-pc:~$ kubectl describe  l2networks -n 6512a712-ae0f-4839-9703-fcec4fb13828 spain-net
work-1 --context nemo-s2
Name:         spain-network-1
Namespace:    6512a712-ae0f-4839-9703-fcec4fb13828
Labels:       <none>
Annotations:  <none>
API Version:  l2sm.l2sm.k8s.local/v1
Kind:         L2Network
Metadata:
  Creation Timestamp:  2025-04-30T13:28:06Z
  Finalizers:
    l2sm.operator.io/finalizer
  Generation:        1
  Resource Version:  54706992
  UID:               35a6a3ab-8ea3-4727-9a10-84cbc92e687a
Spec:
  Config:             10.1.0.0/16
  Network CIDR:       10.1.0.0/16
  Pod Address Range:  10.1.1.0/24
  Provider:
    Dns Grpc Port:  30818
    Dns Port:       30053
    Domain:         132.227.122.23
    Name:           default-slice
    Of Port:        6633
    Sdn Port:       30808
  Type:             vnet
Status:
  Assigned I Ps:
    10.1.1.1:
    10.1.1.4:
  Connected Pod Count:    2
  Internal Connectivity:  Available
  Last Assigned IP:       10.1.1.4
  Provider Connectivity:  Available
Events:                   <none>
```

Figure 40. Network resource created in S2 cluster

Since the L2S-M annotations reported by the mNCC were added to each pod during the deployment process, each pod consequently deployed in each cluster attached to its corresponding network. These details can be found both in the description of the virtual network (i.e., Figure 39 and Figure 40) as well as the annotations present in each pod within the K8s cluster, as depicted in Figure 41 and Figure 42.

```
alex@alex-pc:~$ kubectl get pods -n fcc6f34d-f33a-4bfe-a66b-764d9bedd113 --context nemo-s1
NAME                                        READY   STATUS     RESTARTS   AGE
echo-server-integration-500-7ccbd97fc-2ctrws   2/2     Running    0          15d
echo-server-integration-500-test-connection    1/2     NotReady   0          15d
alex@alex-pc:~$ kubectl describe pod echo-server-integration-500-7ccbd97fc-2ctrws -n fcc6f34d-
f33a-4bfe-a66b-764d9bedd113 --context nemo-s1 | grep l2sm
                  l2sm=true
                  l2sm/app=echo-server-integration-500
                  l2sm/networks: spain-network-1
      DNS_NAME:  echo-server-integration-500.spain-network-1.inter.l2sm
```

Figure 41. Pod deployed in S1 with L2S-M annotations

```
alex@alex-pc:~$ kubectl get pods -n 6512a712-ae0f-4839-9703-fcec4fb13828 --context nemo-s2
NAME                                        READY   STATUS     RESTARTS   AGE
echo-server-integration-501-89b7c85b6-ggsz9   2/2     Running    0          15d
echo-server-integration-501-89b7c85b6-s6xck   2/2     Running    0          15d
echo-server-integration-501-test-connection   1/2     NotReady   0          15d
alex@alex-pc:~$ kubectl describe pod echo-server-integration-501-89b7c85b6-ggsz9 -n 6512a712-a
e0f-4839-9703-fcec4fb13828 --context nemo-s2 | grep l2sm
                  l2sm=true
                  l2sm/app=echo-server-integration-501
                  l2sm/networks: spain-network-1
      DNS_NAME:  echo-server-integration-501.spain-network-1.inter.l2sm
alex@alex-pc:~$ kubectl describe pod echo-server-integration-501-89b7c85b6-s6xck -n 6512a712-a
e0f-4839-9703-fcec4fb13828 --context nemo-s2 | grep l2sm
                  l2sm=true
                  l2sm/app=echo-server-integration-501
                  l2sm/networks: spain-network-1
      DNS_NAME:  echo-server-integration-501.spain-network-1.inter.l2sm
```

Figure 42. Pod deployed in S2 with L2S-M annotations

It is noteworthy that these details provide the DNS name that pods can use to communicate between each other in the newly created virtual networks. Hence, each pod does not need to know the IP address of the other component, but only its DNS entry, simplifying the connectivity setup between each other with a fully compatible solution with the standard CoreDNS present in standard K8s clusters. With this name, each pod can communicate with one another using its entry, as demonstrated in Figure 43 and Figure 44, where a ping command is executed between the pod present in one of the clusters.

```
alex@alex-pc:~$ kubectl exec -it  -n fcc6f34d-f33a-4bfe-a66b-764d9bedd113 --context nemo-s1 ec
ho-server-integration-500-7ccbd97fc-2ctrw -c echo-server  -- ping echo-server-integration-501.
spain-network-1.inter.l2sm -c 3
PING echo-server-integration-501.spain-network-1.inter.l2sm (10.1.1.1): 56 data bytes
64 bytes from 10.1.1.1: seq=0 ttl=64 time=30.564 ms
64 bytes from 10.1.1.1: seq=1 ttl=64 time=1.234 ms
64 bytes from 10.1.1.1: seq=2 ttl=64 time=1.225 ms

--- echo-server-integration-501.spain-network-1.inter.l2sm ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.225/11.007/30.564 ms
```

Figure 43. Ping between pods in S1 and S2 clusters

```
alex@alex-pc:~$ kubectl exec -it  -n fcc6f34d-f33a-4bfe-a66b-764d9bedd113 --context nemo-s1 ec
ho-server-integration-500-7ccbd97fc-2ctrw -c echo-server  -- ping echo-server-integration-501.
spain-network-1.inter.l2sm -c 3
PING echo-server-integration-501.spain-network-1.inter.l2sm (10.1.1.4): 56 data bytes
64 bytes from 10.1.1.4: seq=0 ttl=64 time=7.069 ms
64 bytes from 10.1.1.4: seq=1 ttl=64 time=1.216 ms
64 bytes from 10.1.1.4: seq=2 ttl=64 time=1.311 ms

--- echo-server-integration-501.spain-network-1.inter.l2sm ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.216/3.198/7.069 ms
```

Figure 44. Ping between pods in S1 and S2 clusters

Since the execution of the ping was successful, these results showcase that NEMO workloads can communicate with specialised networking tools thanks to the flexibility of the mNCC.

5.3 NEMO workload scheduling & orchestration

This integration scenario aims to illustrate the workload migration process and orchestration. This process is represented by three different tasks:

- Intent-Based workload migration
- CFDRL workload migration
- Workload horizontal scaling

The first two tasks concern the migration of the workload migration from one cluster to another. The main difference concerns the component that triggers the said migration. This process is depicted in Figure 45.

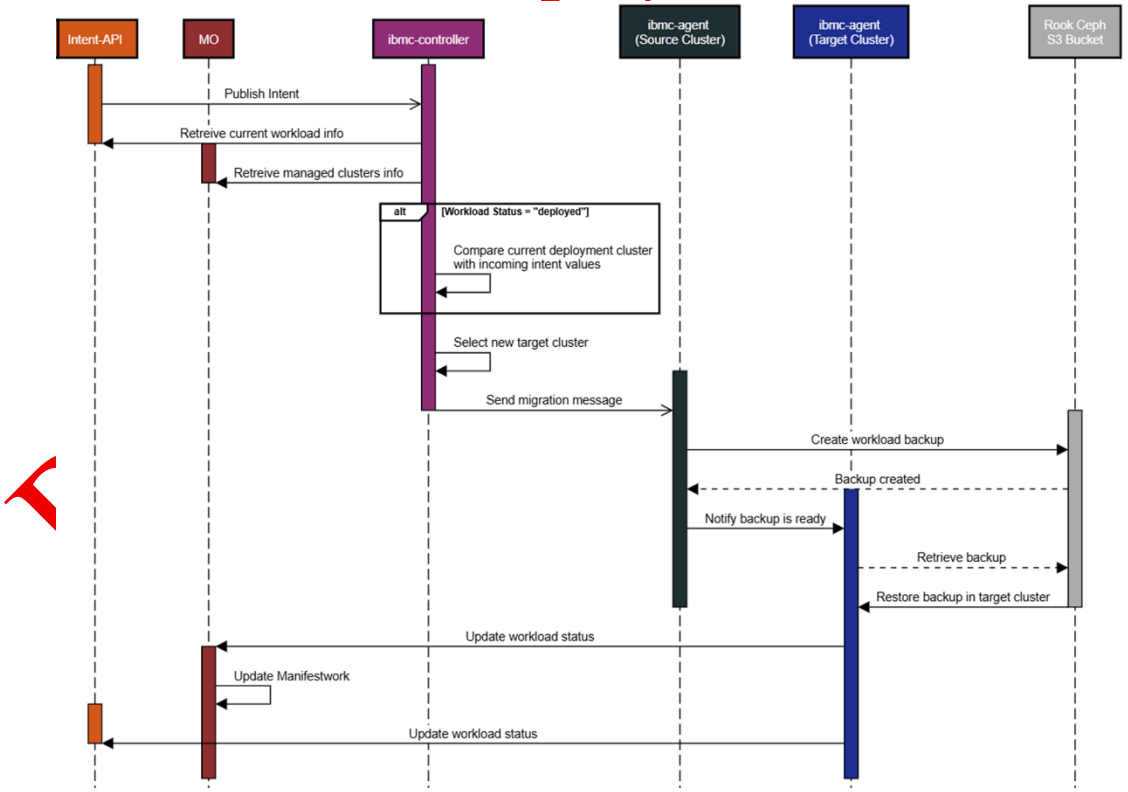


Figure 45. Intent-Based Migration Sequence Diagram

For the Intent-Based migration, the Intent-Based API is responsible for triggering the migration process by publishing an intent in RabbitMQ. This intent contains several requirements that the cluster where the workload is currently deployed must comply with. If one of the requirements is not met, the IBMC selects a new target cluster and performs the migration.

In the case of the CFDRL migration, machine learning algorithms are used to detect whether the cluster where the workload is deployed is not suitable anymore, using the information from workload metrics and trained models. A new target is then selected by the CFDRL and a migration message is sent to the IBMC to trigger migration as shown in Figure 46.

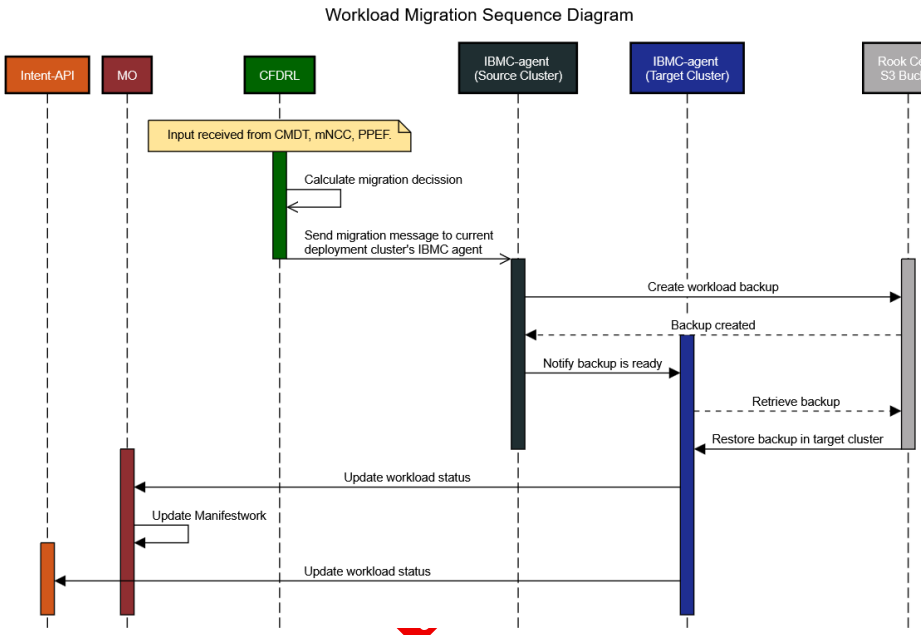


Figure 46. CFDRL Migration Sequence Diagram

Finally, the workload horizontal scaling scenario also relies on the CFDRL component to detect the need of upscaling or downscaling the number of replicas of a deployed workload. When this happens, a message is sent to the MO, which performs the scaling action, Figure 47.

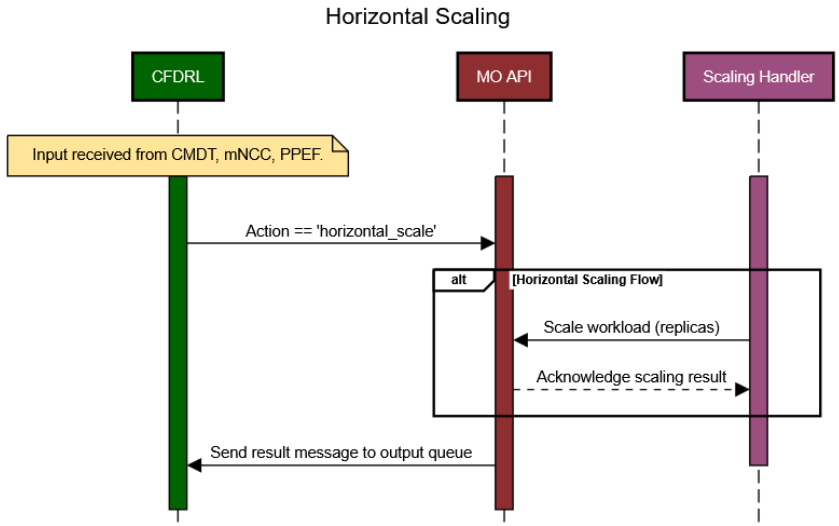


Figure 47. Horizontal Scaling Sequence Diagram

5.3.1 Verification scenario

Test 3: NEMO workload migration	
Objective	The objective of this task is to validate the NEMO workload migration process.
Components	IBMC Intent-Based API MO CFDRL CMDT PPEF mNCC
Features to be tested	• Workload migration: 1. By the creation of an intent, containing one or more requirements which the deployment cluster must met. If any of these requirements is not fulfilled, the ibmc-controller triggers a migration to a more suitable cluster. 2. 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. • Horizontal scaling of a deployed workload
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).

Test 3: NEMO workload migration	
Steps	The steps identified in the associated sequence diagrams are listed below: a. Intent-Based Migration 1. Intent-based API publishes an Intent with one or more requirements (Availability, GreenEnergy, CPU, Memory etc). 2. Ibmc-controller retrieves workload status from the Intent-Based API. 3. Ibmc-controller retrieves the managed clusters information from the MO. 4. If the workload is already deployed and the intent requirements are not met, then a migration action is triggered. 5. A new target cluster compliant with the intent requirements is selected from the available managed clusters list. 6. Ibmc-controller sends a message to the source cluster ibmc-agent. 7. A backup of the workload is created and uploaded to Rook-Ceph S3 Bucket located in the HUB cluster. 8. Upon backup completion, a message is sent to the target cluster ibmc-agent to continue with the migration process. 9. The backup is restored in the target cluster. 10. A message is sent to the MO notifying the workload migration completion. 11. MO updates the corresponding Manifestwork to match the new workload deployment status. 12. A message is sent to the Intent-Based API updating the workload status, specifying the cluster where it has been deployed. b. CFDRL Migration 1. CFDRL gathers workload information from CMDT, PPEF and mNCC. 2. CFDRL algorithm calculates the most suitable cluster for workload migration. 3. CFDRL sends a message to the source cluster ibmc-agent. 4. A backup of the workload is created and uploaded to Rook Ceph S3 Bucket located in the HUB cluster. 5. Upon backup completion, a message is sent to the target cluster ibmc-agent to continue with the migration process. 6. The backup is restored in the target cluster. 7. A message is sent to the MO notifying the workload migration completion. 8. MO updates the corresponding Manifestwork to match the new workload deployment status. 9. A message is sent to the Intent-Based API updating the workload status, specifying the cluster where it has been deployed. c. Horizontal Scaling 1. CFDRL gathers workload information from CMDT, PPEF and mNCC. 2. CFDRL algorithm calculates the required scaling action. 3. CFDRL sends a horizontal scaling message to MO. 4. MO scales up/down the workload replicas. 5. MO sends completion a message back to CFDRL

5.3.2 Results

The rest of the subsections present the corresponding results step by step as indicated by the verification scenario.

5.3.2.1 Intent-Based Migration

Step 1

The initial conditions for the migration scenario require a workload already deployed in the source cluster from where it will be migrated, as shown in Figure 48. LCM Workload Initial Visualization. In this figure, the LCM IU is used to see that the workload with ID "3b004d45-3f08-48b8-b1b3-2abe06509d66" is currently deployed in the dev cluster.

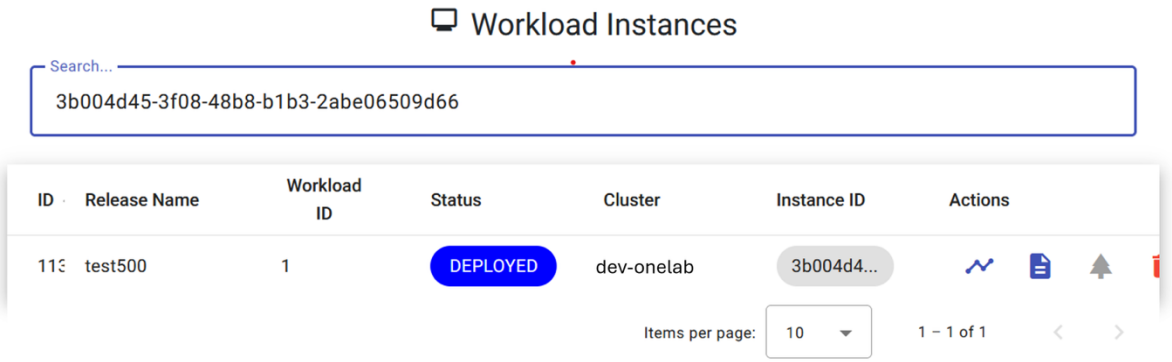


Figure 48. LCM Workload Initial Visualization

To trigger the migration process, an availability intent is created as shown in Figure 49. Availability Intent Creation, which requires that the workload is deployed in a cluster with an availability greater than 95%.

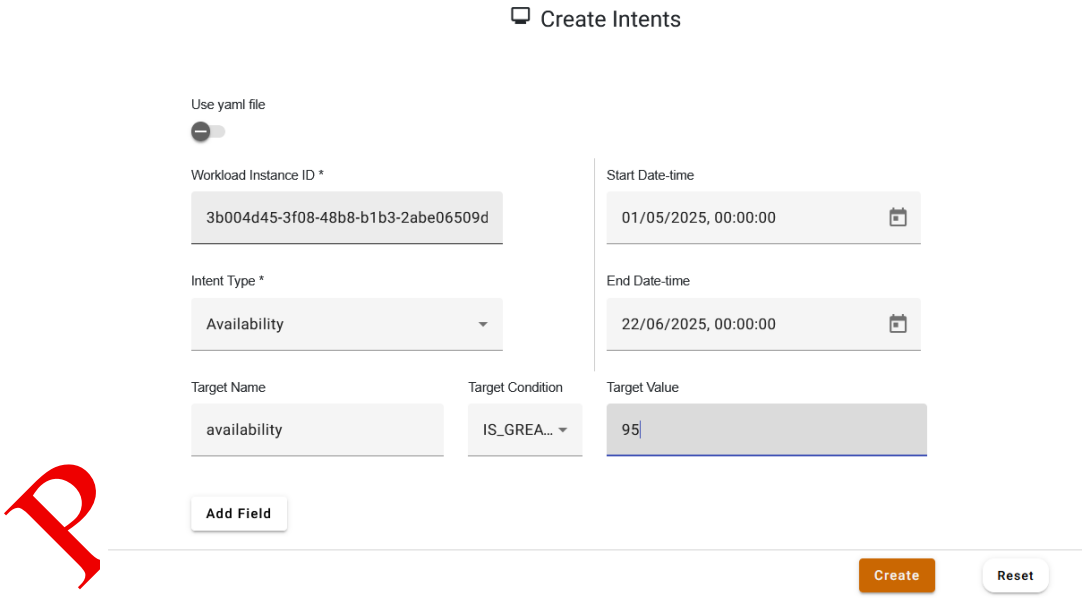


Figure 49. Availability Intent Creation

Steps 2-6

After the intent is created, it is published by the Intent-Based API. In Figure 50, the ibmc-controller deployed in the dev cluster (HUB cluster) receives the message and proceeds to check if the current deployment cluster meets the availability requirement.

Since the dev cluster has an availability value of only 90%, the intent requirements are not fulfilled, so a new target cluster is selected for the redeployment of the workload. The list of managed clusters shows that the staging cluster has an availability of 99.9% thus it is selected as a target cluster for migration.

Finally, a message is sent to the ibmc-agent deployed in the dev-cluster to trigger the migration process.

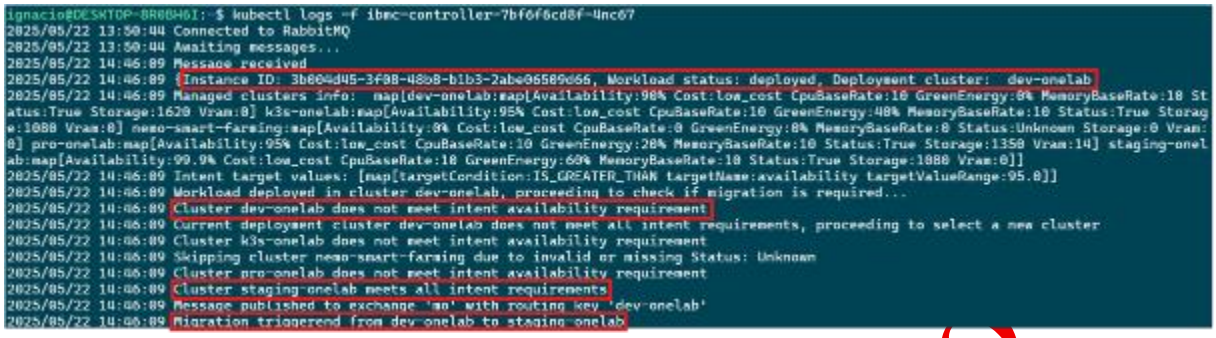


Figure 50. Ibmc-controller processing the intent

Steps 7 & 8

Figure 51 shows the migration message arriving to the dev cluster ibmc-agent, where the backup of the workload resources is performed. Once the backup is finished, a message is sent to the staging cluster ibmc-agent to continue with the migration.

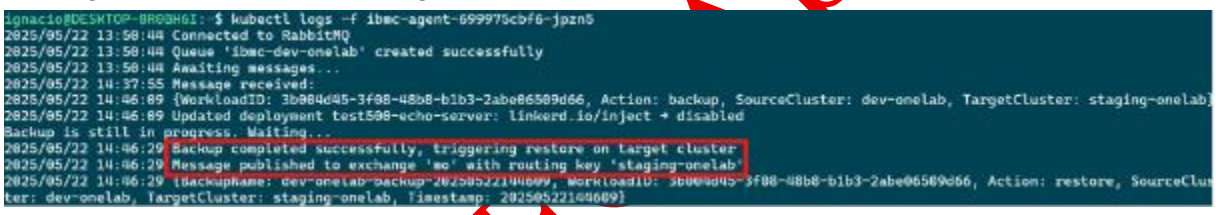


Figure 51. Dev Cluster Ibmc-agent logs

Steps 9-12

In Figure 52, the restore message is received by the staging cluster. There, the ibmc-agent waits for the syncing of the backup uploaded to Rook-Ceph and proceeds to restore the workload resources. When the restore is completed, a message is sent to both MO and Intent-Based API to notify them that the migration has been successfully completed.

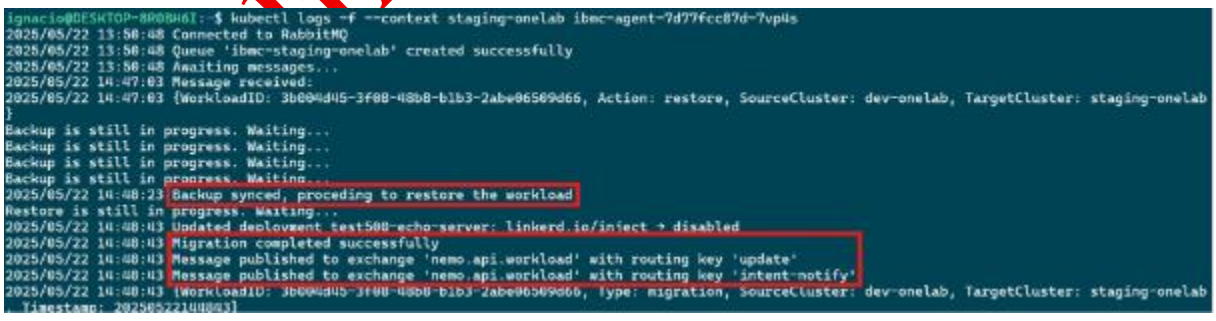


Figure 52. Staging Cluster Ibmc-agent logs

The workload status can be checked again in LCM, where the deployment cluster has been updated from the dev cluster to the staging one, as shown in Figure 53.

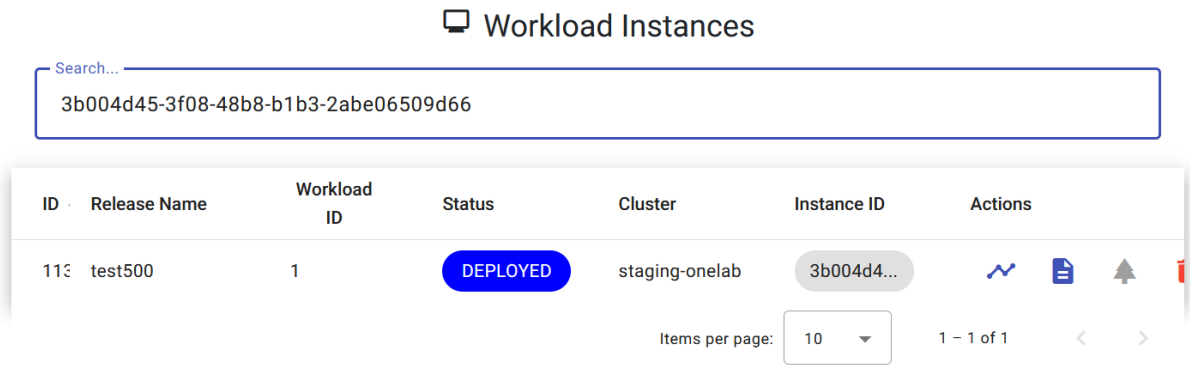


Figure 53. LCM workload final visualization

Figure 54 shows the visualization of the different steps and changes followed by the workload across the complete workflow.

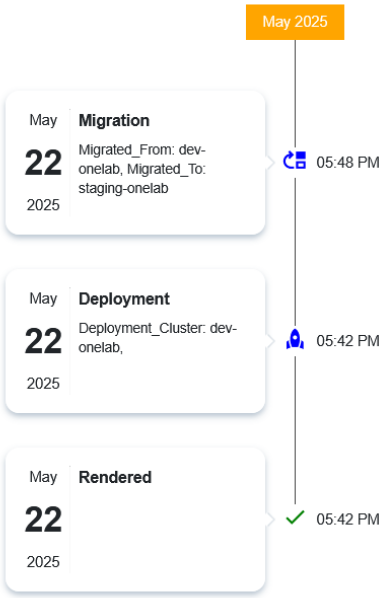


Figure 54. LCM complete workflow view

5.3.2.2 CFDRL Migration and Scaling tasks

Part of the CFDRL has been tailored towards managing and orchestrating workloads in a Kubernetes environment, with a focus on leveraging reinforcement learning (RL) techniques to optimize resource usage and performance of workloads. Below is an explanation of the project's purpose and how it is structured.

The primary goal of this project is to manage workloads efficiently in a cloud-native environment. It uses reinforcement learning (referred to as CFDRL in the project) to make intelligent decisions about workload orchestration. This involves optimizing resource allocation by scaling or migrating workloads dynamically to improve the overall performance of the system.

Scaling and migrating tasks

In the context of workloads in a Kubernetes environment, **scaling** and **migration** are two fundamental resource management tasks:

- **Scaling** refers to adjusting the number of replicas (pods) running for a given workload. This is typically done to handle changes in demand: increasing replicas to improve performance or availability when load increases and decreasing them to save resources when demand drops. In Kubernetes, scaling is a common operation and can be performed automatically (e.g., with the Horizontal Pod Autoscaler) or manually. In the CFDRL system, the scaling task involves learning a policy that decides, based on observed metrics (like CPU, RAM, and latency), how many replicas each workload should have at any given time to optimize resource usage and performance.
- **Migration** involves moving a workload (or its replicas) from one cluster or node to another. This can be necessary for load balancing, fault tolerance, cost optimization, or compliance reasons. Migration is more complex than scaling because it requires coordination between clusters or nodes, and may involve data transfer, downtime, or changes in network latency. In the CFDRL system, the migration task is formulated as learning a policy that decides, based on the current state of the system, to which cluster or node a workload should be moved to achieve optimal performance or resilience.

For simplicity, the CFDRL program tackles them by training and maintaining **two separate reinforcement learning models**: one specialized for scaling decisions and another for migration decisions. Each model learns from its own set of state-action-reward experiences, allowing the system to optimize both tasks independently and effectively in a dynamic Kubernetes environment. The difference between the two models is minor, for this reason the scaling architecture and then report the changes in the migration case are described.

Architectural Overview

The project is built around a modular architecture that integrates several key components:

Containerization with Docker: The project uses Docker to containerize its applications. Multiple Dockerfiles are provided, each tailored for specific purposes. This containerized approach ensures portability and consistency across different environments.

Kubernetes for Orchestration: Kubernetes is used as the orchestration platform, with YAML configuration files defining the deployment and storage requirements. Persistent Volumes (PV) and Persistent Volume Claims (PVC) are configured to manage data storage of the models learned by CFDRL.

Reinforcement Learning Core: The core functionality of the project revolves around reinforcement learning. The main script (main_cfdrl.py) implements the logic for managing workloads and interacting with external APIs. It likely uses reinforcement learning algorithms to analyze workload states and make decisions that optimize system performance. Metrics are logged and stored for further analysis, and old data is periodically cleaned up to maintain efficiency.

Automation and Deployment: Shell scripts are provided to automate the process of building Docker images, pushing them to a container registry, and restarting Kubernetes deployments. This streamlines the deployment process and reduces the potential for human error.

Reinforcement Learning Algorithm in the Project

The CFDRL implements a Deep Q-Network (DQN) algorithm for reinforcement learning (RL). DQN is used to optimize workload orchestration in a Kubernetes-based environment by dynamically deciding the number of replicas for workloads based on system metrics such as CPU usage, RAM usage, and

latency. Below is a detailed explanation of the RL algorithm and its implementation in the project, based on the provided main_cfdrl.py and dqn.py files available in the git eclipse of the NEMO project.

Purpose of the RL Algorithm

The RL algorithm aims to:

1. Dynamically adjust the number of replicas for workloads to optimize resource usage and performance.
2. Minimize latency and resource over-provisioning while maintaining workload efficiency.
3. Learn an optimal policy for workload orchestration through interaction with the environment.

Key Components of the RL Algorithm

1. State Representation: The state is a vector that represents the current environment, including:

- Number of replicas for a workload.
- CPU usage and target.
- RAM usage and target.
- Latency.

The function build_state_for_workload(workload_id) in main_cfdrl.py constructs this state vector for each workload by fetching metrics from shared data structures.

2. Action Space and frequency: The action space is discrete and represents possible decisions the RL agent can make:

- Increase the number of replicas.
- Decrease the number of replicas.
- Maintain the current number of replicas.

The Discrete class in dqn.py defines this action space, ensuring that actions are valid integers within a specified range.

The frequency at which the CFDRL agent takes actions is set to every 1 to 3 minutes, depending on the specific task being performed. This timing is chosen because some of the critical information provided by other system components, such as workload metrics and cluster states, is only updated once per minute. Additionally, certain actions—especially migrating workloads between clusters—can take several minutes to complete in practice. By spacing out decisions in this way, the system avoids overloading the infrastructure with frequent changes and ensures that each action is based on the most recent and reliable data available from the environment. This approach balances responsiveness with system stability and operational efficiency.

3. Reward Function: The reward function incentivizes efficient resource usage and penalizes inefficiencies. It is defined as:

$$reward = number_{of_{replicas}} - latency + \max_{0}(ram_{usage} - ram_{target})$$

It is helpful to carefully balance all the metrics (e.g., CPU usage, RAM usage, number of replicas).

Negative rewards are assigned for higher latency and more replicas. Positive rewards are given for achieving CPU and RAM targets.

4. Deep Q-Network (DQN)

The DQN is implemented in the DQN class in dqn.py. It consists of a Backbone Network: A multi-layer perceptron (MLP) that processes the state and outputs Q-values for each action and a Target Network: A copy of the backbone network used to stabilize training by providing fixed Q-value targets during updates.

The DQNTrainer class manages the training process, including computing the loss using the Bellman equation and updating the target network periodically to improve stability.

5. Replay Buffer: The ReplayBuffer class in dqn.py stores transitions (state, action, reward, next state) for experience replay. Random sampling from it allows the agent to learn from past experiences and break the correlation between consecutive samples. Storage is available with permanent mounted volumes in the Kubernetes manifest so that the replay buffer is available to any run of the RL algorithm.

6. Exploration Strategy: The project uses an epsilon-greedy strategy for exploration, implemented in the EpsilonGreedy class. The agent selects random actions with a probability epsilon, which decays over time to encourage exploitation of learned policies.

Neural Network Architecture: The DQN uses a multi-layer perceptron (MLP) as its backbone network. The architecture is defined in the MLP class in dqn.py.

- Input: State vector (as defined above).
- Hidden Layers: Configurable number of layers and neurons, with ReLU activation.
- Output: Q-values for each action.

Algorithm Workflow:

1. Initialization
   - The agent initializes its neural network, replay buffer, and exploration strategy.
2. Main Loop
   - For each workload:
   - The agent observes the current state.
   - It selects an action (e.g., adjust replicas).
   - The action is executed, and the environment returns the next state and reward.
   - The transition is stored in the replay buffer.
   - Learning: The agent samples a batch of transitions and updates the Q-value function using the Bellman equation and DQN criterion.
3. Periodic Updates
   - The target network is updated periodically.
   - Model weights are saved to disk at regular intervals.

Defining a reward function for latency: Defining a reward function for latency in this system is inherently challenging due to the nature of the workload distribution and the way latency is measured.

Latency is Linked to Node-to-Node Communication. Latency is typically a measure of the delay in communication between nodes in a distributed system. However, in this case, most replicas are deployed on the same node. As a result, there is little to no measurable latency because intra-node communication is significantly faster and often negligible compared to inter-node communication. When replicas are collocated on the same node, the latency metric does not provide meaningful feedback about the system's performance. This makes it difficult to use latency as a reliable component of the reward function.

Components CFDRL takes Information From

The CFDRL system interacts with multiple components to gather information and make decisions for workload orchestration Figure 55 and Figure 56. These interactions are facilitated through RabbitMQ (via the pika library) and HTTP API requests (via the requests library). Below is an explanation of the components CFDRL communicates with and how these communications are implemented.

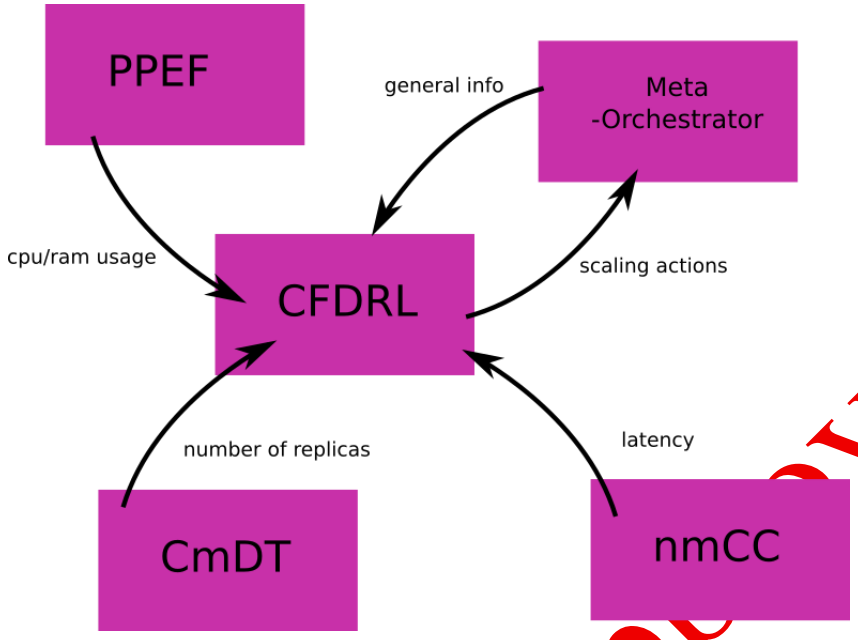


Figure 55. Inputs/outputs from CFDRL point of view


```
"$schema": "http://json-schema.org/draft-04/schema#"
"type": "object",
"properties": {
  "workload_id": {
    "type": "string"
  },
  "cluster_name": {
    "type": "string"
  },
  "number_replicas": {
    "type": "integer"
  }
},
"required": [
  "workload_id",
  "cluster_name",
  "number_replicas"
]
}
```

Figure 56. JSON Schema for MO-CFDRL RabbitMQ communication

Meta-Orchestrator (MO): The Meta-Orchestrator is responsible for managing workload orchestration decisions. It receives messages from CFDRL about the number of replicas for workloads and provides feedback.Communication : RabbitMQ is used for communication. Messages are sent to the cfdrl_mo_hs queue using RabbitMQ's basic_publish method.

CMDT (Cluster Monitoring and Decision Tracking): CMDT provides information about the current state of workloads, such as the number of replicas and the node where the workload is running.Communication: RabbitMQ is used for communication. Messages are received from the CMDT queue.

NMCC (Network Metrics Cost Calculator): NMCC provides network-related metrics, such as latency between nodes or clusters. Communication: RabbitMQ is used for communication. Messages are received from the mncc.metrics.cfdrl queue.

PPEF through Intent API: The Intent API provides workload-specific metrics, such as CPU and RAM usage targets, as well as achieved values. It also provides information about workload clusters. Communication: HTTP GET requests are sent to the Intent API using the requests library. The API response is parsed, and workload metrics are extracted and stored in shared data structures.

Shared Data Structures and Multi-threaded implementations

The program is designed to be multithreaded, enabling concurrent execution of tasks while sharing data between threads. This approach is crucial for handling multiple sources of information (e.g., RabbitMQ queues, APIs) and performing reinforcement learning computations simultaneously. Below is an explanation of how multithreading is implemented and how shared data is managed.

The program uses Python's threading module to create and manage multiple threads. Each thread is responsible for a specific task, such as consuming messages from RabbitMQ queues, calling APIs, or running the reinforcement learning loop. The threads start concurrently and run in parallel.

The information gathered from these components is stored in shared data structures, such as:

- SharedWorkloadData: Stores workload-specific metrics (e.g., replicas, latency, CPU/RAM usage).
- SharedReplayBuffer: Stores transitions for reinforcement learning.
- SharedMNCCBuffer: Stores network metrics from NMCC.

These shared data structures are protected by locks (Lock) to ensure thread-safe access.

Running the CFDRL for the Scaling task:

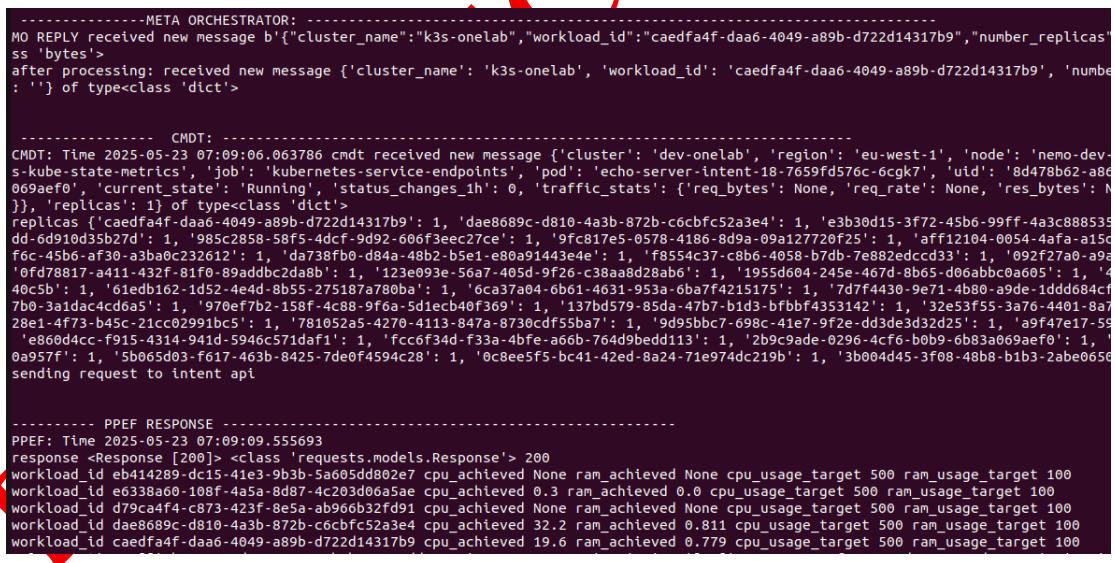


Figure 57. Messages used for Scaling

The CFDRL program was deployed and executed continuously on the OneLab cluster of the NEMO platform, where it interacted in real time with other system components such as the Meta-Orchestrator, CMDT, NMCC, and the Intent API. Over a period of 3 days, with the agent collecting one state-action-reward tuple per minute, the system accumulated a total of 4,320 transitions (3 days × 24 hours × 60 minutes). After this training period, the reinforcement learning agent achieved an average reward of -5.84, indicating improved performance compared to a random policy, which yielded an average reward of -6.54. One way to interpret this is that compared to the random policy, CFDRL almost save on average

one replica to obtain the same cpu usage. This demonstrates that the RL-based approach was able to learn and apply more effective scaling decisions than random actions in this dynamic environment.

Running the CFDRL for the Migration task:

First Note that the model used for migration was very similar to the one in scaling but adding also in the state space information about the current location of all the workloads. The action space was then the destination cluster chosen.

After learning a model specific to the migration task, we run the CFDRL with and without the migration actions and measured over a period of one day an increase of our reward from -11.2 to -10.8 thanks to the migration actions hinting at an improvement due to our model.

However, we also noticed that the variability of the reward over long period of time makes the comparisons between methods complexes.

Viewing the CFDRL and MO and Cluster IBMC interactions:

Here we report screenshots that demonstrate how an action is decided by the CFDRL and then implemented by either the Meta orchestrator or the Cluster IBMC.

1. **Migration:**

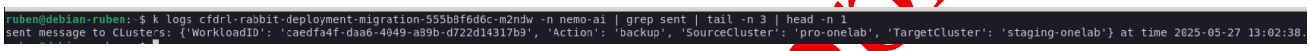


Figure 58. Log of the action taken by CFDRL and sent to the Cluster Ibmc

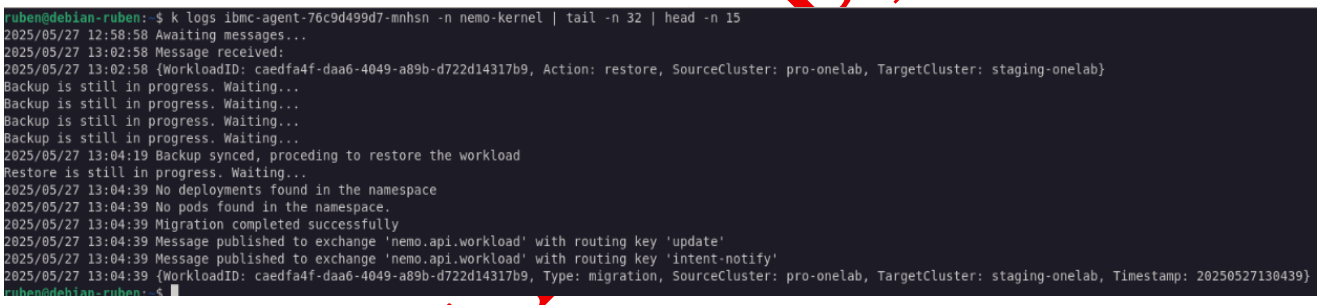


Figure 59. Message published by ibmc

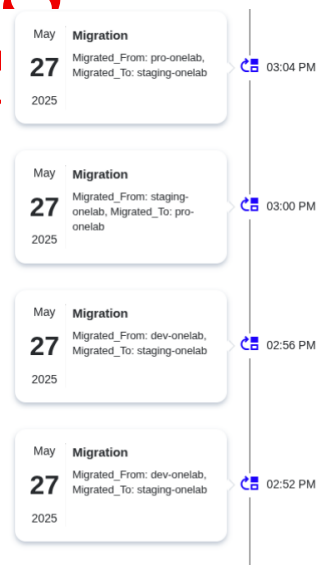


Figure 60. Log of the Migration action being implemented by the cluster IBMC

2. Scaling:

Figure 61. CFDRL decides a scaling action and sends it through Rabbit MQ to MO

Figure 62. Meta orchestrator receives the scaling action and triggers the scaling

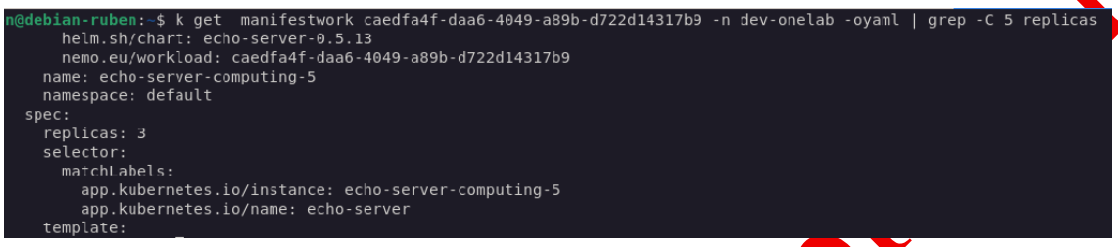


Figure 63. The number of replicas is checked and it is set to 3 as requested

5.4 NEMO workload lifecycle management

The workload lifecycle management scenario illustrates the monitoring of a workload during its lifecycle in NEMO meta-OS collecting workload intents, resources consumption, performance metrics and health status of the workload. Figure 64 shows the sequence diagram of this scenario.

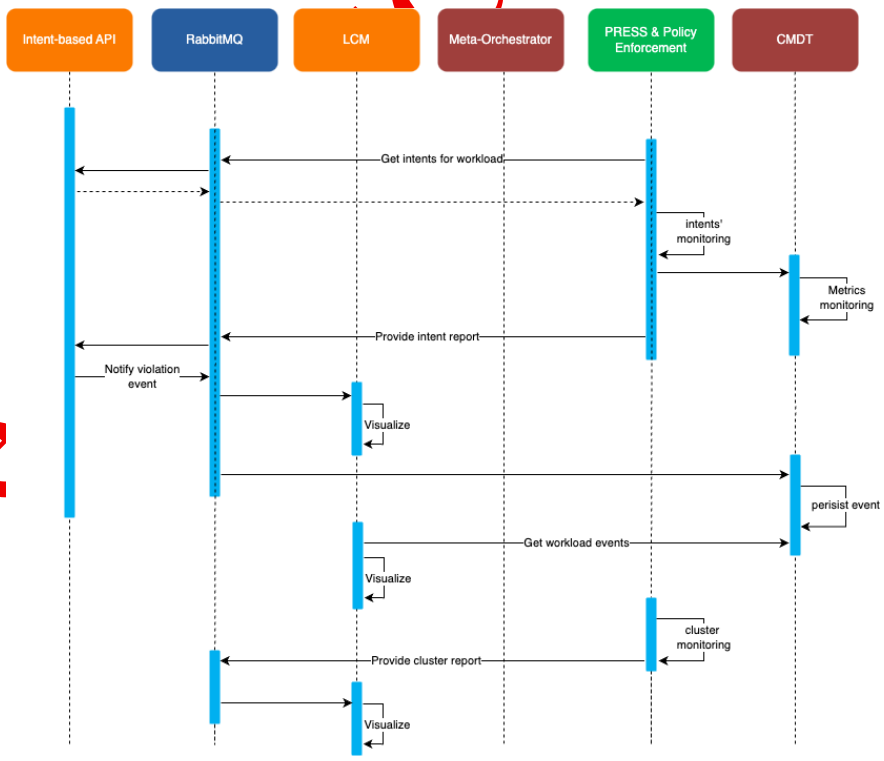


Figure 64. Workload Lifecycle Sequence Diagram

5.4.1 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

5.4.2 Results

This section outlines the step-by-step process described in the scenario above.

5.4.2.1 The NEMO user accesses the LCM UI

The NEMO user accesses the LCM interface by adding its credentials and gains access to the monitoring dashboard through NEMO Keycloak identity manager. Depending on its role, the user has access to its own workloads or workloads he manages.

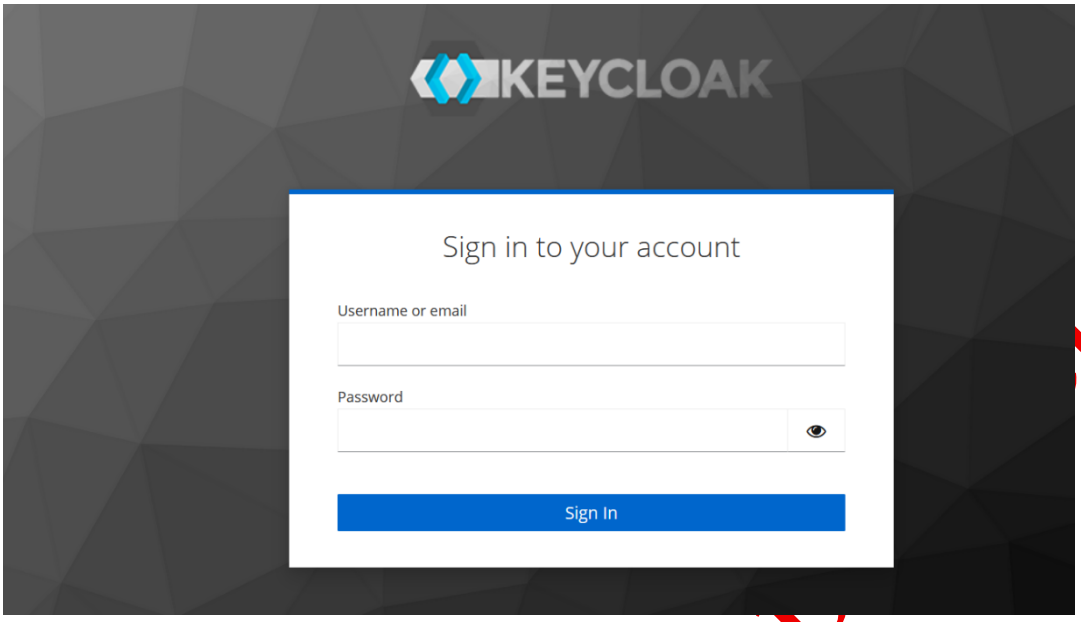


Figure 65. LCM User access

5.4.2.2 NEMO workload monitoring collects metrics that correspond to the NEMO workload (PPEF)

Within the NEMO framework there is a periodic task that evaluates the Intent target conditions for all types of workload intents that are created though the LCM UI. For example, a "DeliverComputingWorkload" Intent allows to the user to set conditions regarding the workload resource usage (CPU, RAM, disk). Figure 66 shows an example of an intent requiring for the workload to be executed in a cluster that can provide more than 2 milliseconds of CPU time.

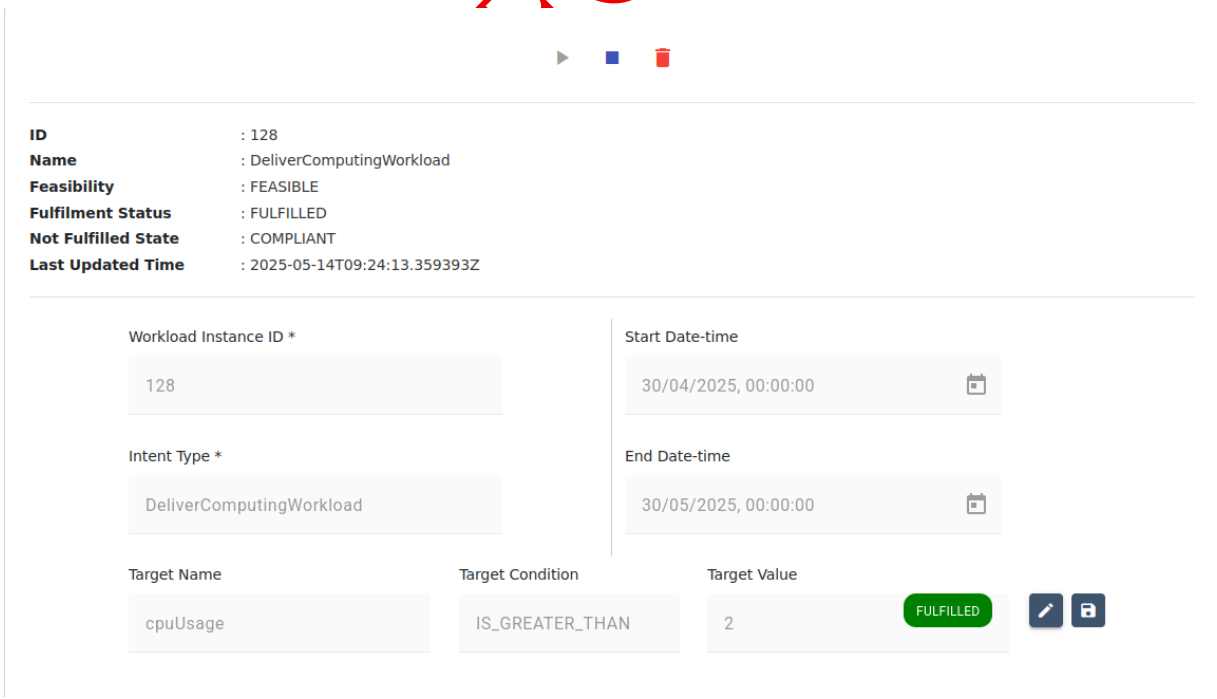


Figure 66. Example of intent conditions

Figure 67 (a) and (b) show the logs of the intent evaluator periodic task. The evaluator checks though the PPEF for the values corresponding to the intents' targets and determines if the required conditions are met. The report produced is sent through RabbitMQ for the Intent API to consume.

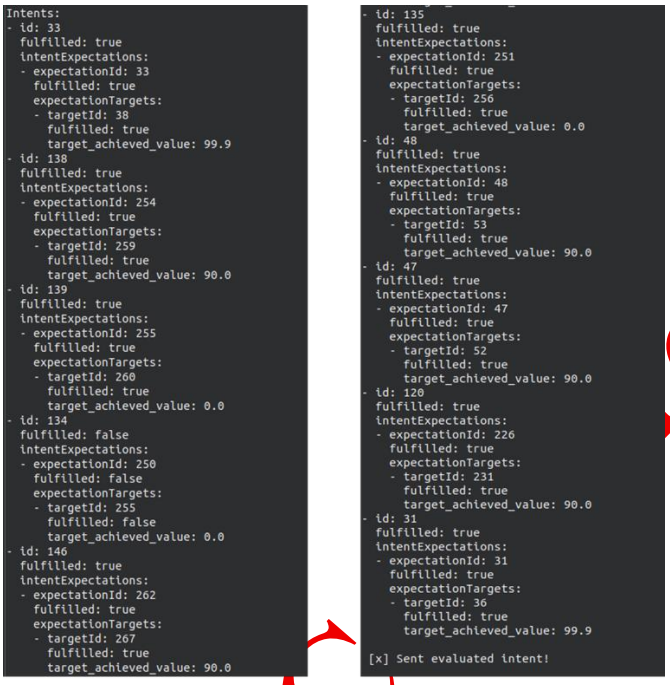


Figure 67. Intent Evaluator logs

5.4.2.3 The collected workload metrics are communicated to the Intent-API

Figure 68 demonstrates the logs of the Intent API task which is responsible for receiving the produced intent evaluation reports. The Intent API receives this report and uses it to update its intents' status ("Fulfilled" or "Not Fullfiled").

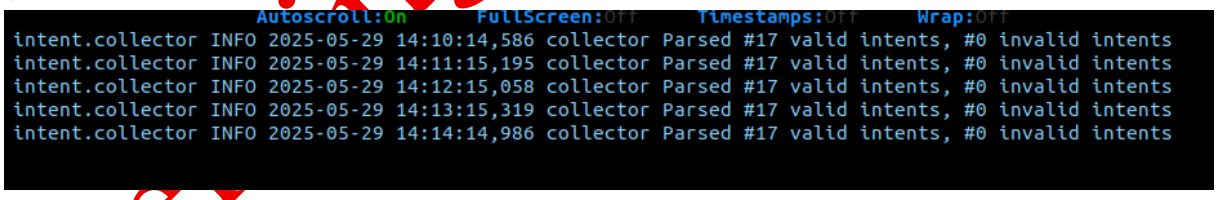


Figure 68. Intent API receives the Intent Evaluation report

5.4.2.4 The NEMO Cluster monitoring collects measurements that concern the NEMO meta-OS operated clusters (PPEF)

In NEMO a monitoring mechanism has been established that periodically communicates with PPEF to retrieve the CPU, RAM and disk usage metrics for a time window of 1 minute to keep track of the state of the NEMO meta-OS operated clusters. Figure 69 shows the logs of the monitoring task. The message includes the resource usage metrics for all NEMO clusters. This message is published to NEMO RabbitMQ, for the LCM to consume the information and display the changes in the dashboard.

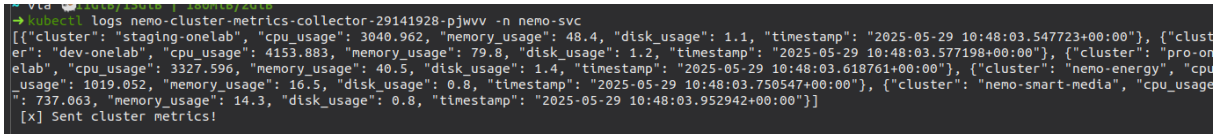


Figure 69. Cluster metrics from PPEF

5.4.2.5 The collected cluster metrics are communicated to the RabbitMQ

Figure 70 gives a closer look at the details of the message that is pushed to the RabbitMQ.

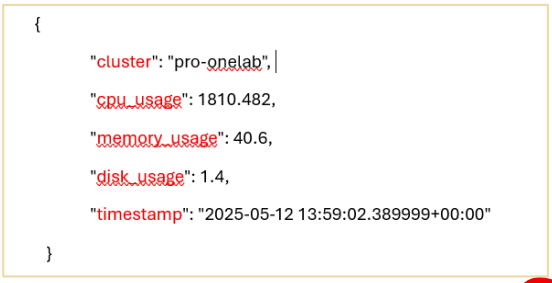


Figure 70. Cluster metrics RabbitMQ payload

5.4.2.6 NEMO workload complementary monitoring (CMDT)

The CMDT component is thoroughly presented in NEMO D2.3 and relies on information gathered by two distinct services:

- kube-state-metrics, collecting pod-related metadata,
- Linkerd and Linkerd-viz, collecting network-related traffic metrics.

Both services forward collected metrics to Prometheus/Thanos services. Finally, the CMDT components internally utilize Prometheus Query Language (PromQL) to retrieve data and expose enriched data directly through the API, as well as via RabbitMQ communication channels, Figure 71.

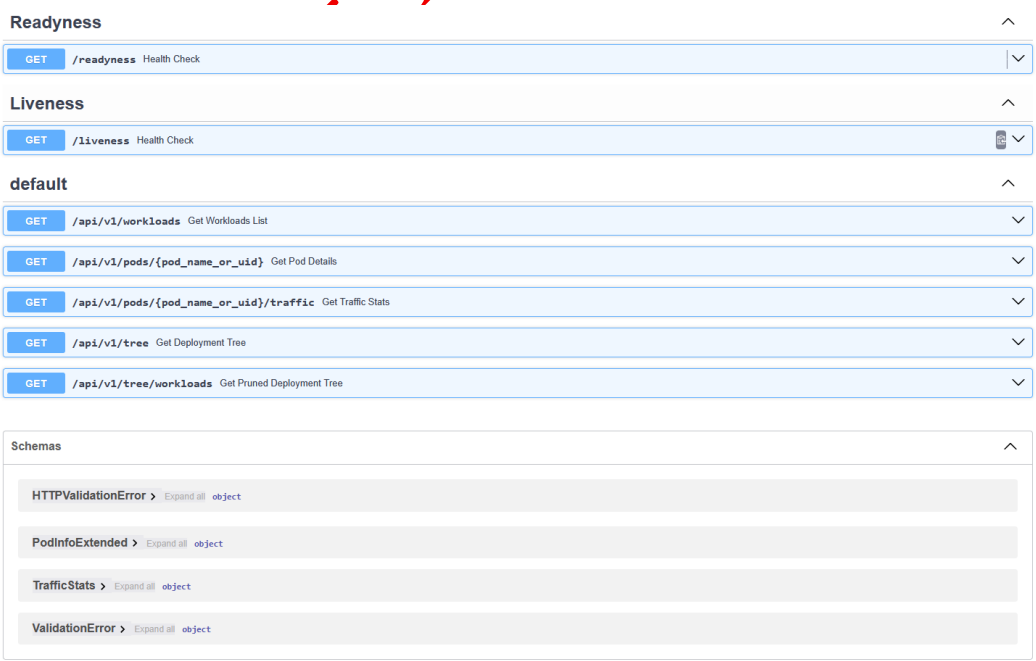


Figure 71. The CMDT component includes a SwaggerUI endpoint with up-to-date documentation and examples.

The CMDT provides the following complementary information:

1. The `*/api/v1/workloads*` endpoint provides a list of workloads on the NEMO cluster. Each NEMO workload has a UUID, and CMDT provides a list of pods from the corresponding NEMO workload. Pod information includes the pod's location (NEMO cluster, region, node, tenant), namespace, service, job, pod's name, and pod's unique ID.
2. The `*/api/v1/pods/{pod_name_or_uid}*` endpoint provides detailed information with all the data provided previous point and adds pod's current status (running, terminated, restarting, etc), number status changes within last hour (if pod was restarting), number of replicas of the same pod, and traffic summary. The traffic summary provides the incoming and outgoing bytes of network traffic, the traffic rate (requests per second), response status codes and their rates, and response times presented in percentiles (in milliseconds), Figure 72.
3. The `*/api/v1/pods/{pod_name_or_uid}/traffic*` endpoint provides a pod's detailed traffic view, where CMDT provides bytes of incoming or outgoing traffic and provides the name of external pods communicating with the corresponding pod, Figure 73.
4. The `*/api/v1/tree*` endpoint provides a tree-like representation of the relationships between all deployments, replicasets, and pods.
5. The `*/api/v1/tree/workloads*` endpoint provides a tree-like representation of all deployments, replicasets, and pods, excluding non-NEMO workload components, Figure 74.

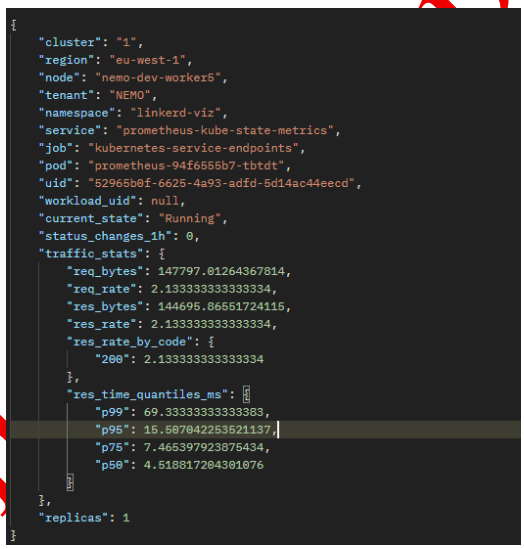


Figure 72. Detailed information about the NEMO workload pod showing its location, status, traffic, and response times.

```
{
    "prometheus-f7c5d8f9f-88xfm": {
        "": [
            1891.693333333335,
            2196.028888888889
        ],
        "linkerd-destination-76b45fb474-xd75s": [
            23.386666666666663,
            0.17333333333333334
        ],
        "linkerd-identity-7584b58569-75mnt": [
            33.04222222222222,
            0.11555555555555555
        ],
        "linkerd-proxy-injector-844b7d5fdc-4lmp4": [
            8.673333333333334,
            0.05777777777777775
        ],
        "metrics-api-6b46bb9794-mdgbx": [
            28.391111111111112,
            0.05777777777777775
        ],
        "prometheus-5b5bcf7656-xrf7n": [
            64.93777777777778,
            0.05777777777777775
        ],
        "tap-6879bb898f-tbrc5": [
            34.56,
            0.11555555555555555
        ],
        "tap-injector-6d644ff87b-f5tz9": [
            33.455555555555556,
            0.19111111111111112
        ],
        "web-5bf57d9658-8z22q": [
            30.146666666666665,
            0.11555555555555555
        ]
    }
}
```


Figure 73. Detailed information about the pod's traffic. Additionally, it displays its internal traffic, as well as inbound and outbound traffic with foreign pods.

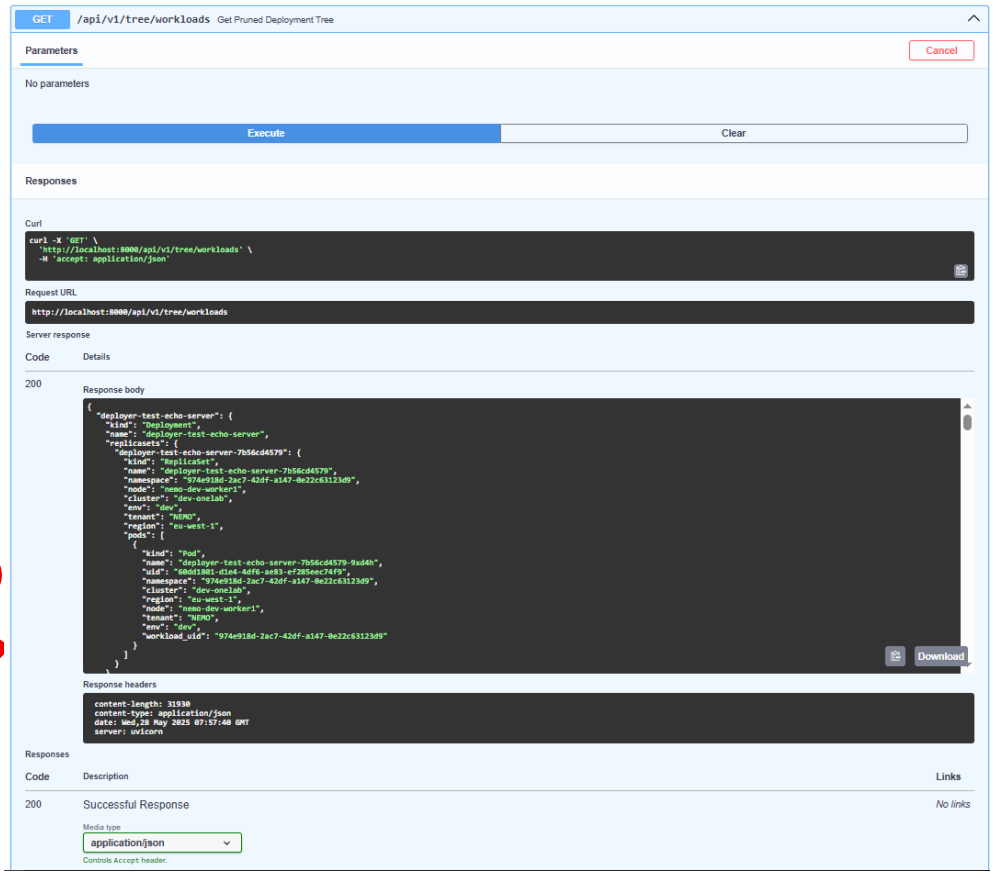


Figure 74. Tree-like representation of all deployments, replicasets, and pods, excluding non-NEMO workload components

5.4.2.7 The LCM aggregates the collected metrics and visualizes them to the NEMO user

The LCM gathers data from the previously mentioned sources and consolidates it into a unified user interface, providing a comprehensive view of the workload lifecycle. This includes historical data, near real-time performance insights, and management capabilities, Figure 75.

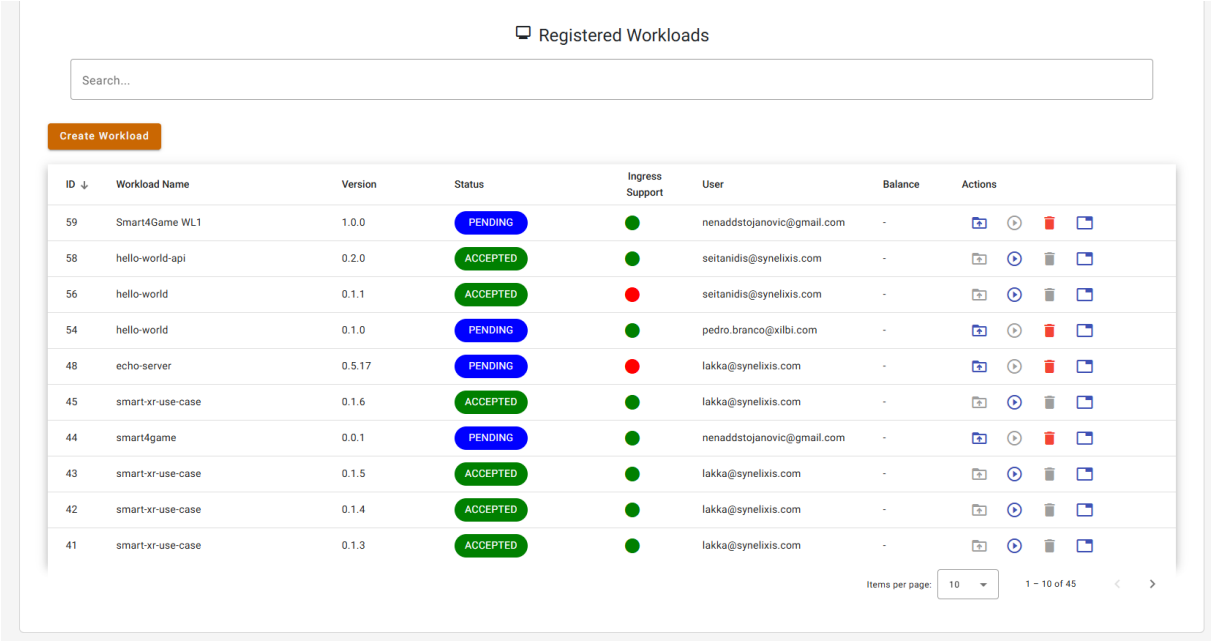


Figure 75. Manage workloads

Each workload is associated with its instances in NEMO meta-OS. The user can manage workload instances, Figure 76, and get detailed information on its activity in a status timeline, Figure 77.

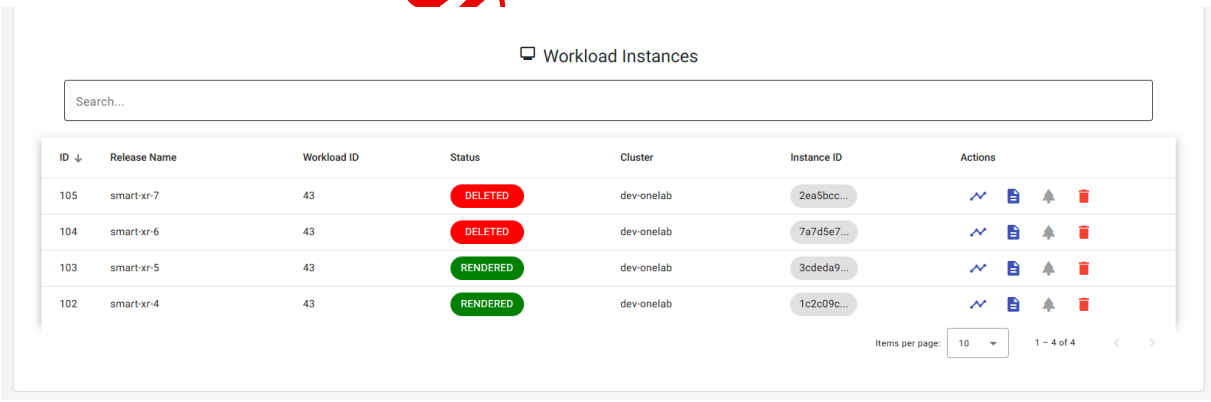


Figure 76. Manage workload instances

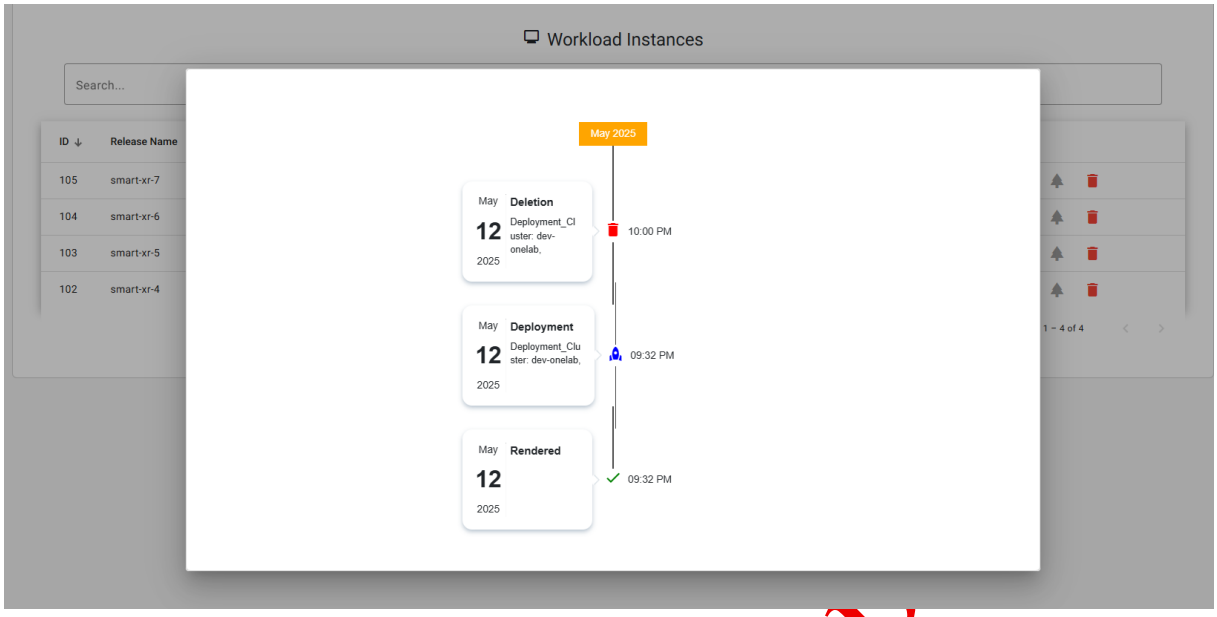


Figure 77. Workload Instance lifecycle timeline

The LCM collects information about the deployment of each workload instance, the ReplicaSets and the number of pods as it is provided by CMDT. Based on this data LCM displays the deployment tree of the workload visualizing the Deployment controller used to manage the ReplicaSets and pods in a tree-like structure, Figure 78.

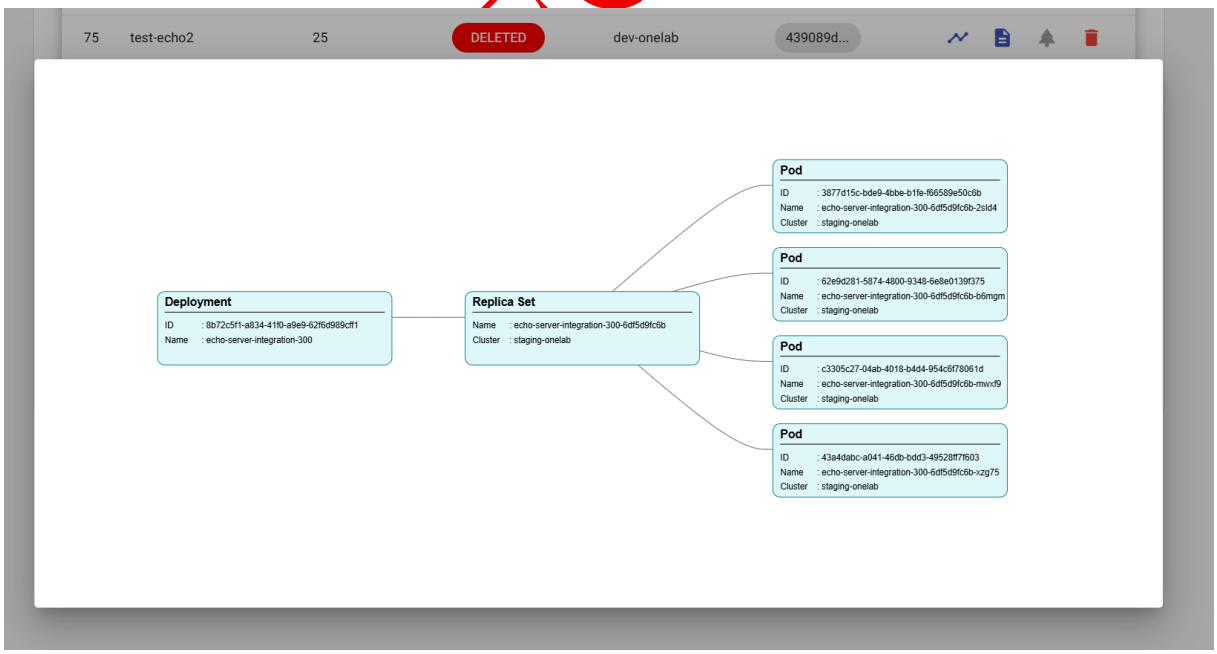


Figure 78. LCM workload deployment tree

While each workload instance is associated with one or more intents, the LCM dashboard provides information about the attached intents and their status, Figure 79. This includes intent violations as presented in the next section.

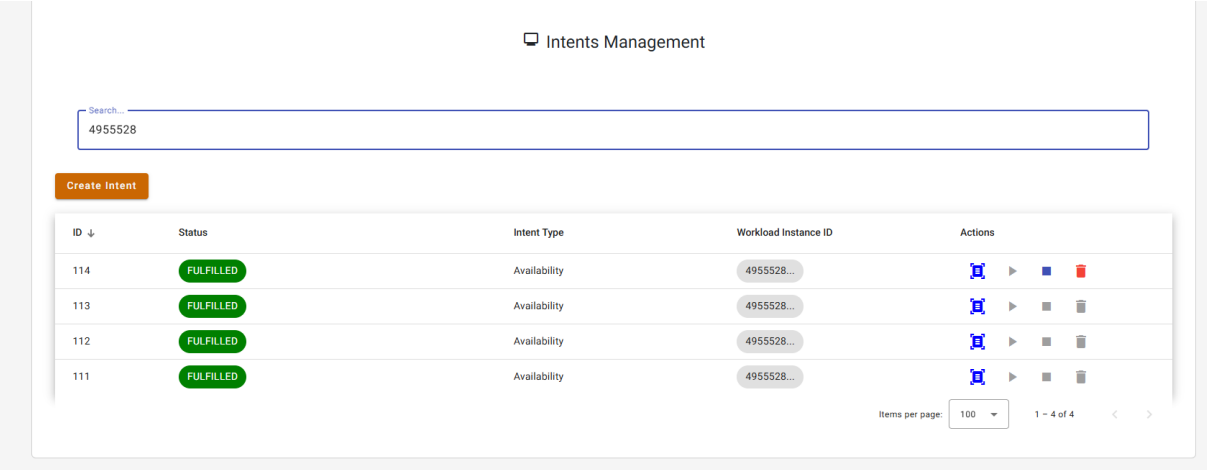


Figure 79. Workload intents management

Finally, the LCM interface shows resource provisioning monitoring and management, Figure 80, providing useful insights in cluster performance (CPU usage, memory usage, disk usage) both in historical and runtime, Figure 81.

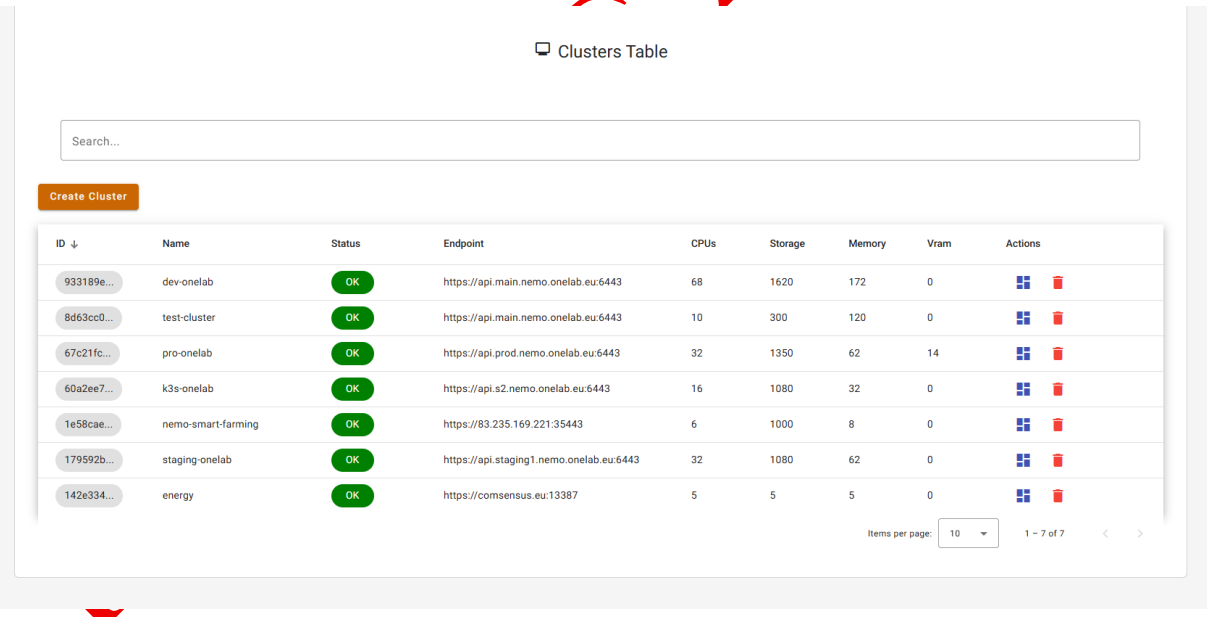


Figure 80. Resource provisioning

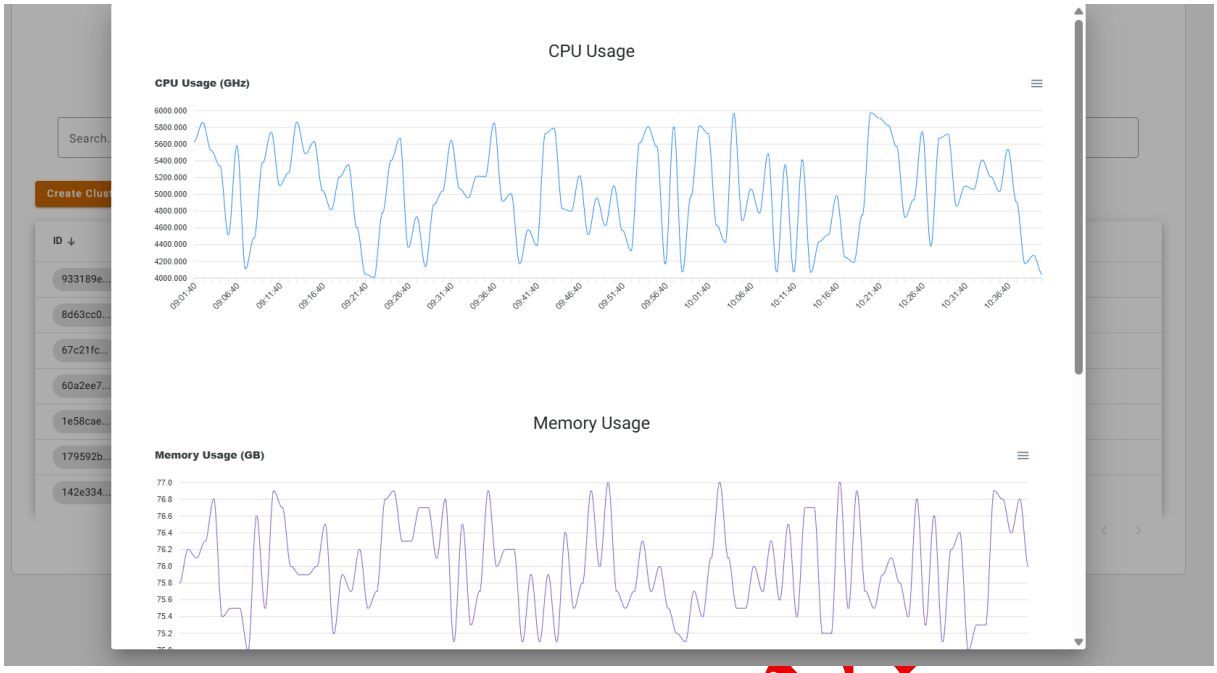


Figure 81. Cluster performance

5.4.2.8 The PPEF report intent violations to the Intent-based API

The PPEF reports the intents status for each workload instance and the LCM displays the status of each intent along with detailed information as presented in Figure 82 if the intent is fulfilled and Figure 83 if the intent has been violated.

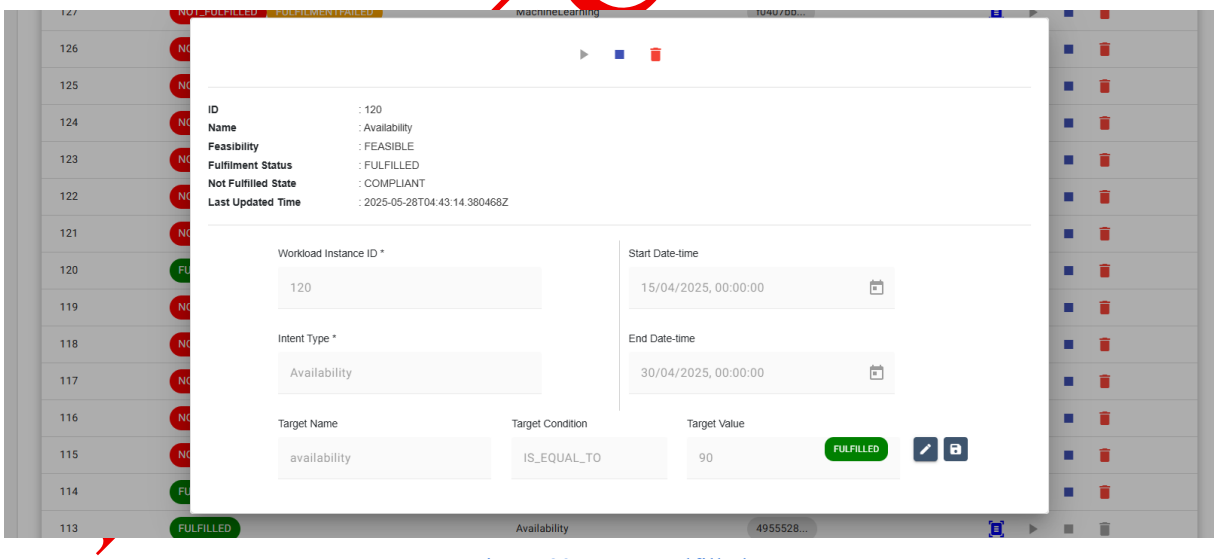


Figure 82. Intent Fulfilled

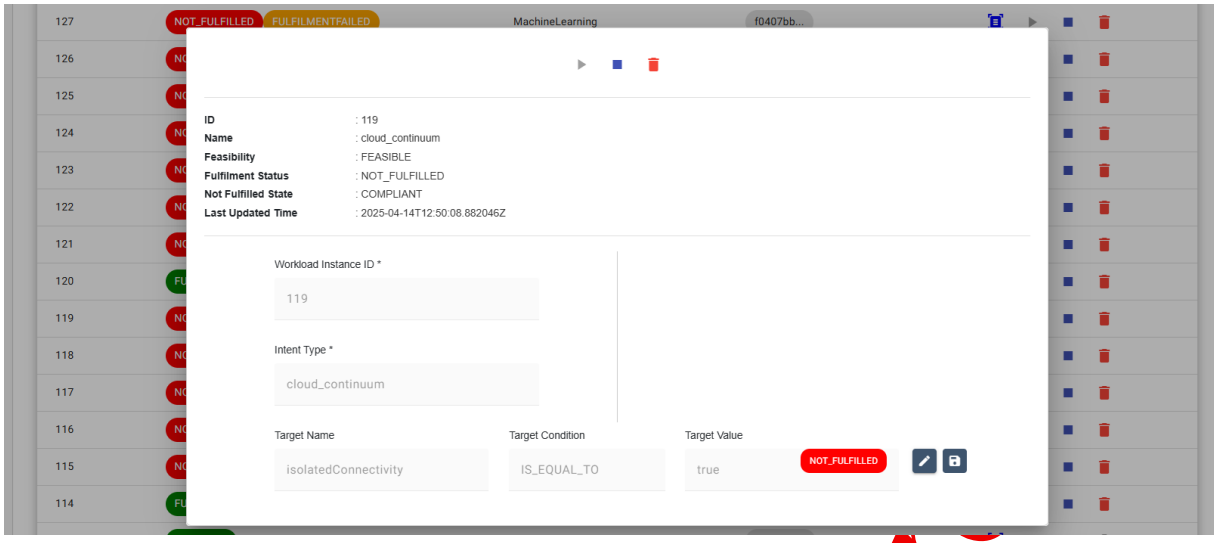


Figure 83. Intent Violation

5.5 NEMO Secure Execution Environment

The following scenario focuses on the deployment of unikernels through the Secure Execution Environment (SEE) component, starting from the meta-Orchestrator (MO). The creation, modification, and deletion of unikernels are essentially the same at this level.

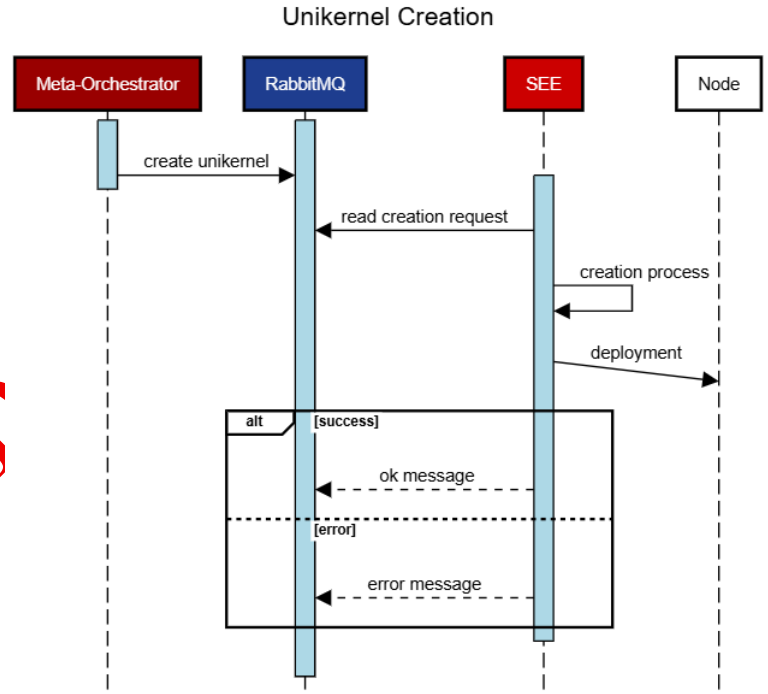


Figure 84. Unikernel creation sequence diagram

5.5.1 Verification scenario

Test 5: Unikernels	
Objective	Deploy Unikernels inside NEMO premises
Components	• Meta-Orchestrator (MO) • RabbitMQ • SEE
Features to be tested	This scenario validates the deployment of unikernel into a specific node. The SEE interface is already deployed in a cluster and communicates with the rest of the NEMO components through RabbitMQ. In particular, MO triggers this scenario to deploy the hermit technology solution (unikernel).
Test setup	SEE, RabbitMQ and MO are deployed in OneLab.
Steps	a. Meta-Orchestrator sends a *create unikernel* message to RabbitMQ. b. SEE reads the creation request from RabbitMQ. c. SEE performs the creation process internally. d. SEE deploys the created unikernel to the Node. e. If the deployment is successful, SEE sends an ok message to RabbitMQ. f. If there is an error, SEE sends an error message to RabbitMQ.

5.5.2 Results

Triggering the POST endpoint */publishToSee,* more details in Figure 85, MO send a message with the following payload, as listed in Listing 11, to the RabbitMQ queue called *nemo.see.create,* and if everything is correct the request would be similar to Listing 12. After that, the SEE interface consumes and receives that message.

```
{
   "reply_to": "see-interface-response",
   "verb": "create",
   "body": {
      "apiVersion": "apps/v1",
      "kind": "Deployment",
      "metadata": {
         "name": "hermit-httpd-app"
      },
      "spec": {
         "replicas": 1,
         "selector": {
            "matchLabels": {
               "app": "hermit-httpd-app"
            }
         },
```

```
        "template": {
          "metadata": {
            "labels": {
              "app": "hermit-httpd-app"
            }
          },
          "spec": {
            "runtimeClassName": "runh",
            "containers": [
              {
                "name": "hermit-httpd",
                "image": "ghcr.io/hermit-os/httpd:latest",
                "imagePullPolicy": "Always",
                "ports": [
                  {
                    "containerPort": 9975
                  }
                ]
              }
            ],
            "nodeSelector": {
              "runtime": "runh"
            }
          }
        }
      }
    }
}
```

Listing 11. Payload for endpoint

```
{
    "correlation_id": "4fa6082a-7479-4a5d-b137-475237d24972",
    "message": "Message published successfully",
    "status": "ok"
}
```

Listing 12. MO message after sending

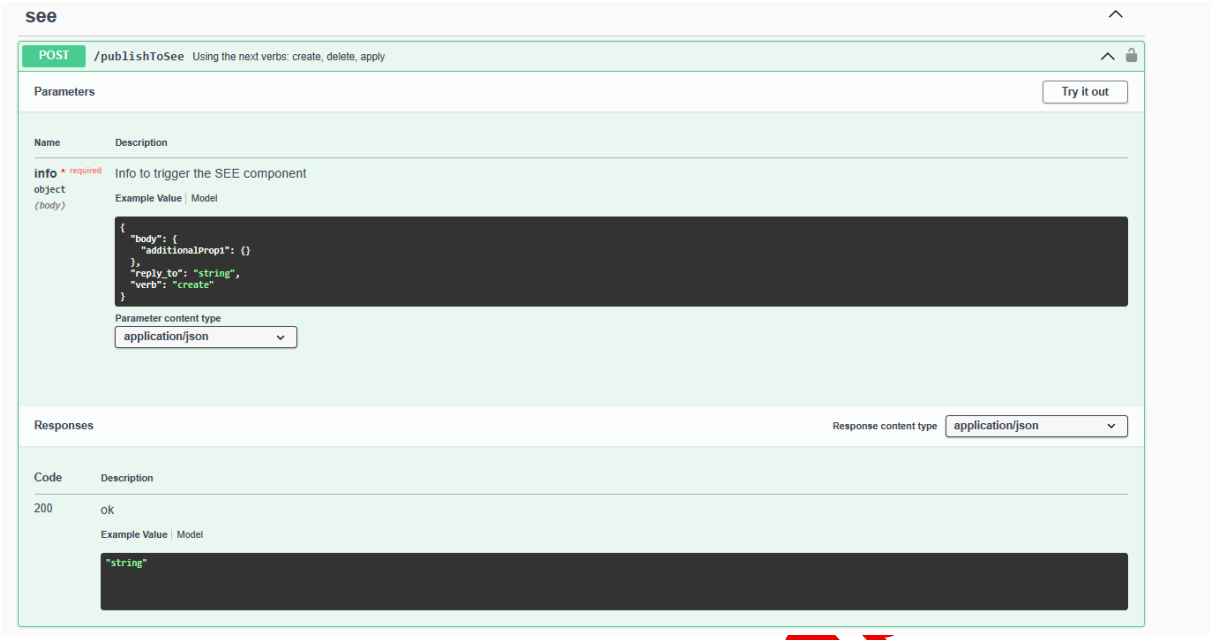


Figure 85. publishToSee endpoint

Depending on the result, the SEE generates and queues a message into RabbitMQ. If something is wrong, the message appears as in Listing 13. Otherwise, the server is sending a message with *a correlation_id* but without a payload Figure 86.


```
{
correlation_id: 4fa6082a-7479-4a5d-b137-475237d24972,
   "ErrStatus": {
      "apiVersion": "v1",
      "code": 404,
      "details": {
         "kind": "pods",
         "name": "nginx"
      },
      "kind": "Status",
      "message": "pods \"nginx\" not found",
      "metadata": {
      },
      "reason": "NotFound",
      "status": "Failure"
   }
}
```

Listing 13. SEE error response

Message 1

The server reported 0 messages remaining.

Exchange	(AMQP default)
Routing Key	see-interface-response
Redelivered	○
Properties	correlation_id: 4aba1ccb-e203-49ee-a208-9089a759b4f0 content_type: application/json
Payload 0 bytes Encoding: string	

Figure 86. SEE success message

In the end, depending on the action represented as the "verb" field in the payload endpoint (create, apply, or delete), the message is published in different queues (nemo.see.create, nemo.see.apply, or nemo.see.delete).

```
queue: nemo.see.create
       nemo.see.apply
       nemo.see.delete
actions:
    create
    apply
    delete

reply_to: see-interface-response
```

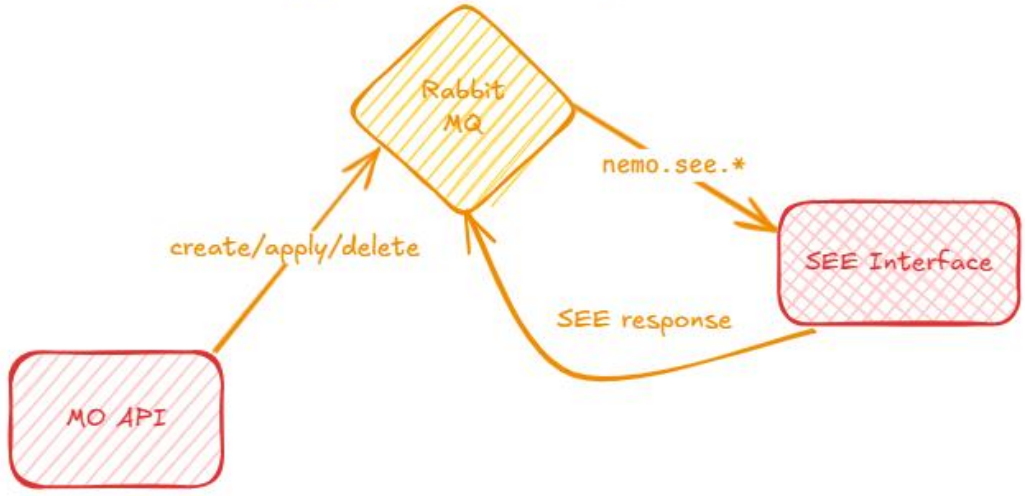


Figure 87. Schematic of the communication channels used by the MO API and the SEE

6 Conclusions

In this document the latest updates regarding the WP4 tools (LCM, IBMC, MOCA and IBA) were presented. The major highlight of this document is the set of integration workflow pipelines presented in section 5. In this document the integration pipelines across WP2, WP3 and WP4 were extended to support the complete integration and validation tests of the final NEMO meta-OS. This provides the reader with a complete and up-to-date vision of what has been developed within the NEMO framework as well as how the components interact with each other. In addition, this document is intended to be used as the handbook of NEMO for other existing and future projects, a detailed user guidelines of the NEMO meta-OS was presented.

With the completion of the NEMO components' development, a thorough testing and integration procedure took place to guarantee that the initial objectives of the project as well as the various KPIs defined have been successfully met. The work carried out within the framework of WP4 lead to a fully functional and complete toolset that provides an easy-to-use ZeroDevOps platform for managing services in the form of workload across the IoT-Edge-Cloud continuum.

Finally, the resulting final version of the NEMO meta-OS will be further validated and verified in the framework of NEMO pilots. The corresponding information will be included in deliverable D5.4 *"NEMO Living Labs use cases evaluation results. Final version"*. Moreover, D5.4 is also forseen to contain results stemming from the integration activities of the NEMO Open Call Projects in the context of NEMO meta-OS pilots.

7 References

[1] NEMO, "D4.2 - Advanced NEMO platform & laboratory testing results. Intermediate version," HORIZON - 101070118 - NEMO Deliverable Report, 2024.

[2] NEMO, "D4.1 - Advanced NEMO platform & laboratory testing results. Initial version," 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] GitLab, "The Role of AI in DevOps," 2023. [Online]. Available: https://about.gitlab.com/topics/devops/the-role-of-ai-in-devops/.

[5] Docker. [Online]. Available: https://www.docker.com/. [Accessed 23 06 2023].

[6] Flux, "Flux - the GitOps family of projects," 2023. [Online]. Available: https://fluxcd.io.

[7] Flux, "Flux multi-tenancy," 2025. [Online]. Available: https://fluxcd.io/flux/installation/configuration/multitenancy/.

[8] B. Ruecker , " The Microservices Workflow Automation Cheat Sheet: The Role of the Workflow Engine," 2020. [Online]. Available: https://camunda.com/blog/2020/02/the-microservices-workflow-automation-cheat-sheet-the-role-of-the-workflow-engine/.

[9] Kubernetes, "Good practices for Kubernetes Secrets," 2023. [Online]. Available: https://kubernetes.io/docs/concepts/security/secrets-good-practices/.

[10] Kubernetes, "Controlling Access to the Kubernetes API," 2023. [Online]. Available: https://kubernetes.io/docs/concepts/security/controlling-access/.

[11] "Kustomize - Kubernetes Native Configuration Management," [Online]. Available: https://kustomize.io/. [Accessed 12 2023].

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

[13] Keycloak, "https://www.keycloak.org," [Online]. Available: https://www.keycloak.org.

[14] "Helm," [Online]. Available: https://helm.sh/. [Accessed 12 2023].

[15] OPENAPI Initiative, "OpenAPI Specification," [Online]. Available: https://www.openapis.org. [Accessed 2024].

[16] RabbitMQ, "RabbitMQ," [Online]. Available: https://www.rabbitmq.com.

[17] Argo CD, "Introduction to ApplicationSet controller," 2023. [Online]. Available: https://argo-cd.readthedocs.io/en/stable/operator-manual/applicationset/.

[18] Argo, "Argo CD - Declarative GitOps CD for Kubernetes," 2023. [Online]. Available: https://argo-cd.readthedocs.io/en/stable/.

[19] Argo, "Argo Workflows," 2025. [Online]. Available: https://argoproj.github.io/argo-workflows/.

[20] NEMO, "D3.3 - NEMO Kernel. Final version," HORIZON - 101070118 - NEMO Deliverable Report, 2025.

[21] gRPC, "gRPC - A high performance, open source universal RPC framework," 2023. [Online]. Available: https://grpc.io/.

[22] Google Cloud, "Cloud Endpoints for gRPC," 2023. [Online]. Available: Cloud Endpoints for gRPC .

[23] R. Gancarz, "Why LinkedIn chose gRPC+Protobuf over REST+JSON: Q&A with Karthik Ramgopal and Min Chen," 2023. [Online]. Available: https://www.infoq.com/news/2023/12/linkedin-grpc-protobuf-rest-json/.

[24] Camunda, "Zeebe: Cloud Native Workflow and Decision Engine," 2023. [Online]. Available: https://camunda.com/platform/zeebe/.

[25] D. Goel, "Elevate Kubernetes Security with Zero Trust," 2023. [Online]. Available: https://d2iq.com/blog/elevate-kubernetes-security-zero-trust.

[26] A. Syed, "Zero Trust Security for Kubernetes with a Service Mesh," 2022. [Online]. Available: https://www.hashicorp.com/blog/zero-trust-security-for-kubernetes-with-a-service-mesh.

[27] KongHQ, "Kong Mesh - Modernized service mesh for development and governance," 2023. [Online]. Available: https://konghq.com/products/kong-mesh.

[28] M. Palladino, "Service Mesh vs. API Gateway: What's The Difference?," 2020. [Online]. Available: https://konghq.com/blog/enterprise/the-difference-between-api-gateways-and-service-mesh.

[29] I. Krutov, "Architecting Zero Trust Security for Kubernetes Apps with NGINX," 2022. [Online]. Available: https://www.nginx.com/blog/architecting-zero-trust-security-for-kubernetes-apps-with-nginx/.

[30] Kata Containers, "About Kata Containers," [Online]. Available: https://katacontainers.io.

[31] Kepler Contributors, "Kubernetes Efficient Power Level Exporter (Kepler)," 2023. [Online]. Available: https://sustainable-computing.io/.

[32] Pixie, "Open source Kubernetes observability for developers," 2023. [Online]. Available: https://px.dev/.

[33] The Linux Foundation, "Kubernetes," [Online]. Available: https://kubernetes.io/. [Accessed 6 2025].

[34] The Linux Foundation, "Cloud Native Computing Foundation," [Online]. Available: https://www.cncf.io/. [Accessed 6 2025].

[35] The Linux Foundation, "Falco," [Online]. Available: https://falco.org/. [Accessed 6 2025].

[36] "trivy-operator," [Online]. Available: https://github.com/aquasecurity/trivy-operator. [Accessed 6 2025].

[37] Flower Labs GmbH, "Flower," open source, [Online]. Available: https://flower.dev. [Accessed 6 2025].

[38] AsynvAPI Initiative, "Building the future of Event-Driven Architectures (EDA)," [Online]. Available: https://www.asyncapi.com. [Accessed 6 2025].

[39] Swagger, "Open API Specification," [Online]. Available: https://swagger.io/resources/open-api/. [Accessed 6 2025].

[40] Swagger, "Swagger Editor," [Online]. Available: https://swagger.io/tools/swagger-editor/download/. [Accessed 6 2025].

[41] Robot Framework Foundation, "Robot Framework," [Online]. Available: https://robotframework.org. [Accessed 6 2025].

[42] BlazeMeter, "Taurus," [Online]. Available: https://gettaurus.org. [Accessed 6 2025].

[43] Selenium, "Selenium," 2025. [Online]. Available: https://www.selenium.dev/.

[44] Istio, "The Istio service mesh," 2025. [Online]. Available: https://istio.io/latest/about/service-mesh/.

[45] Envoy, "Envoy Gateway," 2025. [Online]. Available: https://gateway.envoyproxy.io.

[46] KongHQ, "Kong Gateway," 2025. [Online]. Available: https://konghq.com/products/kong-gateway.

Annex 1-Network Intent

Intent:


```
id: '\''mncc_l2sm_1'\''
userLabel: '\''cloud_continuum'\''
intentExpectations: -
expectationId: '\''1'\''
expectationVerb: '\''DELIVER'\''
expectationObject:
objectType: '\''K8S_L2_NETWORK'\''
objectInstance: '\''spain-network-8'\''
objectContexts: -
contextAttribute: '\''name'\''
contextCondition: '\''IS_EQUAL_TO'\''
 contextValueRange: '\''spain-network-8'\''
contextAttribute: '\''providerName'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''default-slice'\''
contextAttribute: '\''domain'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''api.main.nemo.onelab.eu'\''
contextAttribute: '\''pod_cidr'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''10.1.0.0/16'\''
expectationTargets:
targetName: '\''isolatedConnectivity'\''
targetCondition: '\''IS_EQUAL_TO'\''
targetValueRange: '\''true'\''
expectationId: '\''2'\''
 expectationVerb: '\''DELIVER'\''
expectationObject:
objectType: '\''K8S_CLUSTER_CONFIG'\''
 objectInstance: '\''spain-network-8'\''
 objectContexts:
contextAttribute: '\''name'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''staging-2'\''
contextAttribute: '\''bearer_token'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange:
'\''eyJhbGciOiJSUzI1NiIsImtpZCI6IjZGbm42VVhCWVp4dVhRVEYwSGZTdnJsZEdXX1VsX0lSSGlGZzlFUGpua0kifQ
.eyJhdWQiOlsiaHR0cHM6Ly9rdWJlcm5ldGVzLmRlZmF1bHQuc3ZjLmNsdXN0ZXIubG9jYWwiLCJrM3MiXSwiZXhwIjoxN
zc3NzIyODUyLCJpYXQiOjE3NDE3MjI4NTIsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGV
yLmxvY2FsIiwianRpIjoiZWI2OTRiMjYtZDg2ZS00NDIyLWJmZTktZWVkYzgwOTg3OGI3Iiwia3ViZXJuZXRlcy5pbyI6e
yJuYW1lc3BhY2UiOiJuZW1vLW5ldCIsInNlcnZpY2VhY2NvdW50Ijp7Im5hbWUiOiJsMnNtLWNvbnRyb2xsZXItbWFuYWd
lciIsInVpZCI6IjdiYzIyMTRkLWZmOGEtNDlmYi04YzYxLTczNjE5YmM0MDhiYyJ9fSwibmJmIjoxNzQxNzIyODUyLCJzd
WIiOiJzeXN0ZW06c2VydmljZWFjY291bnQ6bmVtby1uZXQ6bDJzbS1jb250cm9sbGVyLW1hbmFnZXIifQ.aWq9F6wrwslk
nWryEWfkd7lONizjc47WWl-hyn-
WshkyjXKAqUJF86WFD5arbNIyGOglsNop92yN_fSA_FnHCv5F5S7oYXEx0BLlB3NDn3MCICAtklfWFpZ_hHlc2PW1Al8Ua
2jvogQyDhx4LFx8jR7xBJatY6JkxXKrhhLBIZCx7VVhIIEz2B0Niy9OCpU2rvsN8YCB9CfTnDwimVFtQjZ5eG6TXZr8N6n
```

```
xE1Uo68pqfoO7z34GJVWWWWJdwLlqzS5UAdLmlccx0XZutr9-9R1dFfj6hvMnxKP-LXOYLkKVidtzeqr-
X8IrkAHDuM4IqOY1_c4fdC89Un6L-y0hOg'\''
 contextAttribute: '\''api_key'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''https://api.s2.nemo.onelab.eu:6443'\''
expectationTargets:
 targetName: '\''isolatedConnectivity'\''
targetCondition: '\''IS_EQUAL_TO'\''
targetValueRange: '\''true'\''
expectationContexts:
 contextAttribute: '\''k8s_l2_network'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''spain-network-8'\''
contextAttribute: '\''namespace'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''0937609e-a6c4-43d3-b0dd-2d7c3641ee0c'\''
intentContexts:
 contextAttribute: '\''NEMO_WORKLOAD'\''
contextCondition: '\''IS_EQUAL_TO'\''
contextValueRange: '\''0937609e-a6c4-43d3-b0dd-2d7c3641ee0c'\''
 intentPriority: 1
 observationPeriod: 60
intentAdminState: '\''ACTIVATED'\'''
```

Annex 2-LCM subcomponent

```
class WorkloadLifecycleManager:
    """MO integration
    """
    def parse_lifecycle_message(self, message):
        message_type = message['type']
        instance_id = message['workloadID']
        timestamp = message['timestamp']

        try:
            workload_document_instance                                    =
WorkloadDocumentInstance.objects.get(instance_id=instance_id)
        except WorkloadDocumentInstance.DoesNotExist:
            LOG.error('Invalid workload document instance id={}'.format(instance_id))
            return

        match message_type:
            case 'deployment':
                WorkloadDocumentLifecycleEvent.objects.create(
                    workload_document_instance=workload_document_instance,

type=WorkloadDocumentLifecycleEvent.WorkloadDocumentLifecycleType.DEPLOYMENT,
                    deployment_cluster=message['targetCluster'],
                    timestamp=timestamp,
                )

                workload_document_instance.cluster_name = message['targetCluster']
                workload_document_instance.status                         =
WorkloadDocumentInstance.WorkloadDocumentInstanceStatus.DEPLOYED
                workload_document_instance.save()
            case 'migration':
                WorkloadDocumentLifecycleEvent.objects.create(
                    workload_document_instance=workload_document_instance,

type=WorkloadDocumentLifecycleEvent.WorkloadDocumentLifecycleType.MIGRATION,
                    migration_from_cluster=message['sourceCluster'],
                    migration_to_cluster=message['targetCluster'],
                    timestamp=timestamp,
                )

                workload_document_instance.cluster_name = message['targetCluster']
                workload_document_instance.save()
            case 'undeployment':
                WorkloadDocumentLifecycleEvent.objects.create(
                    workload_document_instance=workload_document_instance,

type=WorkloadDocumentLifecycleEvent.WorkloadDocumentLifecycleType.DELETION,
                    deployment_cluster=message['targetCluster'],
```

```
                timestamp=timestamp,
            )

            workload_document_instance.cluster_name = message['targetCluster']
            workload_document_instance.status                                    =
WorkloadDocumentInstance.WorkloadDocumentInstanceStatus.DELETED
            workload_document_instance.save()
        case _:
            LOG.warning('Got message_type: %s', message_type)
            return
```

Annex 3 – ServiceProviderModel

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

contract ServiceProviderModel {
    NemoTokenEstimationSetupContract public nemoTokenEstimationSetup;
    NemoFunds public nemoFunds;
    InfrastructureOwnerModel public infrastructure;

    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;
    mapping(string => uint256) public greenEnergyRewards;

    constructor(
        address _nemoTokenEstimationSetupContractAddress,
        address _nemoFundsAddress,
        address _infrastructure
    ) {
        nemoTokenEstimationSetup = NemoTokenEstimationSetupContract(
            _nemoTokenEstimationSetupContractAddress
        );
        nemoFunds = NemoFunds(_nemoFundsAddress);
        infrastructure = InfrastructureOwnerModel(_infrastructure);

        // Initiate greenEnergyRewards mapping
```

```
    greenEnergyRewards["0%"] = 0;
    greenEnergyRewards["20%"] = 20000000;
    greenEnergyRewards["40%"] = 40000000;
    greenEnergyRewards["80%"] = 80000000;
    greenEnergyRewards["100%"] = 100000000;
}

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 removeServiceProviderInfo(string memory serviceId) public {
    require(
        nemoFunds.isCustomerRegistered(serviceId),
        "The customer is not registered!"
    );

    nemoFunds.removeCustomer(serviceId);
}
function enabledIngress(string memory serviceId) public {
    require(
        nemoFunds.isCustomerRegistered(serviceId),
        "The customer is not registered!"
    );

    nemoFunds.enabledIngress(serviceId);
}
```

```
    function computeCpuCredits(
        ServiceMetrics memory _metrics,
        uint256 _regionalCpuCosts,
        uint256 _cpuBaseRate
    ) public pure returns (uint256) {
        // CPU
        uint256 _cpuTokens = 0;
        uint256 _cpuUsage = _metrics.cpuUsage * 10 ** 3;

        if (_cpuUsage > _regionalCpuCosts) {
            _cpuTokens =
                ((_cpuUsage) / _metrics.clusterCpuUsage) *
                10 ** 8 +
                (_cpuBaseRate * 10 ** 5);
        }

        return _cpuTokens;
    }

    function computeMemoryCredits(
        ServiceMetrics memory _metrics,
        uint256 _regionalMemoryCosts,
        uint256 _memoryBaseRate
    ) public pure returns (uint256) {
        // RAM
        uint256 _ramTokens = 0;
        uint256 _ramUsage = _metrics.memoryUsage * 10 ** 3;

        if (_ramUsage > _regionalMemoryCosts) {
            _ramTokens =
                (_ramUsage / _metrics.clusterMemoryUsage) *
                10 ** 8 +
                (_memoryBaseRate * 10 ** 5);
        }

        return _ramTokens;
    }
    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);

    (
        uint256 _cpuBaseRate,
        uint256 _memoryBaseRate,
        string memory _greenEnergy
    ) = infrastructure.getInfrastructureCostsAndGreenEnergy(
            _metrics.clusterId
        );

    uint256 _tokens = 0;
    uint256 _greenEnergyReward = 0;

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

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

    _tokens =
        computeCpuCredits(_metrics, _regionalCpuCosts, _cpuBaseRate) +
        computeMemoryCredits(
            _metrics,
            _regionalMemoryCosts,
            _memoryBaseRate
        );

    if (_highDemand) {
        _tokens += _highDemandCost;
    }

    // Check infrastructure green energy percentage and charge less
    _greenEnergyReward = greenEnergyRewards[_greenEnergy];

    if (_tokens >= _greenEnergyReward) {
        _tokens -= _greenEnergyReward;
    }

    _tokens = _tokens / 1000;

    nemoFunds.makeTransaction(
```

```
            _metrics.serviceId,
            _metrics.clusterId,
            _tokens
        );

        emit ServiceComputeTokens(
            _metrics.serviceId,
            _metrics.clusterId,
            _cpuUsage,
            _ramUsage,
            _tokens
        );
    }
}
```

Annex 4-Intent for network connectivity request

Intent for network connectivity request. L2S-M network request example.

Intent:

```
id: 'mncc_l2sm_1'
userLabel: 'cloud_continuum'
intentExpectations:
 - expectationId: '1'
   expectationVerb: 'DELIVER'
   expectationObject:
     objectType: 'K8S_L2_NETWORK'
     objectInstance: 'spain-network'
     objectContexts:
       - contextAttribute: 'name'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: 'spain-network'
       - contextAttribute: 'providerName'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: 'default-slice'
       - contextAttribute: 'domain'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: 'api.main.nemo.onelab.eu'
       - contextAttribute: 'pod_cidr'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: '10.1.0.0/16'
   expectationTarget:
    - targetAttribute: 'isolatedConnectivity'
      targetCondition: 'IS_EQUAL_TO'
      targetValueRange: 'true'
 - expectationId: '2'
   expectationVerb: 'DELIVER'
   expectationObject:
     objectType: 'K8S_CLUSTER_CONFIG'
     objectInstance: 'spain-network'
     objectContexts:
       - contextAttribute: 'name'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: 'staging-1'
       - contextAttribute: 'bearer_token'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: 'eyJhbGciO…
       - contextAttribute: 'api_key'
         contextCondition: 'IS_EQUAL_TO'
         contextValueRange: 'https://api.staging1.nemo.onelab.eu:6443'
   expectationTarget:
```

```
    - targetAttribute: 'isolatedConnectivity'
      targetCondition: 'IS_EQUAL_TO'
      targetValueRange: 'true'
    expectationContexts:
    - contextAttribute: 'k8s_l2_network'
      contextCondition: 'IS_EQUAL_TO'
      contextValueRange: 'spain-network'
    - contextAttribute: 'namespace'
      contextCondition: 'IS_EQUAL_TO'
      contextValueRange: 'cbcb208a-d535-434b-bb35-217a64bd516b'
  - expectationId: '3'
    expectationVerb: 'DELIVER'
    expectationObject:
      objectType: 'K8S_CLUSTER_CONFIG'
      objectInstance: 'spain-network'
      objectContexts:
        - contextAttribute: 'name'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'staging-2'
        - contextAttribute: 'bearer_token'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'eyJhbGc…`
        - contextAttribute: 'api_key'
          contextCondition: 'IS_EQUAL_TO'
          contextValueRange: 'https://api.s2.nemo.onelab.eu:6443'
    expectationTarget:
      - targetAttribute: 'isolatedConnectivity'
        targetCondition: 'IS_EQUAL_TO'
        targetValueRange: 'true'
    expectationContexts:
      - contextAttribute: 'k8s_l2_network'
        contextCondition: 'IS_EQUAL_TO'
        contextValueRange: 'spain-network'
      - contextAttribute: 'namespace'
        contextCondition: 'IS_EQUAL_TO'
        contextValueRange: 'nemo-workload'
  intentContexts:
    - contextAttribute: 'NEMO_WORKLOAD'
      contextCondition: 'IS_EQUAL_TO'
      contextValueRange: 'cbcb208a-d535-434b-bb35-217a64bd516b'
  intentPriority: 1
  observationPeriod: 60
  intentAdminState: 'ACTIVATED'
```