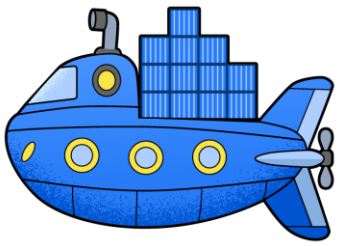


NEMO

Next Generation Meta Operating System

D4.1 Integration guidelines & initial NEMO integration

Document Identification			
Status	Final	Due Date	31/12/2023
Version	1.0	Submission Date	19/01/2024

Related WP	WP4	Document Reference	D4.1
Related Deliverable(s)	D1.1, D1.2, D2.1, D3.1	Dissemination Level (*)	PU
Lead Participant	SYN	Lead Author	Terpsi Velivassaki
Contributors	SYN, INTRA, AEGIS, SPACE, ATOS, MAG, ENG, ESOFT, SU	Reviewers	Theo Kakardakos, NOVO Dimitris Siakavaras, ICCS

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

Document Information

List of Contributors	
Name	Partner
Aitor Alcázar-Fernández	ATOS
Rubén Ramiro	ATOS
Enric Pages-Montanera	ATOS
Dimitrios Skias	INTRA
Panagiotis. Karkazis	MAG
Astik Samal	MAG
Nikos Drosos	SPACE
Emmanouil Bakiris	SPACE
Antonis Gonos	ESOFT
Theodore Zahariadis	SYN
Terpsi Velivassaki	SYN
Spyros Vantolas	AEGIS
Hassane Rahich	SU

Document History			
Version	Date	Change editors	Changes
0.1	13/11/2023	T. Velivassaki (SYN)	Table of Contents
0.2	08/12/2023	R. Ramiro (ATOS), D. Skias (INTRA), T. Velivassaki, Th. Zahariadis (SYN), H. Rahich (SU)	Contributions to NEMO integration approach, prototype and guidelines, Intent-based Migration, the integration infrastructure
0.3	18/12/2023	R. Ramiro (ATOS), D. Skias (INTRA), T. Velivassaki (SYN), P. Karkazis (MAG), A. Samal (MAG)	Contributions to the NEMO integration guidelines, DevSecOps approach & plan, prototype, Intent-based API, MOCA
0.4	29/12/2023	S. Vantolas (AEGIS), N. Drosos (SPACE), E. Bakiris (SPACE), A. Gonos (ESOFT), T. Velivassaki (SYN)	Contributions to Lifecycle Management, updates to Intent-based API
0.5	12/01/2024	P. Karkazis (MAG), A. Samal (MAG), N. Drosos, (SPACE), E. Bakiris (SPACE), T. Velivassaki (SYN)	Updates to Lifecycle Management, V&V, document consolidation and integration; Ready for peer-review
0.5.1	17/01/2024	D. Siakavaras (ICCS)	Peer review
0.5.2	18/01/2024	Th. Kakardakos (NOVO)	Peer review

0.6	18/01/2024		Document enhancements based on peer review; Document finalization
0.7	19/01/2024	Rosana Valle (ATOS)	Quality Check
1.0	19/01/2024	Rosana Valle (ATOS)	FINAL VERSION TO BE SUBMITTED TO EC

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	T. Velivassaki (SYN)	18/01/2024
Quality manager	R. Valle Soriano (ATOS)	19/01/2024
Project Coordinator	E. Pages (ATOS)	19/01/2024

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
AAA	Authentication, Authorization, and Accounting
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Delivery
CFDRL	Cybersecure Federated Deep Reinforcement Learning
CLI	Command Line Interface
CMDT	Cybersecure Microservices' Digital Twin
CI	Continuous Integration
CIS	Center for Internet Security
CLI	Command-line Interface
CMDT	Cybersecure Microservices' Digital Twin
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
DLT	Distributed Ledger Technology
Dx.y	Deliverable number y belonging to WP x
E2E	End-to-End
EC	European Commission
FL	Federated Learning
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
IaC	Infrastructure-as-Code
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
KPI	Key Performance Indicator
LCM	Life-Cycle Manager
MANO	Management and Orchestration
meta-OS	Meta-Operating System
ML	Machine Learning
mNCC	Meta Network Cluster Controller
MO	Meta-Orchestrator
MOCA	Monetization and Consensus-based Accountability
MQTT	Message Queuing Telemetry Transport
OS	Operating System

OT	Operational technology (OT)
P2P	Peer-to-Peer
PoC	Proof of Concept
PPEF	PRESS & Policy Enforcement Framework
PRESS	Privacy, data pRotection, Ethics, Security & Societal
RaaS	Resources-as-a-Service
RAM	Random Access Memory
RBAC	Role-Based Access Control
RL	Reinforcement Learning
SBOM	Software Bill of Materials
SDK	Software Development Kit
SDLC	Software Development Life Cycle
SEE	Secure Execution Environment
SLA	Service Level Agreement
SLO	Service Level Objective
SUT	System Under Test
TEE	Trusted Execution Environment
TSN	Time Sensitive Networks
V&V	Validation & Verification
VIM	Virtual Infrastructure Manager
WP	Work Package
YAML	Yet Another Markup Language

Executive Summary

The highly distributed and diverse landscape in the envisioned meta-OS in NEMO urges for component-based microservices architectures for the development of both applications and the NEMO framework per se. In such diverse environments, software integration can be challenging even despite the adoption of Zero-everything practices in GitOps. Moreover, as the diversity and required speed of delivery urge for automating GitOps processes to the extent possible, validation tasks must be also considered and integrated in the software lifecycle. The present document aims to minimize the integration and validation challenges, suggesting development and integration guidelines to NEMO developers to be adopted during the complete software lifecycle.

First, conventions related to design, development and integration urge developers to follow modular cloud- and edge-native design and ensure the NEMO specifications on interfaces and data models are coherent and followed. Moreover, containerization and automation techniques are suggested to address consistent GitOps processes. In addition, NEMO gives due emphasis on expanding the NEMO functionality and potential through the open-source community, releasing NEMO components in the public code repository of Eclipse Research Labs. Last, but not least, strong focus on security is considered across the full lifecycle and stack of the NEMO framework.

The NEMO integration strategy is thoroughly presented, providing insights on the implementation of DevSecOps within NEMO. NEMO adopts continuous integration (CI) and continuous deployment (CD) through GitLab and automation CD tools. NEMO relies on OneLab infrastructure of the Sorbonne University for the provision of its integration, qualification, and production environment. Together with the integration and deployment processes, validation tests and scans are injected in different steps of the NEMO framework lifecycle. NEMO follows a comprehensive validation and verification (V&V) plan, with diverse types of tests from development till runtime, addressing unit, system, functional and security tests. Last, but not least, the Integration and V&V plan, following an agile approach is also presented.

The document presents the integrated view of the NEMO framework. Concretely specified interactions and workflows in NEMO will guide effective integration from both functional, technical and logical point of view. This report presents high-level interactions and functional relevance of current NEMO developments, as well as early integration activities.

Towards adoption of NEMO by envisaged end users, including both application providers and infrastructure owners, NEMO develops a Service Management middleware. It comprises four components which address the NEMO exposure through API and SDK, ZeroOps deployment and Lifecycle Management, as well as monetization and accountability of NEMO-enabled applications and resources.

Future work includes integration of NEMO components in a more mature first version, the delivery and execution of validation tests and further development of components towards third-party integration. This work is planned to be reported in deliverable document D4.2 “Advanced NEMO platform & laboratory testing results. Initial version”, due in the last quarter of 2024.

1 Introduction

NEMO aspires to meet the ambitious goal of developing a meta-Operating System (metaOS), which will democratize the usage of resources anywhere in the IoT, edge and cloud continuum, while enabling the delivery of innovative applications of diverse capabilities' requirements through any device at everyone's hand. This vision aims to be materialized in NEMO, without compromising cybersecurity, trust and privacy requirements, while enabling business development and prosperity on top of this innovative ecosystem.

The NEMO functionalities come to reality through a comprehensive and modular architecture, presented in the deliverable document "D1.2 - NEMO meta-architecture, components and benchmarking. Initial version" [1]. The architecture assigns NEMO components with tasks, so that they can, both individually and as a whole, vindicate any claim in the NEMO vision. The NEMO components will be implemented within the framework of work packages (WPs) WP2 and WP3, while "WP4: NEMO DevZeroOps Service Management Space" is responsible for automated delivery of the integrated NEMO components. The present document is the first deliverable of WP4.

1.1 Purpose of the document

Development, integration and testing activities following a structured methodology is key to successful project delivery. A well thought-out solution that aims to be materialized in a system or a platform or even a service needs to be supported by coherent development, integration and testing approaches, coordinated schedule and effective inter-process communication. This document presents the development and integration guidelines for the NEMO project. Moreover, the integrated view of the NEMO meta-OS is presented, highlighting interactions among the NEMO components. Based on these development and integration guidelines, as well as the specified NEMO interactions, a comprehensive integration, validation and verification strategy is defined for the NEMO project. Moreover, early integration activities have been already fruitful, and early proof of concept results are presented.

Furthermore, since appropriate infrastructure, tools and frameworks are key enablers for the integration and testing activities, the present document introduces those within the project's CI/CD lifecycle.

Considering the overall concept of the proposed meta-OS, the adoption potential is technically relevant to interaction and integration points with third parties. NEMO aims to address this challenge through a Service Management middleware, as presented in this document. Through this, NEMO provides programmatic access to application developers in a friendly way, aiming to tailor meta-OS offered services to techniques and practices already familiar to developers, thus introducing minimum deviation from existing pipelines. Following NEMO API calls, full-stack automation is leveraged in the NEMO meta-OS, enabling Zero-Ops deployment and configuration for the meta-OS operations. Beyond usability, monetization of resource and workload related operations are key for timely time to market exploitation. NEMO inherently supports the development of business models to allow exploitation and monetization of supported services. This functionality is part of the Service Management middleware, as well.

This document (D4.1) is the first iteration of the three deliverables in the context of WP4 and aim to describe the integration and validation activities of the NEMO project. The other two deliverables are D4.2 "Advanced NEMO platform & laboratory testing results. Initial version", due in the last quarter of 2024, and finally D4.3 "Advanced NEMO platform & laboratory testing results. Final version", which will be produced in the second quarter of 2025.

1.2 Relation to other project work

The integration and testing strategy act as the driver of the development process. Thus, this document is strongly connected with all the technical WPs (WP3, WP4). Furthermore, the work presented in this document strongly relates to WP1 activities, as it considers the technical specifications arising from the requirements' elicitation process and the architectural specifications. Moreover, the approach for the validation and verification processes within the meta-OS, as defined in WP1, is technically applied as described in the present document. In addition, the platform integrated view and current prototype implementations will be applied and tailored to each of the NEMO trials within WP5. The document provides valuable feedback for the project outcomes' communication, dissemination and exploitation, especially regarding the definition of the NEMO exploitable outcomes and their unique value proposition, as well as the development of new business models. Last, but not least, the document reports technical options and prototype functionalities, which are meant to be used and extended by third parties joining the project through the Open Calls.

1.3 Structure of the document

The remainder of this report is organized as follows.

Section 2 provides the NEMO development & integration guidelines.

Section 3 presents the DevSecOps approach and how security is injected in the project's CI/CD pipeline. With a focus on open science practices, this section presents the open-source NEMO repository on the GitLab instance of Eclipse Research Labs. Based on this, CI/CD pipelines are enabled, supporting ZeroOps deployments for the NEMO components, as presented in this section, too. Moreover, the Integration & Validation plan within the NEMO lifecycle is outlined.

Section 4 presents the envisioned integrated NEMO framework. The section elaborates on the functionality supported in the preliminary (v0) version of NEMO and presents early integrated activities.

Section 5 presents the NEMO components comprising the Service Management middleware and aimed for integration of and adoption by third parties.

Last, Section 6 draws conclusions and provides next steps.

2 NEMO development & integration guidelines

This section outlines design and implementation approaches to guide the NEMO development and integration activities.

2.1 Cloud- & edge-native design

Edge computing, at its core, provides compute, networking, and storage capabilities at the network's border, closer to the source of the data[1]. This comes with significant benefits in the use of resources, but also in network delivery, bringing reduced latency and greater exploitation of available bandwidth, which gives room to more resilient and network-intensive applications. At the same time, it gives the possibility for data to remain local and support data sovereignty.

Compared to the cloud environment, edge computing is quite diversified. The edge environment intrinsically has limitations. Edge devices are highly diverse, but they are often battery powered and thus carry less capable processors. Moreover, many edge devices must interact with legacy equipment; they usually feature ports not found on Information Technology (IT) equipment and leverage protocols specific to operational technology, which lead to very different edge devices across sites or applications. Thus, edge computing implies distributed data processing across diverse devices and/or sites, which, generally speaking, happens at small scale. On the other hand, cloud is a quite different computing environment. The cloud offers on-demand availability of resources. You can create new virtual machine instances, add network capacity, or change the network topology anytime. Cloud resources are naturally limited, but those limits are so high that most users will never bump into them, so cloud computing/network resources are practically unlimited. Overall, the cloud is heterogeneous, centralized, and large-scale, while the edge is heterogeneous, distributed, and small-scale.

Pushing the boundaries in modern software development and integration suggests adhering to the Cloud Native approach. As defined by Cloud Native Computing Foundation (CNCF), the "Cloud Native" definition[2] is expressed as:

"*Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.*

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil."

Edge-native applications share several characteristics with cloud native applications, as both are based on microservices, which expose endpoints which enable loosely coupled service composition. Moreover, both cloud- and edge-native design allow adopting DevOps for continuous integration of such applications. However, edge-native applications can be different from the cloud-native ones, as continuous deployment might not be applicable for time critical edge applications. Edge-native applications are often targeted for given environments and domains, which means that might be coupled to Operational technology (OT) processes, but also quite dependent on the installed infrastructure – often requiring a considerable capital investment. This affects both the computing/power/network capabilities exposed for application delivery, but also the application lifespan, which in this case is desired to be long enough. NEMO urges for consideration of edge device heterogeneity, constrained capabilities, their potential mobility and also potential infrequent need for updates in the design of application and services delivering or running on top of the meta-OS.

[1] https://opensource.com/article/23/3/what-edge-native-application
[2] https://github.com/cncf/toc/blob/main/DEFINITION.md

The abovementioned considerations with respect both to the resulting platform, but also to the supporting cloud infrastructure that enables the development, integration and testing activities, are in line with the NEMO project objectives that are addressed through WP4 (T4.4) for the integration and testing of the NEMO platform.

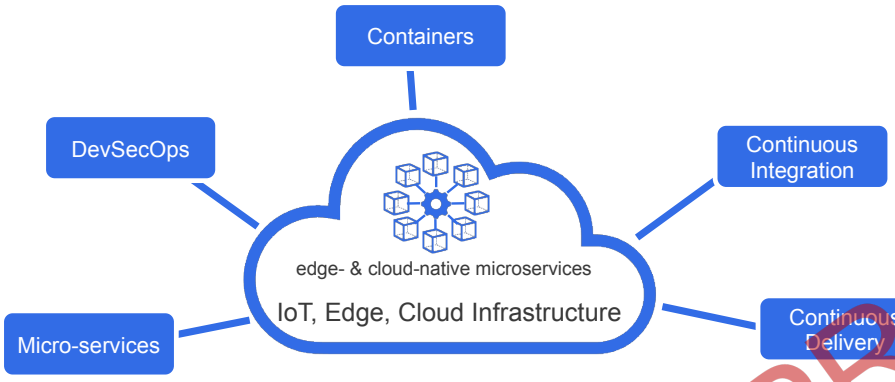


Figure 1: Cloud Native software development approach adopted by NEMO.

The aforementioned design approach is further complemented with an additional set of key recommendations, as guidelines, which are proposed to the NEMO consortium technical teams aiming to greatly ease the integration process, rendering it a lot faster and at the same time more effective.

Large systems such as the envisaged NEMO platform are composed of a set of components that are developed by different geographically dispersed technical teams of the NEMO consortium partners. These components or modules are core elements of the NEMO platform and provide target functionalities. Once developed, they should be able to interoperate with each other and deliver the required functionality that is dictated by the NEMO architecture. Therefore, modular software development practices should be applied by the NEMO developers. In practice, this means that each component or module should:

- Be implemented as a self-contained building block independently from each other.
- Expose their functionality through a set of well-defined and well-documented interfaces.
- Utilize well-defined data models as means of structuring the communicated information (ideally in a standardized format) that is exchanged from one component to another.

Following this approach, each technical team of the project would only need to be aware of the external interfaces exposed by each developed module, thus abstracting arbitrary complexity that otherwise, inevitably would be introduced.

The ultimate goal of modular software design is to define isolated and simple abstractions where the interface should be much simpler than the implementation. Simultaneously, it aims to guarantee that if a change affects only a module's implementation but not its interface, then required fix or update of a module source will not affect any other modules of the system.

2.1.1 Microservices and workflows

Microservices' architecture or Modular Design refers to a system's architecture in which a complex application is composed of a set of independently deployable and loosely coupled modules plugged together. Each module or "microservice" is understood as a software piece of singular functionality, able to be executed independently, but is, however, required in order to deliver the whole application functionality, together with the rest modules, organized around business capabilities. Three groups of elements are the basic components to realize the modular design:

- Modules or microservices

- Well-defined interfaces
- A set of protocols for interconnecting those microservices.

Although this design urges for independent modules, the delivery of holistic application functionality assumes dependency among the execution of the cooperating modules or scheduled/ordered execution of such microservices. This is addressed through workflows, which define both the steps to be executed, but also the workflow's state. Workflow engines have been introduced to address the automation of execution of workflows, including Argo Workflows [2], Apache Airflow [3], Zeebe [4], etc.

The microservices' architecture implies significant benefits arising from functionality -and thus application' teams'- decoupling. As noted in this Redhat article[3], some of the prominent benefits include:

- Shorter time to market, as a result of shorter development cycles and easier adoption of agile development and deployment approach.
- Increased scalability, since service decoupling allows for distributed deployment across servers and clusters, supporting greater level of scalability.
- Increased resilience, as microservices are independent pieces of functionality, so failure of a single microservice does not necessarily lead to failure of the whole application.
- Easier deployment, as smaller service pieces are easier to deploy, as they normally have less computing/networking/resource requirements compared to a monolithic application, at the expense, of course, of the need for service orchestration.

On the other hand, splitting application into smaller pieces requires increased coordination and management, in order to address complexity and efficiency. Specifically, challenges arise for initial configuration with regards to scaling in order to ensure dependencies and software versions' compatibility are respected, but also for tracing through logging and monitoring which can be both complex in a distributed setup. Moreover, service discovery and connectivity require special attention in distributed environments.

NEMO adopts the microservices' architectural style for both core NEMO components and applications consuming the meta-OS services. For the NEMO meta-OS components, splitting the functionality into components will not only accelerate the development, integration and validate time, but will also yield a more efficient, scalable and resilient framework, making better use of available resources. In addition, the NEMO applications adopting the microservices' architecture will be able to achieve increased performance and make the most out of the available resources in the meta-OS.

2.1.2 Communication style

An important aspect that affects microservices' based application delivery refers to the communication style adopted for the microservices. Figure 2 presents three communication architectural styles for microservices, namely [5]:

- Asynchronous communication by commands and events, often referred to as "event-driven architecture" and implemented through a message or event bus, e.g. Kafka, RabbitMQ (AMQP), etc.
- Synchronous communication by request/response, mainly referring to REST (REpresentational State Transfer) APIs
- Both synchronous and asynchronous communication by request/response via the Remote Procedure Call (RPC) model, mainly referring to gRPC [6].

[3] https://www.redhat.com/en/topics/microservices/what-are-microservices

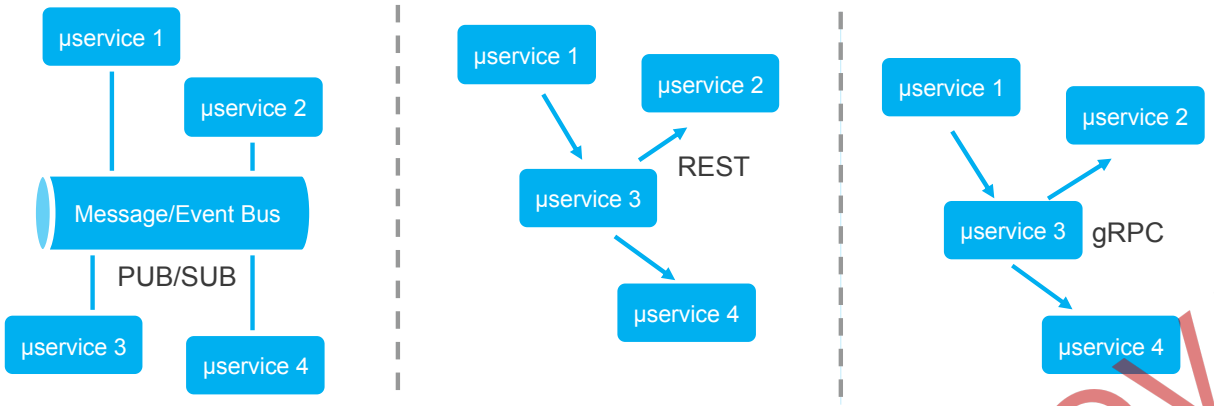


Figure 2: Communication architectural styles for microservices

Asynchronous service-to-service communication, also referred to as publish/subscribe (pub/sub) messaging, enables any message published to a topic to be immediately received by subscribers of that topic. This communication style offers the following benefits.

- The pub/sub messaging pattern is scalable and flexible, in the sense that adding and removing subscribers is a matter of configuration and no complex programming is required.
- Pub/sub is asynchronous, meaning that it can facilitate communication that is prominent to time-outs, thus reluctant on introducing delays and potentially stalling the system communications.
- Pub/sub messaging can be used to enable event-driven architectures, or to decouple applications in order to increase performance, reliability and scalability [7], so it is used in serverless and microservices architectures.

The adoption of the RESTful communication implies also significant benefits for NEMO development, integration and testing, including:

- Easy implementation and maintenance: Most developers are familiar with REST, while it allows for development of polyglot APIs, enabling them to code in their preferred language, while ensuring smooth communication among their modules.
- Increased scalability: REST is stateless on the server side, which implies reduced requirements in memory and storage, as resources are committed only during request processing and are released thereafter. This is important in the meta-OS environment, in which horizontal scaling can be applied, allowing the server side to be replicated on a set of heterogeneous nodes. Keeping and communicating states in such an environment would be complex to manage and would possibly lead to high response times.
- Flexibility in response format: REST supports both JavaScript Object Notation (JSON) and Extensible Markup Language (XML) or other through the *Accept* header. Apart from the apparent benefits in application development, it allows for selection of smaller in size formats, which could be also parsed faster (e.g. JSON).

Finally, gRPC is a modern open-source high performance Remote Procedure Call (RPC) framework, developed by Google, that can run in any environment. It uses HTTP 2.0 as its underlying transport protocol, but HTTP is not exposed to the API designer. In gRPC, a client application can directly call methods on a server application on a different machine as if it was a local object, making it easier to create distributed applications and services [8]. gRPC is mainly used in microservices' architecture to address efficient communication among polyglot microservices, in mobile apps' development to

connect mobile devices, browser clients to backend services, as well as for generating efficient client libraries. Some of the main advantages of gRPC, also highlighted by Linkedin engineers [9] include:

- Increased communication performance: Due to its binary serialization and compressed data formats through protocol buffers, gRPC results in faster data transfer. Moreover, it uses HTTP/2 to enable multiplexing, header compression and server push, while its inherent design enabling fully asynchronous non-blocking bindings and advanced threading models contributes to improved efficiency, as well.
- Easy implementation and advanced capabilities: gRPC provides code generation capabilities that use Protocol Buffers to define the service and message interfaces, allowing developers to easily maintain and update the codebase across different languages. Also, it offers advanced capabilities, including bidirectional streaming, flow control, and deadlines.
- Polyglot support: gRPC officially supports a number of programming languages[4], which a significant factor for complex information systems, possibly composed of polyglot microservices.

However, gRPC interfaces provide limited options for proxy-based access control, which can be easily applied in URL-based endpoints.

Moving to a more holistic view, microservices have been proved perfect fit for independent functionality delivery. However, their interaction, management and scheduling within workflows has to be addressed for integrated application delivery. This is where communication based on workflow engines came into play. In addition, the parallel evolution of multi-cluster orchestration solutions and automation in such environments urges for considering the management of such workflows within distributed, possibly multi-cluster environments. This technological need has led to relevant features in DevOps tools, like Argo CD [10] and Flux [11]. Specifically, the Argo CD *ApplicationSet* controller [12] adds Application automation and seeks to improve multi-cluster support and cluster multitenant support within Argo CD. It is thus possible to use a single Kubernetes manifest to target multiple Kubernetes clusters or deploy multiple applications from one or multiple Git repositories with Argo CD, while within multitenant clusters it is possible to deploy applications using Argo CD, without needing to involve privileged cluster administrators. Moreover, Flux supports multi-tenancy in its v2 [13], together with its ability to manage deployments to multiple remote Kubernetes clusters from a central management cluster and the support for progressive delivery. These are also critical features to allow multi-tenant execution of workflows in multi-cluster environments.

In a distributed, heterogeneous and multi-cluster environment, like the NEMO meta-OS, it is highly recommended to adopt each of the architectural communication styles that are mostly applicable per case. In particular, in NEMO an amalgamation of these microservices' architectures is sought, with REST for enacting synchronous communication among its components through common HTTP APIs and pub/sub messaging for asynchronous communications. A significant option when trying to define interfaces and data models is to utilize standardized tools and practices. With respect to interfaces standardized and consistent interfaces (e.g. RESTful APIs following the OpenAPI specification [14]) shall be adopted where applicable. Moreover, it is highly recommended to adopt automated workflow execution for increased application and resource efficiency across the meta-OS resources.

[4] https://grpc.io/docs/

2.2 Code and API documentation

Proper documentation, in general, facilitates greatly the development and integration activities. It can be approached as a vertical necessity for any project, especially in a complex one such as NEMO, and should be part of every development and integration step. Starting with source code documentation, which greatly helps to comprehend a piece of software that is written even by the same developer at some earlier time and especially if it has been written by someone else. In addition, source code documentation, significantly eases a future update or correction activity within module's code, where source code "comments" will help the developer to comprehend the code's functionality and avoid misconception mistakes that could easily occur. In general, it is a very good practice to be adopted by the project's developers.

In addition, API documentation, especially in electronic format, greatly benefits both internal integration activities (within the project), as well as integration or extension of the concerned software modules in third-party systems and applications. NEMO adopts OpenAPI specification for synchronous APIs and AsyncAPI [15] specification for event-driven APIs. OpenAPI specification creates a RESTful interface for easily developing and consuming an API by effectively mapping all the resources and operations associated with it. It was initially created by Swagger [16] and has been donated to the Linux Foundation, being now at version 3.1 and the world standard for RESTful APIs. AsyncAPI was influenced by OpenAPI in the way it structures its definitions, using YAML (or JSON) and re-using the same structures found in OpenAPI, where possible. It makes use of JSON Schema for some of its model definitions and includes support for others (e.g., Avro). AsyncAPI is also part of the Linux Foundation and gets distinguished as the industry standard for defining asynchronous APIs. OpenAPI is fully supported until v3.0 in the open-source version of the *Swagger Editor*, while the beta version of the open-source *Swagger Editor Next* provides additionally full support of AsynAPI and partial support of OpenAPI v3.1 [17].

2.3 Containerization

Containerization is a major technology that offers an abstraction layer over the application layer which packages code and library dependencies. It strongly facilitates the DevSecOps operations that are realized via the NEMO CI/CD pipeline and is aligned with the Cloud Native paradigm that is adopted by NEMO. Containerization is a lightweight alternative to a virtual machine that involves encapsulating an application in a container with its own operating system. The most popular containerization technology application is Docker [18]. Docker is a set of Platform as a Service (PaaS) products that use OS-level virtualization to deliver software in packages. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Moreover, Kubernetes (K8s) which is an open-source system for orchestrating containers (described in Section 2.4.2 that is also adopted by NEMO, provides for automated deployment, scaling, and management of containerized applications and is appropriate for working with Docker containers. In alignment with the cloud-native approach and the modular architecture principles outlined above, NEMO adopts Docker as a containerization framework and, thus, strongly encourages the technical developers of the project to wrap their implemented technical outcomes in docker containers.

2.4 Automation

Automation lies at the core of the meta-OS under different scopes. In the following, specific practices are suggested under different angles, as integral parts of the meta-OS to achieve effective orchestration of workloads across remote devices of varying capacities.

2.4.1 ZeroTouch provisioning

The highly diverse meta-OS environment embraces edge deployments which should allow highly varying scenarios in data processing and management at the edge in a similar way that this happens in the cloud. The meta-OS should be capable of adapting workload orchestration to scenarios of diverse computing needs and capabilities, which are served by a multitude of diverse devices.

Such network and infrastructure complexity implies increased requirements in settings and configurations, which will allow the management and integration of such devices into the meta-OS ecosystem. Complexity does arise not only from hardware or software specificities, but also from potential custom policies or requirements arising from diverse administrative domains and sites.

Thus, homogenized and centralized orchestration of workloads across such highly varying environment and infrastructure can be achieved by propagating network configurations in reasonable time for application delivery and also at low complexity and high replicability.

NEMO encourages automation in configuration and management of diverse meta-OS nodes, including edge and cloud devices, in line with *ZeroTouch Provisioning (ZTP)*[5]. ZTP has been already adopted for provisioning network devices such as firewalls, wireless access points, routers, network switches, etc., e.g., by Cisco[6],Juniper Networks[7], Through ZTP, NEMO will be able to automatically provision meta-OS clusters with reduced labor costs and increased deployment efficiency.

ZTP eliminates the need for onsite, manual configuration and deployment, which reduces labor costs and improves deployment efficiency.

2.4.2 ZeroOps deployment at the network edge

ZeroOps has significantly improved the productivity of developing teams, alleviating a huge burden of manual configurations and installations across devices or virtual nodes. ZeroOps harnesses the merits of cloud-native design in practice. NEMO welcomes the adoption of ZeroOps practices for workload configuration and deployment at the edge devices, as well. In order to achieve this, NEMO suggests adopting a DevOps approach, which will support:

- CI/CD with static code analysis and vulnerability checks, including automated release process.
- Unified operational runtime for containerized workloads with built-in runtime protection and application observability.
- Infrastructure and operational runtime monitoring with automated standard operating procedures and incident remediation respecting user-defined policies.

2.4.3 AI advancing automation

Artificial Intelligence (AI) has matured enough so that it is present in enhancing and automating multiple facets of business life. AI can thus be well applied to advance automation in DevOps. As stated by GitLab [19]: "*AI in DevOps involves the use of machine learning (ML) and other artificial intelligence technologies to automate and optimize the software development and delivery process. This includes everything from automating testing and deployment processes to improving resource management and enhancing security*". Relevant to this concept is AIOps, with Gartner [16] defining that "*AIOps combines big data and machine learning to automate IT operations processes, including event correlation, anomaly detection and causality determination"*.

[5] https://www.redhat.com/rhdc/managed-files/ve-reinventing-telecommunications-open-innovation-ebook-230150-202302-en.pdf
[6] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/prog/configuration/1612/b_1612_programmability_cg/zero_touch_provisioning.html
[7] https://www.juniper.net/documentation/us/en/software/paragon-automation22.1/paragon-automation-user-guide/topics/concept/ems-ztp-overview.html

As the main objective in DevOps has been to leverage and facilitate collaboration among development teams, AI can introduce intelligent enhancements in this collaboration. Indicatively, AI may advance the following activities:

- AI for continuous integration and continuous delivery or deployment (CI/CD): AI can be leveraged to identify and merge/build/integrate individual pieces of code which have successfully passed appropriate tests, and deploy them to production environment
- AI for test automation: AI can be used to automatically run tests on developed code
- AI for assisting code development: AI can provide code suggestions at development time
- AI for enhancing system observability and automation: AI can be used to automate contextualization of operational data, allowing for predicting issues, suggesting mitigations and automating alerts during the application life cycle.

The use of AI in the DevOps lifecycle implies significant benefits in terms of efficiency, improved quality, scalability, as well as reduction of costs. With AI, resources and capacities can be more effectively used. However, in order to maximize the benefits, the introduction of AI requires smooth integration into the Operational Technology (OT) processes. This is crucial to prevent fragmentation among the development and integration teams or even among the processes within the same team. Moreover, in order for AI to be effective in automating and truly improving the DevOps process, it should rely on reliable observability data and ML development processes.

NEMO embraces leveraging AI in the DevOps lifecycle, respecting and addressing potential challenges in its incorporation into the development, integration and deployment of the NEMO framework.

2.5 Open source

Open-source tools are essential elements to achieve increased levels of openness and adoption and can leverage the wider developer community, enabling diversification of suppliers. Greater involvement and engagement of third-party developers may trigger faster and wider functionality extension, more transparent verification process and increased maintenance and sustainability potential.

NEMO supports the adoption and extension of open-source frameworks for building the NEMO meta-OS components and plugins.

Moreover, NEMO aims to actively support and contribute to the open-source community, by making the NEMO source code openly available since the early stages of development. Our goal is to achieve a positive impact to industrial innovation, related to multi-orchestration, edge/cloud computing and advanced networking, including 5G/6G enhancements, as well as relevant technological innovations.

2.6 Security first

The European Union legislated the first network and information security (NIS) directive for EU Member States in 2016, updated by the NIS 2 Directive [20] that came into force in 2023. Both NIS and its update NIS 2 directives aim at building cybersecurity capabilities across the Union, mitigating threats to network and information systems, contributing to the Union's security and to the effective functioning of its economy and society.

Some key aspects highlighted in NIS 2, which are relevant for the development of a secure meta-OS environment include:

- “Member States should encourage the use of any innovative technology, including artificial intelligence, the use of which could improve the detection and prevention of cyberattacks, enabling resources to be diverted towards cyberattacks more effectively”.
- “Rather than responding reactively, active cyber protection is the prevention, detection, monitoring, analysis and mitigation of network security breaches in an active manner, combined with the use of capabilities deployed within and outside the victim network.”

- "Since the exploitation of vulnerabilities in network and information systems may cause significant disruption and harm, swiftly identifying and remedying such vulnerabilities is an important factor in reducing risk. Entities that develop or administer network and information systems should therefore establish appropriate procedures to handle vulnerabilities when they are discovered."
- "Essential and important entities should ensure the security of the network and information systems which they use in their activities. Those systems are primarily private network and information systems managed by the essential and important entities' internal IT staff or the security of which has been outsourced."
- "Essential and important entities should adopt a wide range of basic cyber hygiene practices, such as zero-trust principles, software updates, device configuration, network segmentation, identity and access management or user awareness, organise training for their staff and raise awareness concerning cyber threats, phishing or social engineering techniques."

Moreover, the US Cybersecurity & Infrastructure Security Agency (CISA) released the Zero Trust Maturity Model as a roadmap for transitioning to a Zero Trust architecture in 2021 and updated it in 2023 [21]. According to it, the US National Institute of Standards and Technology (NIST) defines Zero trust as providing "*a collection of concepts and ideas designed to minimize uncertainty in enforcing accurate, least privilege per-request access decisions in information systems and services in the face of a network viewed as compromised*".

In common terms, Zero Trust assumes all actors, systems, and services operating in and between networks cannot be trusted. The ZeroTrust model aims at guaranteeing secure access to resources only when and to whom necessary, based on three fundamental concepts [22]:

- **Secure network:** Always assume that the network is hostile and compromised. Internal and external data and information on the network is constantly exposed to security threats.
- **Secure resources:** Any source of information that exists on the network should be viewed with suspicion, regardless of the location.
- **Authentication:** Users, devices, and traffic from internal or external networks should not be trusted by default. Zero trust should be based on access control using the right authentication and authorization.

Securing resources and data becomes increasingly complex and challenging across multi-variate multi-domain virtualized environments, composed of multi-clouds and multi-clusters.

Kubernetes clusters can be accessed through the Kubernetes API using kubectl, client libraries, or by making REST requests by both human users and Kubernetes service accounts [23]. Kubernetes provides several security hooks to implement ZeroTrust for requests to the API, as depicted in Figure 3. These include Authentication, Authorization, Admission Control, Logging, and Auditing.

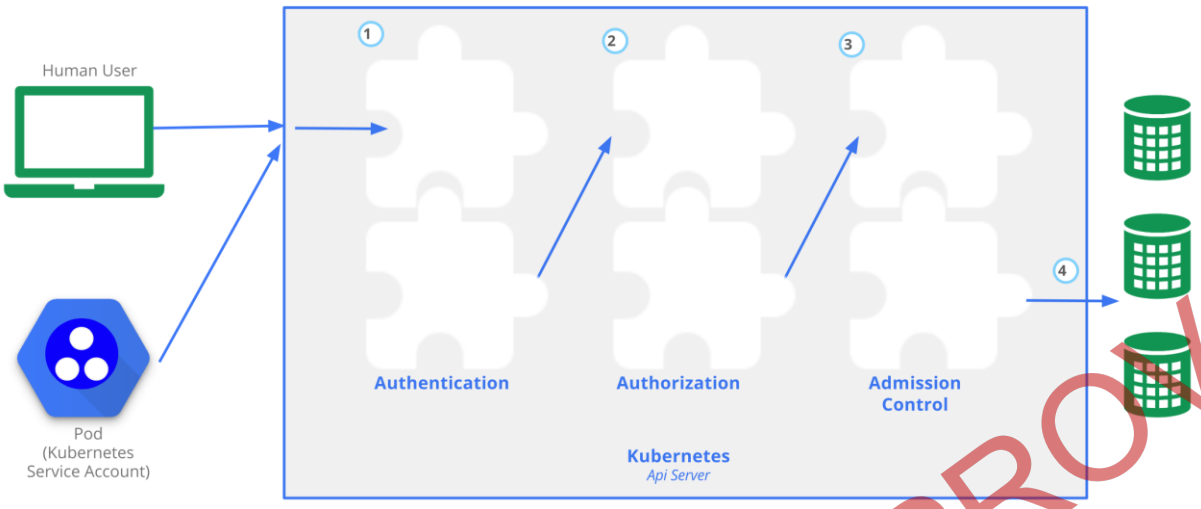


Figure 3: Kubernetes security hooks for implementing Zero Trust [23]

In the context of adopting a cybersecurity culture, the following actions can be taken to realize Zero Trust within Kubernetes, besides access control applied to the K8s API [24]:

- **Delivering identity-based service-to-service access and communication:** Service identity should be enforced and used for service-to-service authentication and authorization to ensure that access to necessary resources is granted only to proper services.
- **Enforcing encryption in secret and certificate management and hardening Kubernetes encryption:** Besides adopting "Good practices for Kubernetes Secrets" [25], secrets should be encrypted either at rest or in transit and be subject to fine-grained Role-Based Access Control (RBAC) policies to limit access to secrets based on roles and responsibilities. Moreover, secrets stored in the external secret management system should be regularly rotating. Access credentials should be time-bound, requiring the user or application to refresh their credentials at defined intervals.
- **Enabling observability with audits and logging:** Audit logs kept per user/role/service identity basis would allow for greater insights and accountability over event data.

Widely adopted approaches to realize the aforementioned actions involve service mesh solutions. A service mesh with a secrets broker can compensate for the inherent security weaknesses of secrets management in Kubernetes, addressing the secrets' encryption and centralized management, as well as secrets' time-based rotation. Moreover, a service mesh by design deploys sidecar along with every application deployed, allowing application-aware traffic management, observability, and robust security capabilities. When integrated with open-source monitoring tools (e.g. Prometheus, Grafana, etc.), increased capabilities for analyzing service-networking patterns and enforcing security are provided for the security manager. Notable service mesh solutions, which could undertake adding security capabilities in the network, include Istio [26], Consul [27], Kong Mesh [28], etc.

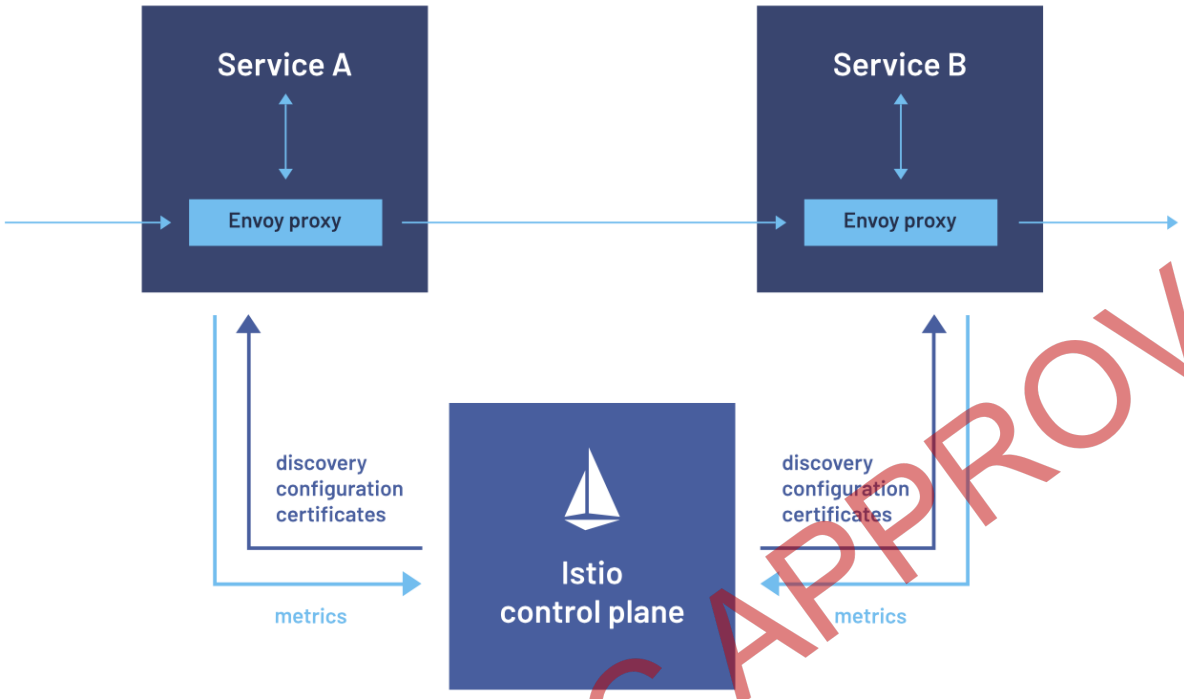


Figure 4: Istio service mesh operation, adding a sidecar proxy to each deployed application [26]

Moreover, API Gateway solutions can provide control and data planes which can enforce traffic and user policies, before granting access to underlying APIs, acting as reverse proxy to those for incoming requests. An API Gateway can be used in the context of implementing ZeroTrust for applying L7 user policies (e.g., AuthN/AuthZ use cases, rate-limiting, developer on-boarding, monetization or client application governance), for applying L7 traffic policies (e.g., enforcing networking policies to connect, secure, encrypt, protect and observe the network traffic between the client and the API gateway, as well as between the API gateway and the APIs), as well as for supporting the full lifecycle API management by connecting third-party LCM solutions to the API gateway to execute policy enforcement [29]. Popular API Gateway solutions include Kong API [30], Envoy Gateway [31], etc.

Similarly, Ingress Controllers can enforce ZeroTrust through policy decision/enforcement for incoming traffic in Kubernetes environments, acting quite similar to API Gateways. Indicatively, NGINX solution demonstrates a combined solution to Zero Trust through the NGINX Ingress Controller and the NGINX Service Mesh [32]. The two components together address ZeroTrust requirements for both external access to Kubernetes clusters and inter-cluster service communication.

The challenge in the meta-OS multi-cluster environment is to enforce Zero Trust principles, as implemented for Kubernetes clusters in the multi-cluster environment, in a coherent and centralized manner. NEMO already considers an API Gateway and Service Mesh solution for enforcing Zero Trust principles in the NEMO meta-OS.

3 Integration and Lab V&V approach and tools

3.1 DevSecOps in NEMO

NEMO adopts DevSecOps practices to support the platform's integration and testing activities and the management of the foreseen NEMO platform releases. DevSecOps, in plain words, involves security enhancements throughout the DevOps cycle. These may refer to a set of practices that complement Agile Software development, and which integrates security initiatives at every stage of the software development lifecycle to deliver secure and robust applications. DevSecOps, short for Development, Security, and Operations, as illustrated in Figure 5, automates the integration of security at every phase of the software development lifecycle, from initial design through integration, testing, deployment, and software delivery.

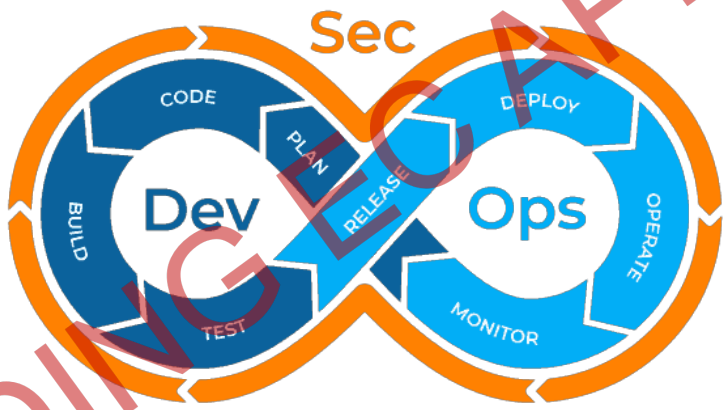


Figure 5: NEMO DevSecOps

3.1.1 Security enhancements

As already mentioned, DevSecOps introduces security enhancements into the CI/CD pipeline. The security tests can be distinguished in Static Application Security Tests (SAST) and Dynamic Application Security Tests (DAST). Each category is further elaborated in the following sections.

Static Application Security Testing (SAST), also known as "white box testing", allows developers to find security vulnerabilities in the application source code earlier in the software development life cycle. SAST is utilized to check the code without executing it. It also ensures conformance to coding guidelines and standards without actually executing the underlying code. Incorporating a static analyzer into the CI/CD loop helps forestall programming bugs from the early stages of the development before getting to a higher level.

Dynamic Application Security Testing (DAST), also known as "black box testing", investigates for security vulnerabilities and weaknesses in a running application. It is performed later in the development lifecycle, as it requires a built and tested application. The tester has no knowledge of the application's source code or the technologies or frameworks the application is built on. DAST, in a nutshell, tests the security of developed software by feeding it with malicious data trying to detect security vulnerabilities and to determine security vulnerabilities that are linked to the operational deployment of an application.

Runtime Application Security Protection (RASP), called runtime application security protection, or RASP, shields running applications from a variety of online risks and attacks. RASP functions at the runtime level, monitoring and protecting the application while it is being executed, in contrast to conventional security measures that concentrate on the network or the application code itself.

3.2 NEMO CI/CD approach

NEMO will be heavily based on open-source projects and tools from Cloud Native Computing Foundation (CNCF), and the results of previous H2020 projects. The latest version of the software will be automatically integrated upon successful source code updates and compatibility tests and a new version will be deployed on the Integration Infrastructure hosted in the OneLab. Before each major or minor release cycle, the Qualification Infrastructure will be used for extensive functional and penetration testing and bug fixing, without interfering with the development of new releases, happening in the Integration Infrastructure or with the normal use of the Living Labs, which is equivalent to a Production Infrastructure. In this way, at release time, the pilots will be updated with zero downtime.

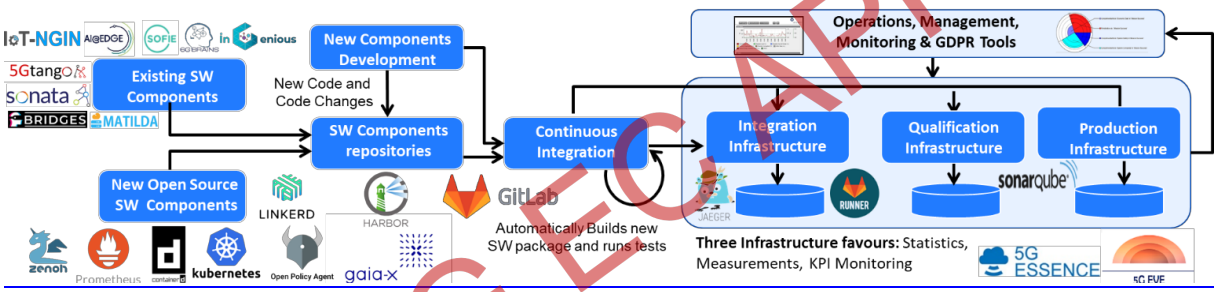


Figure 6: NEMO CI/CD/CP approach

3.2.1 CI/CD environment and tools

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

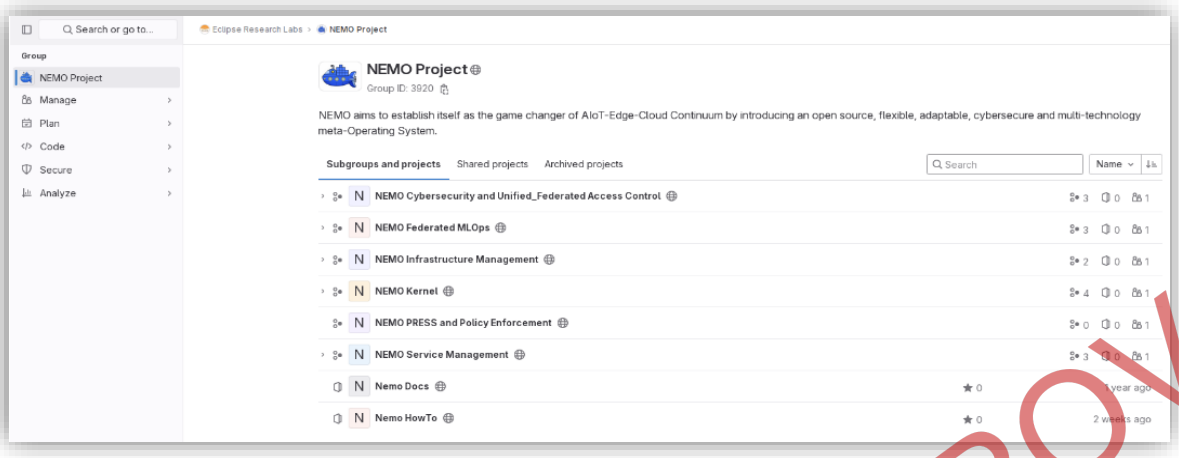


Figure 7: NEMO code repository in Eclipse Research labs

Within each subgroup, the development activities are organized based on the implemented outcomes of the relevant tasks.

3.2.2 NEMO Automated Deployment and Configuration

Automated deployment and configuration are essential features in the context of NEMO and will be achieved via the usage of Flux CD [11]. Flux is a Cloud Native Computing Foundation (CNCF) [33] graduated project and can be defined as an open-source continuous delivery and GitOps tool designed to simplify and automate the deployment and lifecycle management of applications and infrastructure on Kubernetes.

With Flux CD, the desired state of the applications and configurations is stored as code in a Git repository. Then, Flux CD continuously monitors repositories for changes and applies the essential updates to the Kubernetes cluster automatically. In Figure 8, an overview of the Flux operations is presented. The most important Flux components are presented in the following:

- **Source Controller:** Its main role is the provision of a common interface for retrieval of artifacts, by defining a set of Kubernetes objects that cluster admins and operators can interact with to offload operations related to Git and Helm to a dedicated controller.
- **Kustomize Controller:** A Kubernetes operator that is specialized in executing continuous delivery pipelines for infrastructure and workloads defined by Kubernetes manifests, assembled with Kustomize [34].
- **Helm Controller:** A Kubernetes operator that allows the declarative management of Helm releases via Kubernetes manifests.

As far as the operational principles and workflow of Flux is concerned, a simplified explanation is as follows. To begin with, Flux CD runs as an agent within the Kubernetes cluster of interest, continuously monitoring both the cluster resources and the predefined Git repository, which includes configurations and resources of the cluster's desired states. When changes are recorded in the Git repository (e.g. new commits), Flux detects these changes and automatically synchronizes the cluster to match the new desired state, by deploying (or updating) the existing resources and custom configurations.

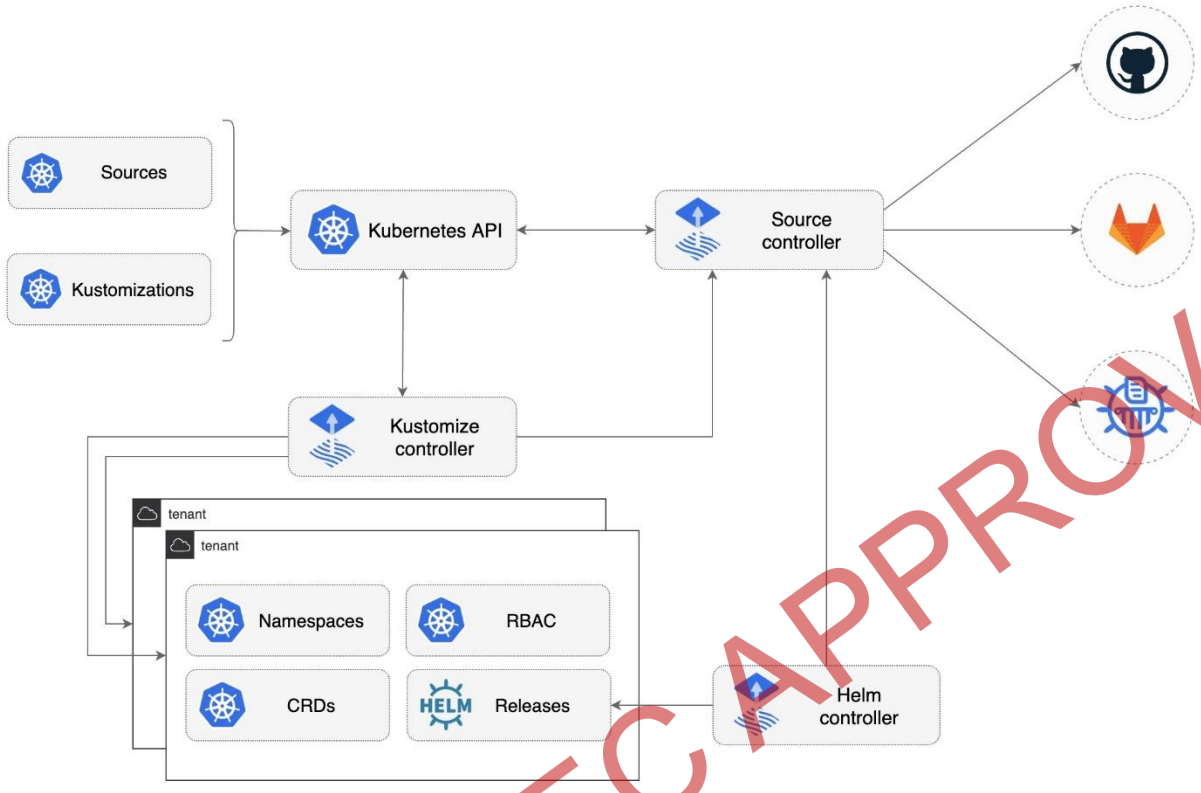


Figure 8: Flux CD and GitOps Toolkit

3.3 Cloud/Edge/IoT Integration and Validation Infrastructure

3.3.1 OneLab

The OneLab facility is a state-of-the-art test platform for exploring the design of digital infrastructures. It provides control and remote access over a large and diverse set of virtualized and programmable resources from IoT to the Cloud. OneLab federates multiple facilities among which the NITOS (Network Implementation Testbed using Open Source platforms) and the FIT (Future Internet Testing facilities) (Figure 9) test platforms and offers the possibility to run large-scale experiments combining multiple heterogeneous resources through one single portal.

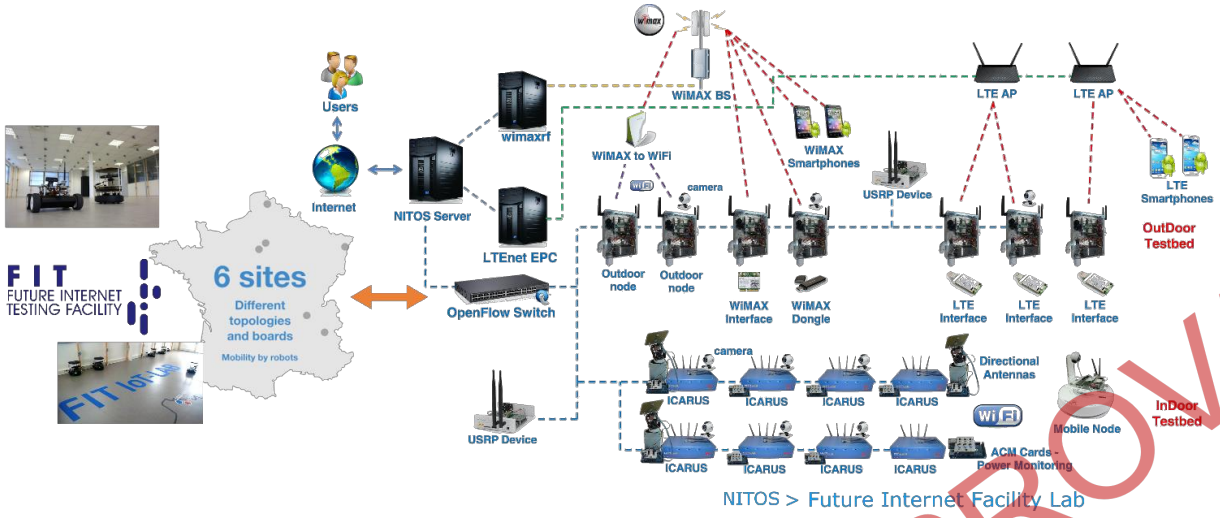


Figure 9: SU OneLab (FIT NITOS & FIT IoT-Lab) as NEMO Integration, Test and Validation Infrastructure

OneLab provides a large set of IoT resources including mobile and wireless IoT devices equipped with various sensors such as ambient light, temperature, atmospheric pressure and temperature sensor, tri-axis accelerometer, tri-axis magnetometer, tri-axis gyrometer and also cloud and bare-metal resources allowing experimentation with cloud-based technologies with over a hundred of computing cores available to experimenters.

OneLab offers, through the federated NITOS Lab facility, a SDR (Software Defined Radio) testbed consisting of wireless nodes attached with USRP devices and a SDN testbed equipped with multiple OpenFlow enabled switches to run experiments with network switching and routing protocols.

3.3.2 OneLab Clusters for NEMO

Two different Kubernetes Clusters have been configured to serve different purposes. A main cluster as the production cluster and an auxiliary cluster for components that require a Graphics Processing Unit (GPU) workload.

These clusters are composed of a set of nodes: a master node to handle the scheduling and scaling of applications, and also the management of the cluster, and worker nodes to perform tasks assigned by the master node, such as deploying the container and hosting the applications. In the production setting, multiple worker nodes are used to offer redundancy and to ensure the service availability.

The main and auxiliary cluster setup is specified in Table 1 and Table 2.

Table 1: Main Cluster

Node ID	Node Type	Specifications	Storage
A	Master	8 CPU Cores 16Go RAM	
B	Worker and Storage	16 CPU Cores 32Go RAM	120Go (Ephemeral) 150Go (Ceph OSD)
C	Worker and Storage	16 CPU Cores 32Go RAM	120Go (Ephemeral) 150Go (Ceph OSD)
D	Worker and Storage	16 CPU Cores 32Go RAM	120Go (Ephemeral) 150Go (Ceph OSD)

Table 2: Auxiliary Cluster

Node ID	Node Type	Specifications	Storage
A	Master	4 CPU Cores 8Go RAM	
B	Worker and Storage	8 CPU Cores 16Go RAM	150Go (Ephemeral) 200Go (Ceph OSD)
C	Worker and Storage	8 CPU Cores 16Go RAM	250Go (Ephemeral) 200Go (Ceph OSD)
D	Worker and Storage	8 CPU Cores 16Go RAM	250Go (Ephemeral) 200Go (Ceph OSD)
E	Worker and Storage	4 CPU Cores 8Go RAM GPU Nvidia Tesla T4	80Go (Ephemeral)

3.4 NEMO V&V approach

To ensure that all the components and applications of NEMO are properly integrated into the platform, the NEMO V&V approach has been conceived in which appropriate tests and checks are foreseen before the deployment of each component. The goal of V&V is two-fold (i) to support developers in evaluating the performance of their services and (ii) to address the concerns that service operators have in hosting third party services upon their infrastructure. For these reasons, NEMO V&V will provide a well-structured framework, as part of the DevOps approach, that facilitates several tests per each new service from the development, integration, and deployment phases ensuring, on the one hand, that the new service addresses the requirements and Key Performance Indicators (KPIs) and, on the other hand, that it is compatible with the innovative features that the NEMO platform offers (i.e. resource scaling, high availability, full-stack automated operations, etc.). The main guidelines of the V&V approach have been presented in D1.2. Based on these, the V&V procedures are distributed among NEMO components (i.e. API, SDK, CI/CD pipelines etc.) to guarantee that the appropriate tests/checks are executed in time and potential errors are identified before they cause any problems. For example, on the one hand, tests that are related to the development of the components (i.e. unit, integration, system tests, etc.) are part of the NEMO CI/CD/CP approach and on the other hand, compatibility tests of the uploaded user applications are going to be executed in the NEMO API, through the NEMO consumer (application developer) may to register his/her application as a NEMO workload (more details on the API operation can be found in Section 5.4). Considering the heterogeneity of modern network services, each service requires different testing approaches and tools. Therefore, the V&V approach should provide common tests applicable to all services (i.e., NEMO platform compatibility tests), but should also support the integration of services' specific tests that verify specific aspects of each service. The V&V tests are integrated in the NEMO CI/CD pipeline, as indicated in Figure 10. As derived from the figure, automated tests of various scopes, as well as security tests, are foreseen during continuous integration (CI) and continuous deployment (CD) phases. The tests in the CD part could be triggered, both for developments taking place within the CI process (on the NEMO code repository), as well as for components committed as *deployments* in the Deployment code repo and provided to the CD component.

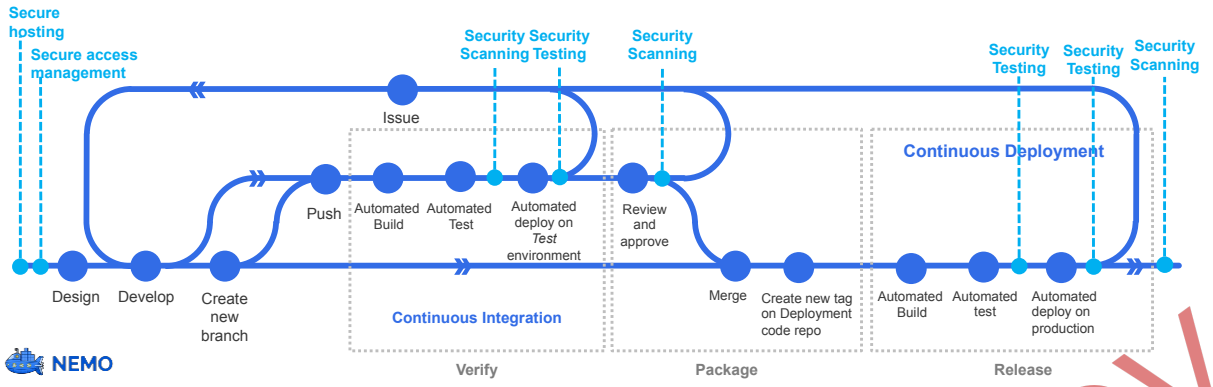


Figure 10: Validation and verification in the NEMO CI/CD pipeline

3.4.1 Testing Categories

According to the guidelines defined in D1.2, NEMO will implement the different types of tests that can be used for the evaluation of the code during the Software Development Life Cycle (SDLC) of the platform as well as the verification of the user applications. These tests are the following.

Requirement analysis: The definition of accurate and reliable testing procedures requires the identification of proper functional and non-functional requirements starting from the design phase. NEMO facilitates this need by providing specific tools and methodologies to collect, document, and prioritize all technical requirements in a clear and measurable way. For the components of the NEMO platform, Kubernetes Admission Controllers may be used to add semantic tags regarding the resource limits per service. This will ensure that NEMO services will work properly and prevent potential misbehaviors. Regarding user services, the same approach will be followed based on the same framework, exploiting information from the NEMO CMDT, regarding the definition of intents.

Functional/Integration testing: These tests aim to verify that a software application behaves according to its specifications and test the interactions between different components or systems of NEMO to ensure they work together as intended. According to different technologies applied to NEMO components, there are several open-source tools under consideration as implementation reference. For example, regarding web services and UIs, Selenium [35] is well-known solution that provides a flexible and easy way to implement test procedures. Selenium is a powerful and widely adopted open-source framework for automating web browsers, primarily used for functional testing of web applications. Offering compatibility with multiple programming languages, including Java, C#, and Python, Selenium enables testers and developers to write scripts for automated browser interactions. Its key components include Selenium WebDriver for browser automation, Selenium Grid for parallel execution across multiple machines, and Selenium IDE for record and playback functionality. Selenium supports a variety of browsers, allowing cross-browser testing, and its flexibility makes it suitable for a range of applications, from simple websites to complex web-based production systems. Due to its extensive community support, active development, and compatibility with various testing frameworks, Selenium has become a fundamental tool in the software testing ecosystem. For backend services, tools like Robot Framework [36] are quite suitable. Robot Framework is an open-source and highly extensible test automation framework designed for both acceptance testing and robotic process automation (RPA). Developed using Python, Robot Framework utilizes a keyword-driven approach that emphasizes readability and collaboration between technical and non-technical team members. It employs a simple, tabular syntax in test case files, making it accessible to individuals with varying levels of programming expertise. The framework supports a wide range of test libraries and can be extended with custom libraries using Python or Java. Robot Framework is versatile, allowing automation of web, mobile, desktop, and API testing. Its modular architecture and support for parallel test execution contribute to its scalability, and integration capabilities with CI tools enhance its suitability for agile development

environments. The framework's popularity is attributed to its simplicity, maintainability, and robust support for test automation across diverse application types.

This category of tests is going to be executed within the CI/CD/CP process through specific automated pipelines.

Performance testing: It focuses on evaluating the speed, responsiveness, scalability, and stability of a software service under various conditions. The primary goal is to ensure that the application meets performance expectations and can handle the expected load without degrading its performance. Performance testing involves simulating different scenarios and analyzing the system's behavior under these conditions. During the evaluation process, the following aspects are taken under consideration:

- Load Testing evaluates the system's ability to handle a specific load or concurrent user interactions.
- Stress Testing evaluates the system's behavior under extreme conditions, beyond its specified limits.
- Endurance Testing evaluates the system's ability to handle a sustained workload over an extended period.
- Scalability Testing evaluates the system's ability to scale up or down with changes in user load or system resources.
- Benchmark Testing compares the performance of the application against industry standards or competitors.
- Responsiveness Testing evaluates the system's responsiveness and user experience.

As the definition of this type of testing procedure is highly dependent on the characteristics of the system under test (SUT) is difficult to create a solution that addresses all the potential needs. So, NEMO decided to include these tests in the CI/CD/CP approach to facilitate the integration with any open-source framework that fits better to the developers' needs. Some of the most popular open-source tools are:

Apache JMeter [37] is a widely used, open-source performance testing tool designed for evaluating the performance and scalability of web applications. Developed in Java, JMeter provides a user-friendly graphical interface that allows testers to create and execute load tests, performance tests, and stress tests. It supports various protocols, including HTTP, HTTPS, FTP, and more, enabling the simulation of diverse user scenarios. JMeter allows users to define test plans, set up thread groups, and configure samplers to simulate user interactions. With features such as assertions, listeners, and reporting tools, JMeter facilitates comprehensive analysis of application performance, identifying bottlenecks, and providing valuable insights into system behavior under different loads. Its extensibility and active community support contribute to its popularity in the field of performance testing.

Taurus [38] is an open-source test automation framework designed for continuous testing and performance testing. Offering a configuration-driven approach, Taurus simplifies the setup and execution of tests using various testing tools, including JMeter, Gatling, Selenium, and others. It provides a simple YAML-based configuration file, allowing users to define test scenarios, parameters, and desired configurations in a straightforward manner. Taurus aims to enhance testing and integrate seamlessly with CI systems by providing flexibility, scalability, and support for diverse testing needs. With its ability to orchestrate and execute tests across multiple tools, Taurus facilitates efficient and comprehensive testing in continuous integration pipelines. The framework's user-friendly approach makes it accessible to both technical and non-technical users, contributing to its popularity in the realm of automated testing and continuous integration.

Syntax testing: It checks and validates the correctness of the syntax in code, and the compatibility of the users' applications with the NEMO platform. Besides the adherence to the NEMO syntax rules, these tests are going to check if the users' applications include all the required information for the successful deployment of the application in the NEMO continuum. These tests are going to be included in the Intend-based SDK/API component (see section 5.4) and the following open-source tool will be

considered as one of the potential reference implementations. Checkov [39] is an open-source static code analysis tool used for finding misconfigurations in infrastructure as code (IaC) and supports various IaC tools such as Terraform, CloudFormation, and Kubernetes manifests. It scans IaC files to identify security and compliance issues, ensuring that cloud infrastructure configurations adhere to best practices and compliance standards. Checkov leverages a comprehensive set of built-in checks and policies, and it allows users to customize and extend these checks based on their specific requirements. It can be easily integrated into CI/CD pipelines so developers can proactively identify and address security vulnerabilities and misconfigurations in their cloud infrastructure, promoting a secure and compliant cloud environment.

System testing: It evaluates the overall NEMO platform rather than individual components and considers the interactions and dependencies between components to ensure they work cohesively. These tests follow, after the successful execution of all the individual integration tests between the NEMO components and combined with the performance tests ensure the current version of the platform is ready to be released. These tests will simulate specific and complicated end-to-end actions to verify that the NEMO system is functional. Some of these tests will consist of the registration of new VIMs, the upload/update/deployment/delete of user services, migration of running microservices, etc. The implementation of these tests can be based on the open-source tools that have been referred above (i.e. Selenium, Robot Framework, etc.), and will be executed by specific CI/CD/CP pipelines manually triggered before every code release.

Security testing: These tests' procedures in NEMO aim to identify vulnerabilities and weaknesses in software components to ensure that the system is robust and resistant to security threats and attacks. Security testing involves assessing various components of the software, including its infrastructure, code, and user interfaces, to identify potential security risks. The security testing types that are going to be implemented in NEMO platform are aligned to the following approaches and are executed in the context of DevSecOps (see section 3.1).

- Static Application Security Testing (SAST): Analyzes the source code, bytecode, or binary code of an application for security vulnerabilities.
- Dynamic Application Security Testing (DAST): Evaluates the security of a running application by actively testing it for vulnerabilities during runtime.
- Interactive Application Security Testing (IAST): Combines elements of both SAST and DAST, providing real-time analysis of applications during runtime.

3.4.2 Assessment & Labelling

The NEMO V&V framework aims to deliver a base mechanism that can be used for service certification. If the under-test services successfully pass all the predefined tests, then it can be considered a certified service and it can be safely deployed in the NEMO continuum. The outcomes of each test are going to be kept to the CMDT and execution details and logs are going to be stored in the CI/CD/CP framework.

3.5 Integration & V&V Plan

NEMO will follow an agile and incremental approach of iteration cycles, grouped in 3 Phases, as depicted in Figure 11.

- **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.

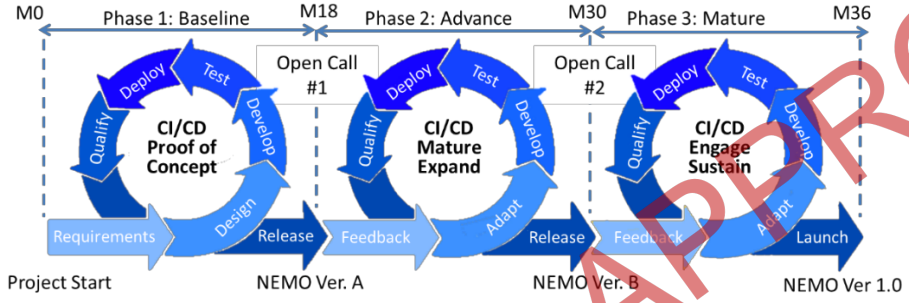


Figure 11: NEMO project phases and main meta-OS version releases

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

4 NEMO Integrated Platform (Ver. 0)

NEMO aims to build the *meta-Operating System* (meta-OS), which will enable multi-cluster and multi-network orchestration of containerized workloads across the IoT, edge and cloud continuum. As a (meta-)OS, NEMO will be user-centric, facilitating users to develop and deploy on top of NEMO. Moreover, NEMO will enable cloud and infrastructure providers to integrate their computing and networking resources into NEMO's infrastructure. The meta-OS architecture to realize this objective is defined through 8 architectural views, as detailed in D1.2 [1].

Following the defined functional architecture, the v0 of the NEMO integrated platform focusses on the specifications of the interactions among NEMO components. These are depicted in the high-level architecture presented in Figure 12. The development of the relevant functionality is materialized through the individual NEMO components, each of which is developed as cloud-native micro-service.

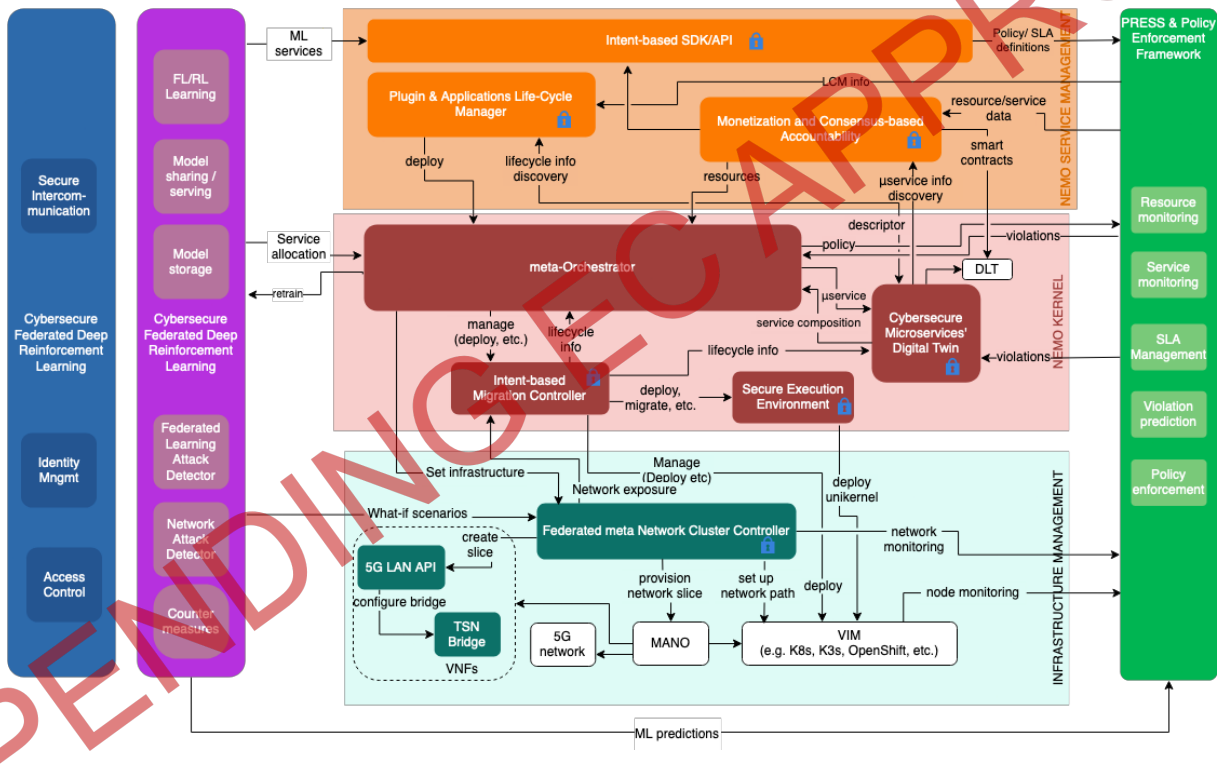


Figure 12: The NEMO high-level architecture

The main updates towards the developed NEMO functionality are provided in the next subsection.

4.1 Meta-OS functionality in NEMO v0

At this stage of the project, the development of NEMO components is in progress and initial versions have been released, as described in D2.1 and D3.1. The following subsections provide an overview of the NEMO functional layers and main logic supported in the current versions.

4.1.1 NEMO Infrastructure Management

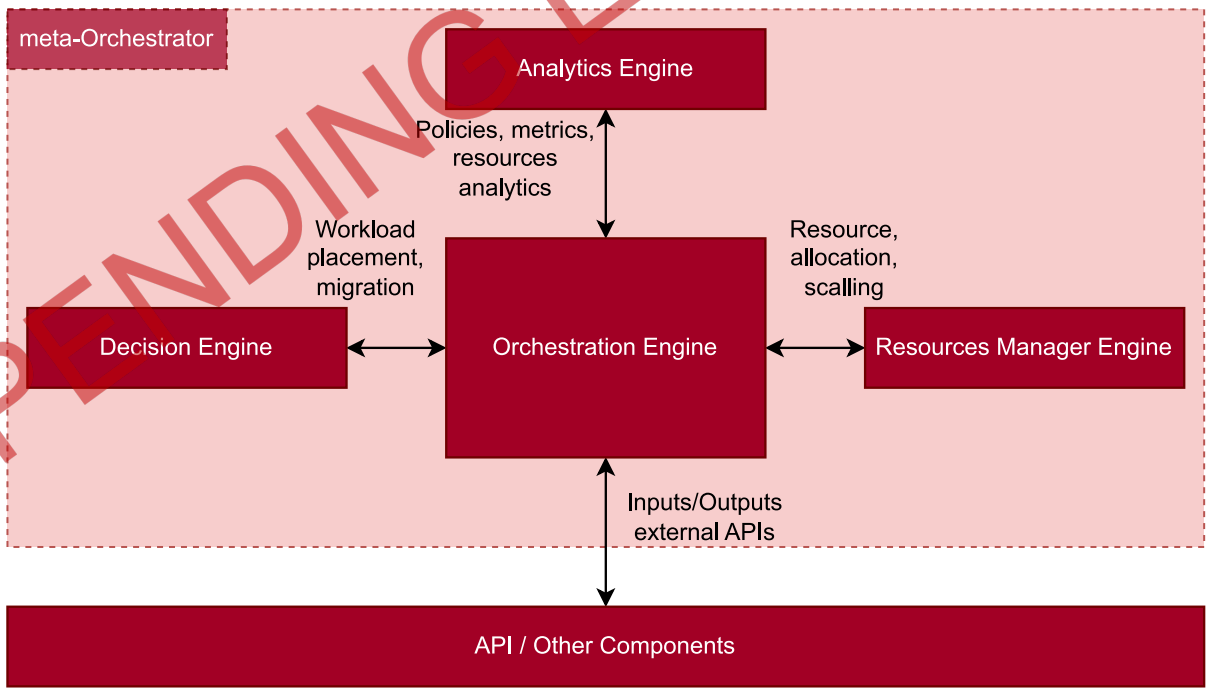
This layer deals with the management of infrastructural resources, mainly referring to the network level. It integrates and builds upon existing state-of-the-art solutions for infrastructure management, such as Kubernetes and MANO. This layer aims to support seamless connectivity among NEMO resources in a

secure and coherent way across all distributed heterogeneous nodes it may comprise, while respecting application requirements and without compromising privacy or usability. The Federated Meta-Network Cluster Controller (mNCC) delivers point-to-multipoint connectivity, manages connectivity resource reservation requests and supports network status assessment and prediction. Moreover, this layer integrates Time Sensitive Network (TSN) to support deterministic communication between wireless and fixed devices in the context of private 5G networks. Initial versions of these components are focused on the network exposure through a user-friendly API that will facilitate the development of applications and services on top of network resources. The API also considers the definition and communication of intents for application delivery over the network. Moreover, the TSN component is focused on intent-based slice creation though RESTful API on 5G Core prototype, able to create 5G LAN for private 5G networks, as prerequisite for delivering TSN functionality.

4.1.2 NEMO Kernel

This layer aims to support homogenized management of workloads across the IoT-edge-cloud continuum within the meta-OS. Central component is the meta-Orchestrator (MO), supported by the Cybersecure Microservices' Digital Twin (CMDT), the Intent-based Migration Controller (IBMC) and the Secure Execution Engine (SEE).

The meta-orchestrator (MO) system is designed to enable decentralized computing workflows across IoT-Edge-Cloud. It acts as a central orchestrator, managing complex distributed systems while optimizing resource utilization and improving scalability. Integrating and coordinating with different components within the distributed system architecture which facilitates interoperability and compatibility. The MO's intelligent decision-making capabilities ensure efficient coordination and resource management, contributing to the system's adaptability and efficiency.



The component developed and tested of the MO at this stage is:

Orchestration Engine: This central brain coordinates and manages complex computing resources to ensure seamless integration and simplification of distributed computing. Moreover, this central component enables the MO to manage and control the distributed computing workflow efficiently.

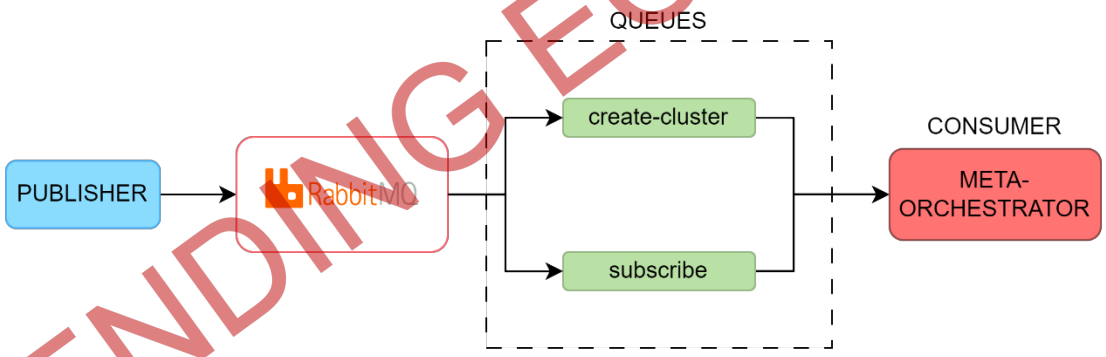
The other multifaceted components of the MO are presently in progress, as the developmental phase of the project is still in the design stage. Following the establishment of the communication and services, the remaining components will assume their respective roles and engage in seamless communication within the MO system, thus, their development and integration into the system are inevitable. These components are the following.

Analytics Engine: This component monitors and analyses performance metrics to identify bottlenecks and inefficiencies, providing insightful reports and visualizations that facilitate data-driven decision-making.

Resource Manager: The Resource Manager oversees the lifecycle of resources, including provisioning, scaling, monitoring, and de-provisioning services. It maintains a comprehensive view of system capabilities and optimizes resource usage and allocation to meet dynamic demands effectively.

Decision Engine: This central intelligence unit enforces policies, optimizes, and allocates workloads based on workload characteristics and performance metrics.

The current capabilities of the meta-orchestrator cover two main tasks: the automated creation and configuration of a multi-cluster environment, including the installation of Open Cluster Manager (OCM) to manage it, and a subscription service designed for deploying applications.
These tasks are triggered when the MO consumes a message from the designated queue in RabbitMQ. The message sent by the publisher contains the necessary information in YAML file format.



In the case of the cluster creation, the YAML file consists of a single field: "numberofclusters." When the MO consumes this message, it first creates a cluster designated as the "hub" cluster. Then, additional clusters are created according to the number specified in the YAML file and are joined to the hub as “managed clusters”.
Concerning the subscription task, the YAML file consists of three fields: "name," "namespace," and "pathname". The name and namespace fields denote the name of the application and the namespace where it will be deployed. The pathname references a Git repository, Helm chart, or S3 object containing the manifests for the deployment. Upon message consumption, the MO propagates the subscription to all managed clusters, which can then download directly from the storage location, hence performing the deployment. After the deployment, the repository will be monitored for new or updated resources.

The initial version of the NEMO Kernel includes specifications and basic functionality of the rest components of the NEMO kernel, too. CMDT supports a simple NEMO workload descriptor to facilitate the discovery, findability, coordination, and tracking of micro-service instances and unikernels. Moreover, cybersecurity aspects have been considered through the model of a Distributed Ledger Technology (DLT) at the core of the CMDT.

In addition, the SEE framework has been elaborated to enable migration of microservices, unikernels, and binaries to the IoT devices, the edge, or the cloud. One objective is to produce a stronger isolation by running common Linux containers or unikernels in virtual machines. In its first version, SEE integrates container runtime runh, which is based on the Open Container Initiative[8] (OCI) Runtime Specification for supporting both unikernels and Linux containers. Ongoing work is dedicated to incorporating support for Kata Containers [40].

4.1.3 NEMO Service Management

This layer acts as a DevZeroOps layer offering full-stack automated operations, greatest flexibility, improved developers' productivity and direct monetization and sustainability. It is composed of the NEMO Plugin & Applications Life-Cycle Manager (LCM), the Intent-based API/SDK and the Monetization and Consensus-based Accountability (MOCA) components.

The NEMO Plugin & Applications Life-Cycle Manager (LCM) aims to enable deployment of workloads, applications or plugins, onto NEMO, through the user's space, as well as support their lifecycle. The Intent-based API/SDK enables and facilitates third parties to interact with NEMO, exposing NEMO functionalities as programmatic APIs and providing software libraries for facilitating NEMO adoption and NEMO-compliant development. Moreover, the Monetization and Consensus-based Accountability (MOCA) aims to support monetization and accountability for both the applications and plugins running on NEMO, but also for network resources integrated into the NEMO infrastructure. The components of this layer are described in detail in section 5.

4.1.4 NEMO Cross-cutting Functions

The NEMO cross-cutting functions include:

- Cybersecurity and unified/federated access control, which ensures the security of metaOS operations across the metaOS layers, in the context of cloud native cybersecurity, federated access and identity management across the metaOS components, as well as secure and encrypted inter-process communication.
- Data & Services Policy Compliance Enforcement via multi-faced, policies able to cope with the different aspects of the applications life cycle (security, privacy, costs, environmental impact, etc). These functions ensure that PRESS rules and GDPR, as well as user-defined rules, are respected across the metaOS layers and components.
- Cybersecure Federated MLOps, which provides inherent integration of AI operations and services into the metaOS, yielding AI-based decisions and or controls alongside the metaOS. This function aims to support the complete Machine Learning (ML) lifecycle, e.g., from ML development and training to serving and inference performed within metaOS components, ensuring AI cybersecurity.

Towards cybersecurity, NEMO implements an Identity Management and Access control component. This component provides Authentication, Authorization and Accounting (AAA) services to NEMO components and defines roles to support RBAC rules, relying on Keycloak [41]. Moreover, it incorporates the NEMO Access Control module, which enforces custom and configurable controls on NEMO RESTful interfaces exposed to third parties. The Access Control module already integrates with the Identity Management. It also supports the Intent-based API in the workload provisioning process. In its first version, the Access Control may receive the list of exposable interfaces for a given components and automatically expose them, having configured and applied specified access policies. Moreover, the cybersecurity vertical includes the Intercommunication Management / Security module, which caters for secure interconnection among NEMO components. This module is based on RabbitMQ [42] and

[8] https://opencontainers.org/

current work includes fine-grained definition of internal NEMO interactions. Moreover, initial integrations have been developed for the meta-Orchestrator.
Policy Compliance is addressed by the PRESS & Policy Enforcement Framework (PPEF). PPEF aims to enforce privacy compliance applying GDPR principles, as well as workload policies defined in SLA fashion. PPEF collects information on policy compliance through a set of measurable metrics, which are analyzed to identify potential SLA violations. Both collected metrics and potential violation events are fed into the meta-Orchestrator to be considered in workload management. The initial version of PPEF includes collection of measurable metrics of diverse scope (e.g. node and service usage, energy consumption) based on Prometheus, Kepler [43], Scaphandre [44] and Pixie [45], as well as their federation into Thanos to support multi-cluster monitoring.
The Cybersecure Federated MLOps vertical is addressed through the Cybersecure Federated Deep Reinforcement Learning (CF-DRL) component. The first version of CFDRL focused on enhancing and speeding up the learning process of machine learning models in a collaborative and distributed way. It combines two learning paradigms, namely Federated Learning (FL) and Reinforcement Learning (RL). The NEMO solution is based on the Flower [46] Federated Learning (FL) framework, investigating the impact of secure aggregation mechanisms in federated learning environment. It also includes a proof of concept on applying FL in a simple RL environment. Moreover, CFDRL caters for cybersecurity during the learning process through malicious attack detection and mitigation. Initial work includes addressing privacy preservation challenges in knowledge aggregation and transfer, integrating Flower with the Private Aggregation of Teacher Ensembles (PATE) technique. Moreover, generative AI has been used to investigate and address attacks in Intrusion Detection System (IDS) data, normally belonging to FL nodes. IDS datasets have been synthetically created through a Generative Adversarial Network (GAN) and label flipping attack has been applied during FL training. Accordingly, a first approach on such attack detection has been developed vis a secure aggregation technique.

4.2 NEMO v0 PoC

This section presents early integration activities among the NEMO components.

4.2.1 Meta-Orchestrator: Orchestration Engine Component Test & Deployment

The code and the first deployment test can be found in the public and official repository of NEMO project.

To test locally, a docker container running RabbitMQ must be available:

```
docker run --hostname=my-rabbit --name rabbitmq -p 15672:15672 -d rabbitmq:3-management
```

For the local deployment, edit the YAML file under */test/cluster_create.yaml* to match *numberofclusters: 1* and run *main.go* to start listening for requests of cluster creation

```
go run main.go
```

Go to the test folder in a different terminal and run *cluster_create.yaml* to start the creation of the hub cluster and one managed cluster:

```
go run cluster_publisher.go
```

Subscription service

To use the subscription service, go to the integration-component folder and do:

```
go run subscription.go
```

In a different terminal, go to the test folder and run *subscription.yaml* to propagate the subscription to the repo indicated in the *subscription.yaml*:

```
go run subscription_publisher.go
```

4.2.2 NEMO Access Control: Component Deployment and Early Integration

The source code of the NEMO Access Control component can be found in the NEMO GitLab repository. It can be installed via the available Helm Chart, using the following commands.

```
$ git clone https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-cybersecurity-and-unified_federated-access-control/access-control/nemo-access-control.git
$ cd nemo-access-control
$ chmod +x install.sh
$ ./install.sh
```

The configuration of the different parameters is possible via the available JSON config file. The parameters required to set up the Access Control are described in Table 3.

Table 3: Access Control installation parameters

Installation parameters	Description	Example value
namespace	The namespace of the Kubernetes cluster where the Access Control will be deployed	nemo
password	The password for the PostgresDB	password
admin_node_port	The NodePort where the Admin API is available	31760
admin_gui_api_url	The (internal) URL where the Admin API is available	http://127.0.0.1:31760

The Admin API needs to run as a NodePort Service, so that the Kong Manager UI knows the IP address it runs on, in order to correctly retrieve the available resources (Services, Routes, Plugins).

Figure 13 shows the output of the installation script.

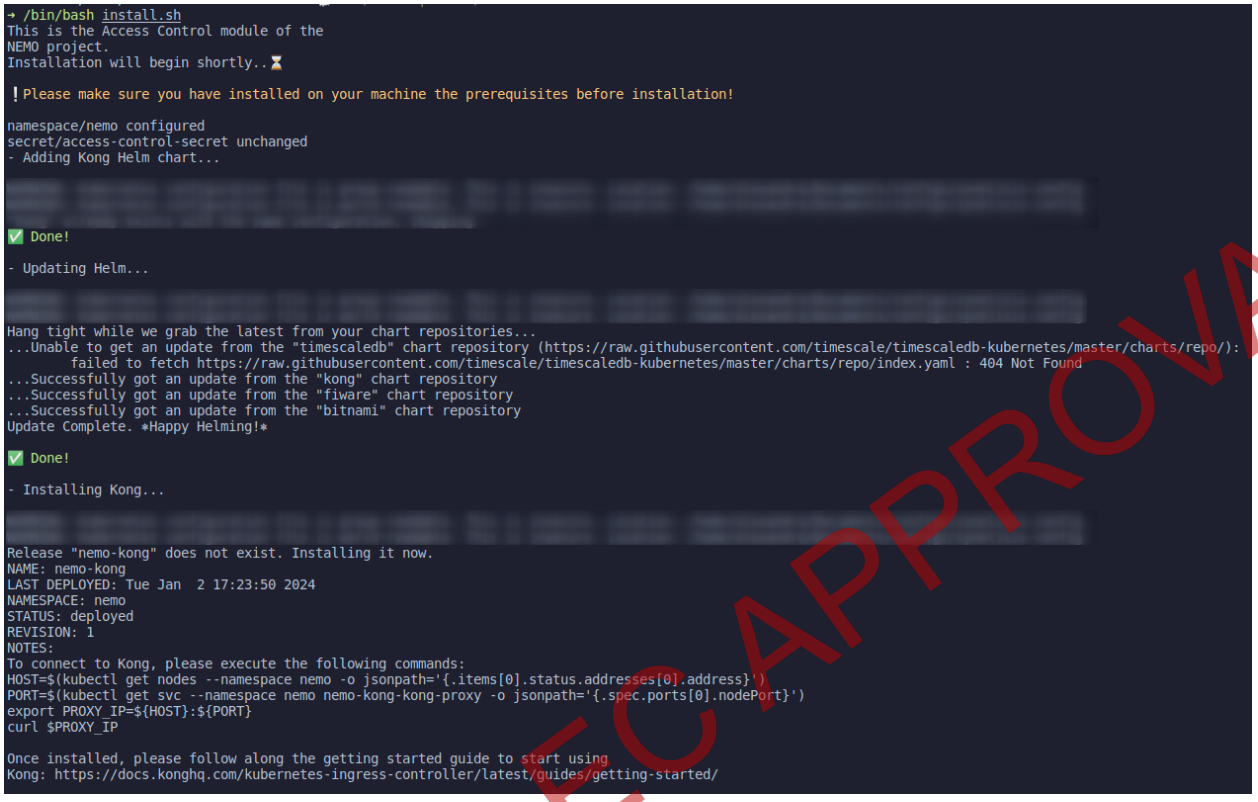


Figure 13: Access Control installation script output

If we check the Kubernetes resources, we can see that all the necessary components have been installed successfully (Figure 14).

Figure 14: Access Control Kubernetes Deployments

We can access the dashboard of the Access Control by checking the port the Kong Manager runs on (*nemo-kong-kong-manager*) (Figure 15, Figure 16).

Figure 15: Access Control Kubernetes Services

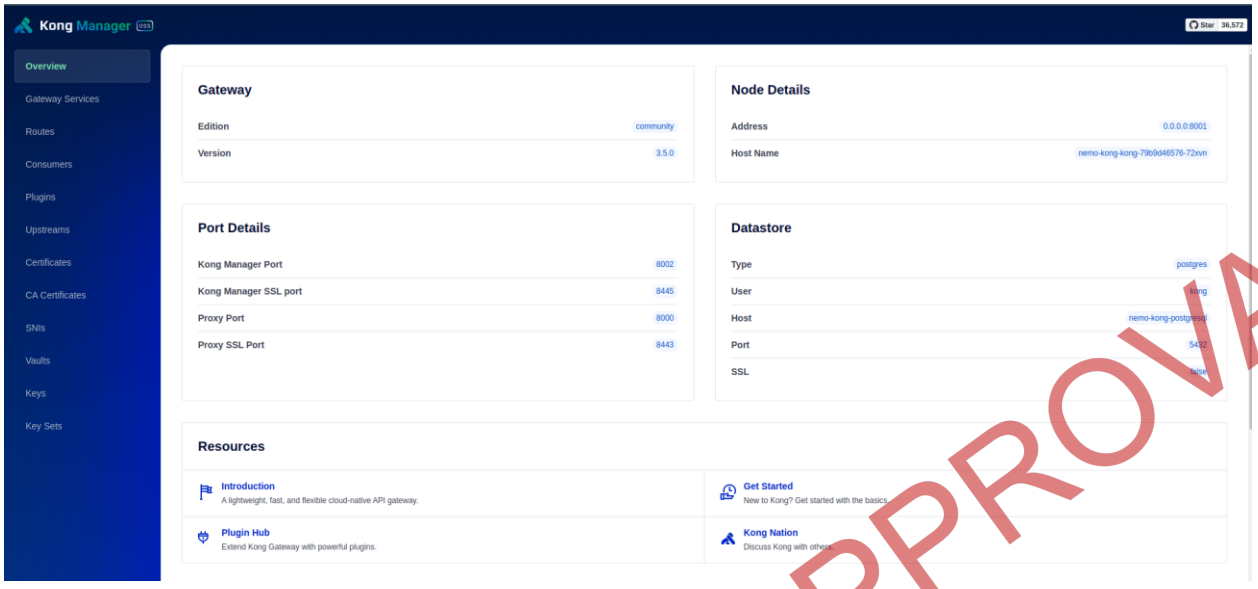


Figure 16: Kong Manager Dashboard

The integration of the NEMO Access Control component with the Intent-based API to support workload provisioning is implemented in via the "Automated Interfaces Exposure API" of *Workload Provisioning*. This is available in the NEMO GitLab repository and allows for the creation and exposure of new resources in the Access Control, based on input detailing the exposable interfaces.

The API can be installed using the following commands:

```
$ git clone https://gitlab.eclipse.org/eclipse-research-labs/nemo-project/nemo-service-management/intent-based-sdk_api/workload-provisioning.git
$ cd workload-provisioning
$ docker compose up –build
```

The API awaits for an object that has the following format.

```
{
    "host": "127.0.0.1",
    "port": "8000",
    "endpoint": "/register",
    "service_name": "test",
    "route_name": "test",
    "route_paths": ["/register"],
    "keycloak_client_id": "user",
    "keycloak_client_secret": "secret",
    "keycloak_realm": "nemo"
}
```

The role of each variable is explained in Table 4.

Table 4: Automated Interfaces Exposure API payload parameters

Payload Parameters	Description	Example
host	The IP of the service that will be protected from Access Control	127.0.0.1
port	The port of the service that will be protected from Access Control	8000
endpoint	The endpoint of the service that will be protected from Access Control	/register
service_name	The name of the Kong Service	test
route_name	The name of the Kong Route	test
route_paths	The endpoints which will be exposed by the Access Control instead of the real service endpoint	[“/register”]
keycloak_client_id	The Keycloak Client ID for the service (required to enable the Keycloak plugin)	user
keycloak_client_secret	The Keycloak Client secret for the service (required to enable the Keycloak plugin)	secret
keycloak_realm	The Keycloak Realm for the service (required to enable the Keycloak plugin)	nemo

Figure 17 shows the usage of the API’s Swagger, in order to create the Kong resources.

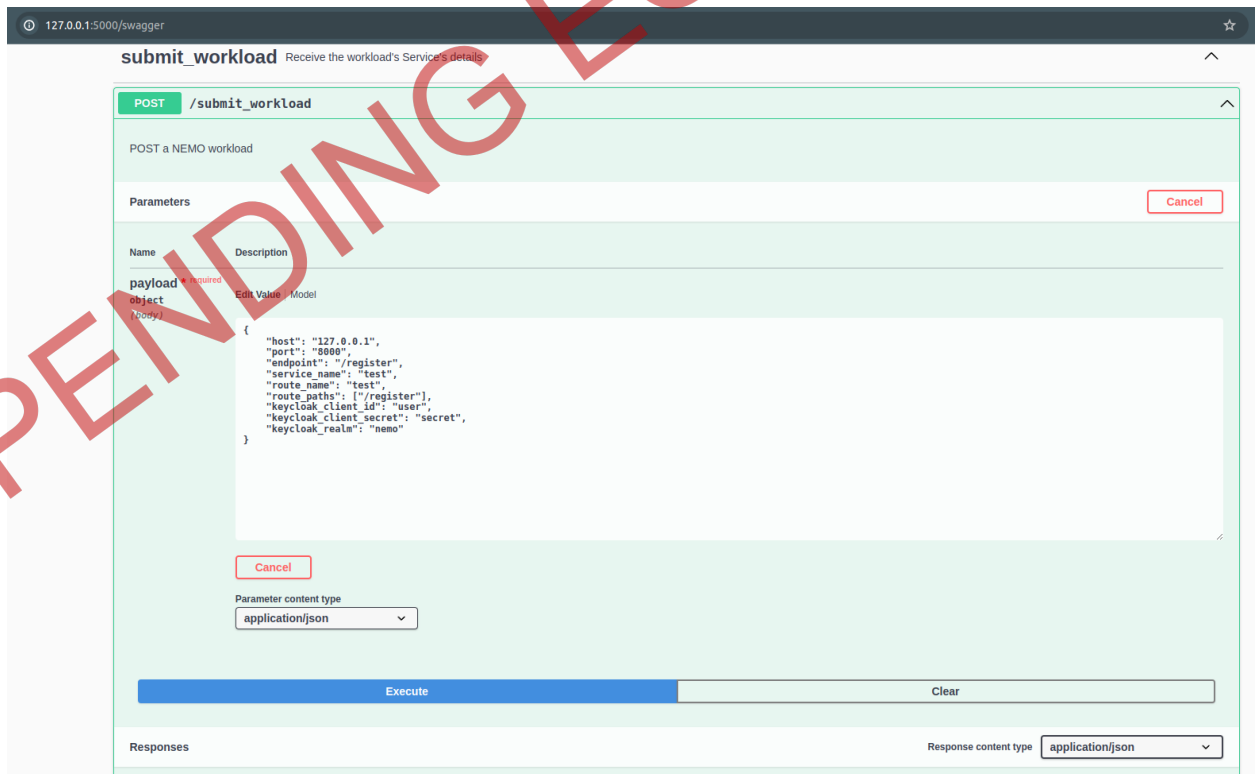


Figure 17: Automated Interfaces Exposure API Swagger - Request

Figure 18 shows the response of the API. The request was successful and a Kong Service, Route and Kong Plugin were created.

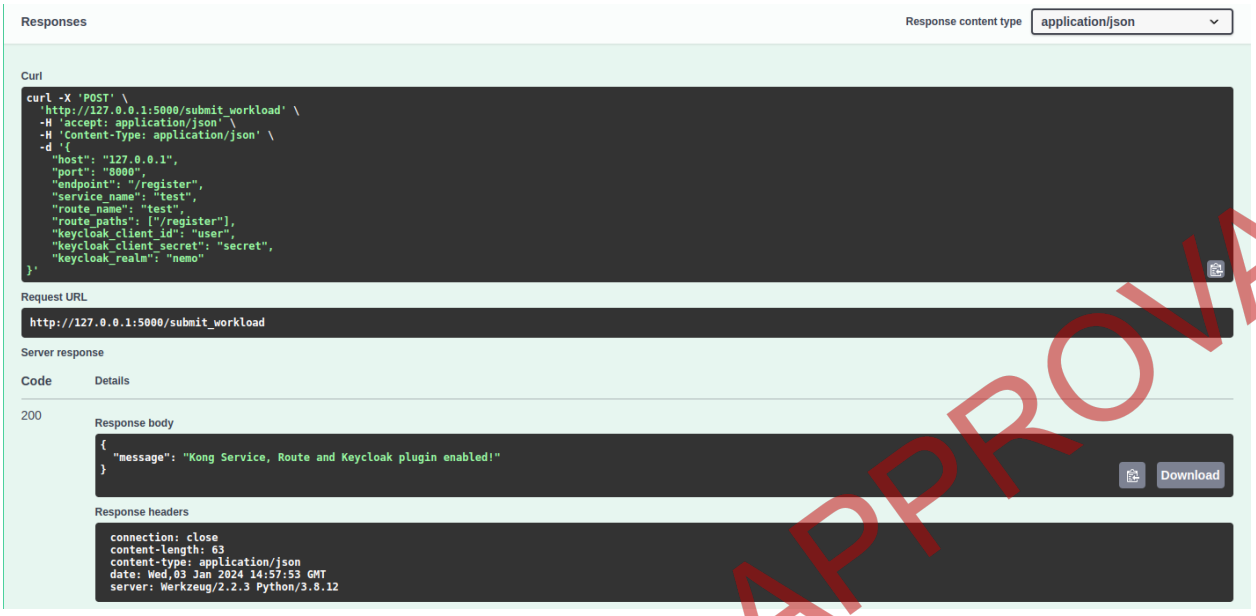


Figure 18: Automated Interfaces Exposure API Swagger - Response

Figure 19 shows the logs of the API Server. We can see the details of the objects created in Kong.

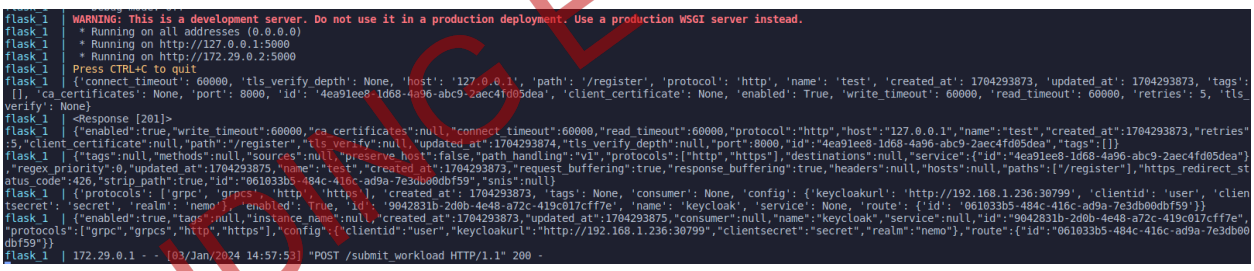


Figure 19: Automated Interfaces Exposure API logs

In the Kong Manager Dashboard, we can see that the Service "*test*" was successfully created and its details (Figure 20, Figure 21).

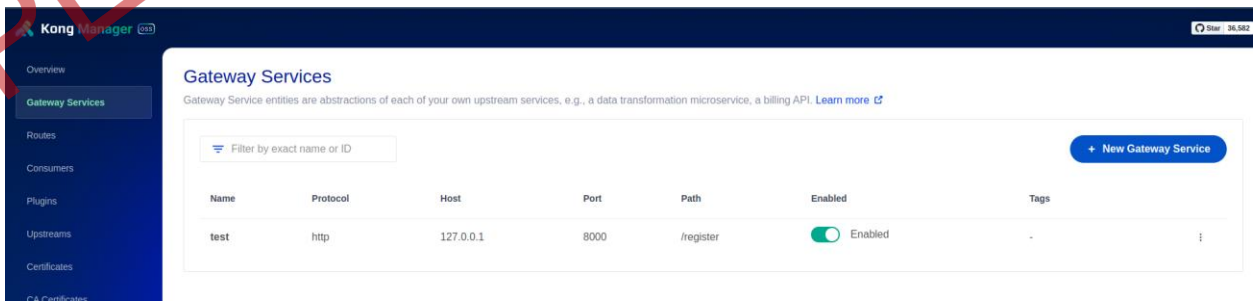


Figure 20: Kong Manager - Service

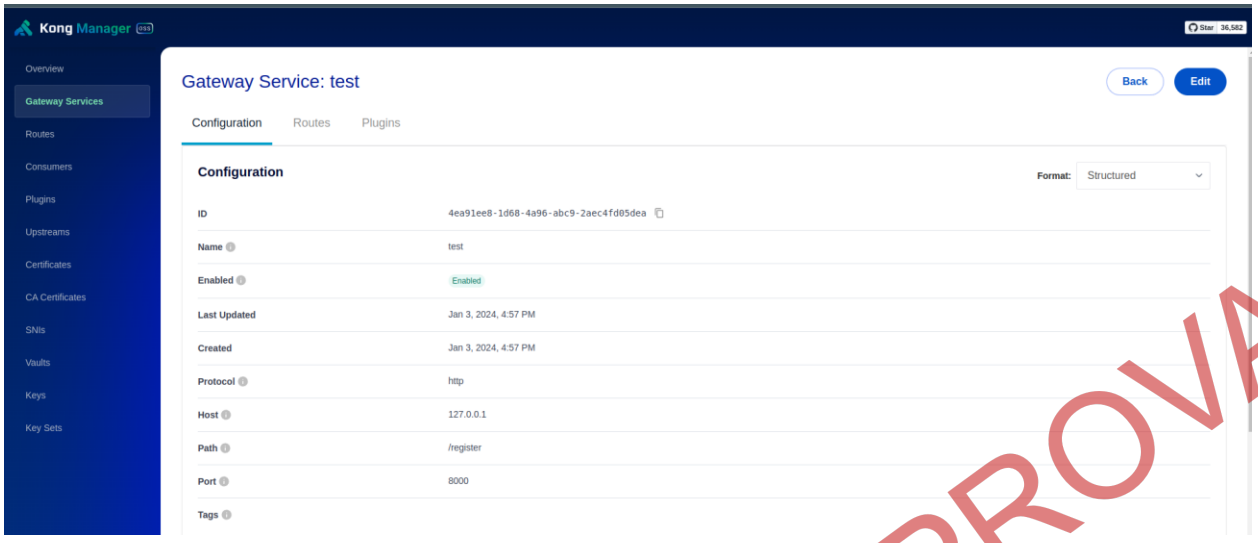


Figure 21: Kong Manager - Service details

Figure 22 and Figure 23 show the "*test*" Kong Route and its details.

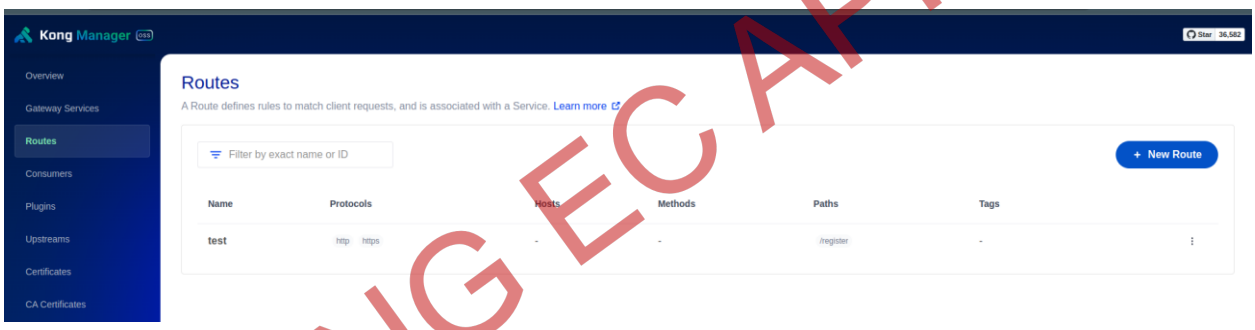


Figure 22: Kong Manager – Route

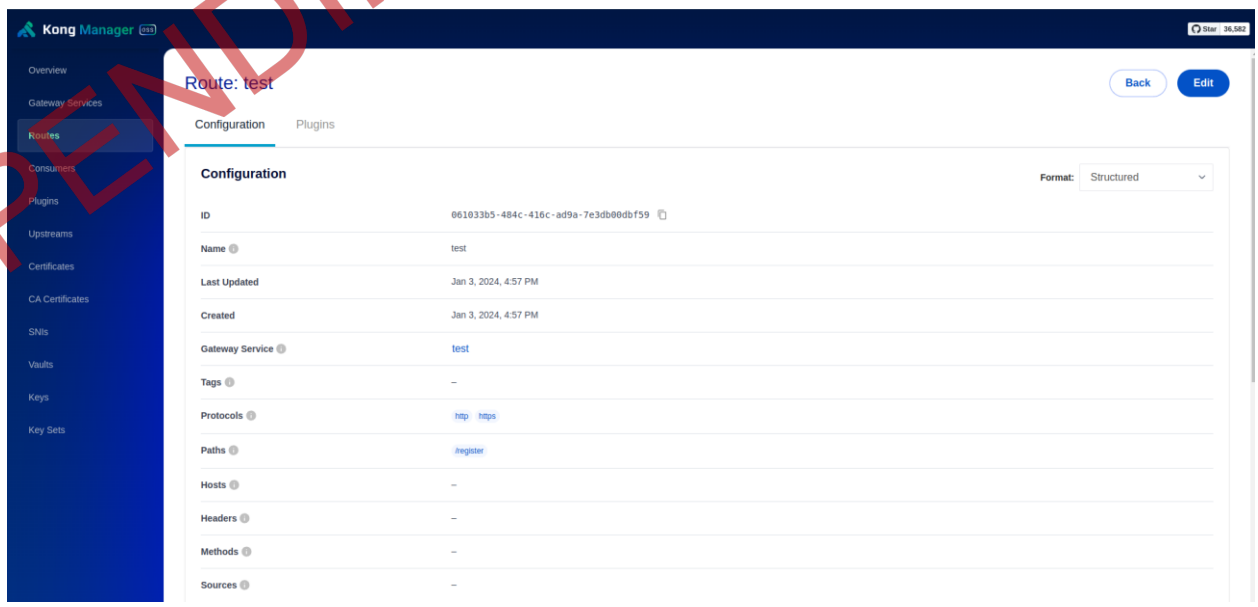


Figure 23: Kong Manager – Route details

Figure 24, Figure 25 and Figure 26 show the Keycloak plugin that was enabled for the "*test*" Route and its details.

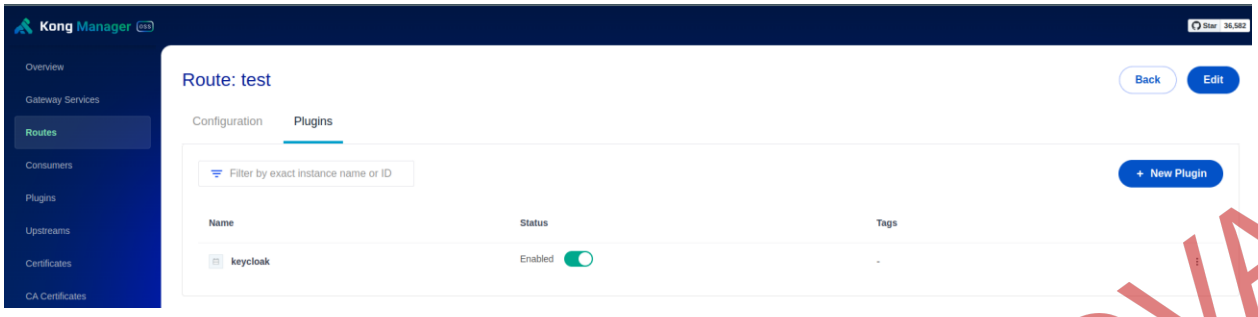


Figure 24: Kong Manager – Keycloak plugin

Figure 25: Kong Manager – Keycloak plugin details (1)

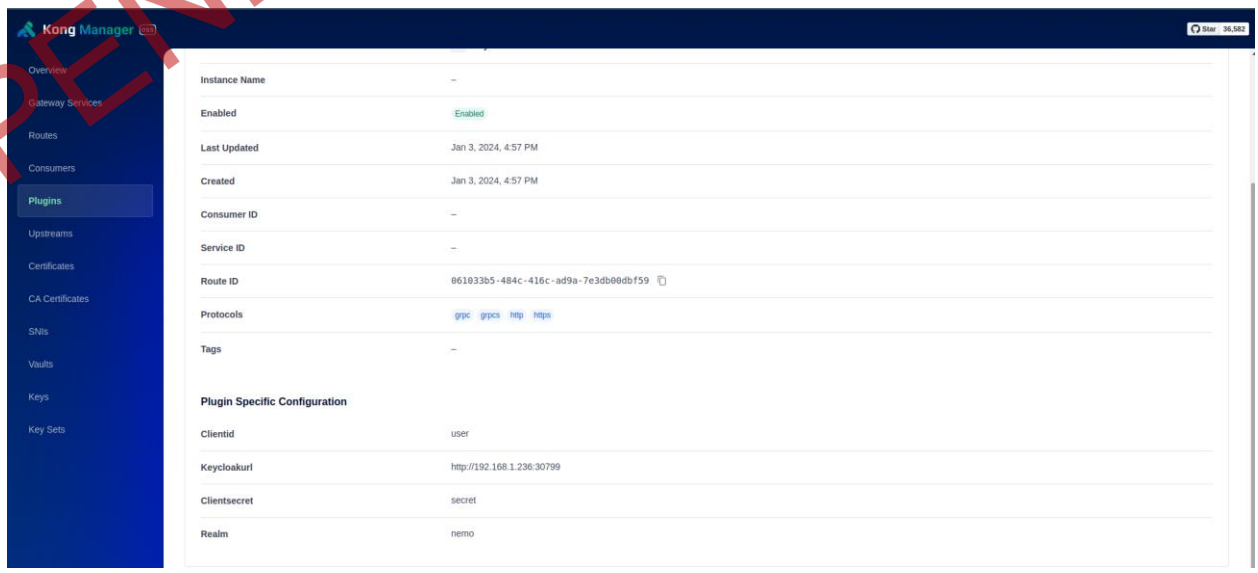


Figure 26: Kong Manager – Keycloak plugin details (2)

5 NEMO components towards third-party integration

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

5.1 Intent-based Migration Controller

5.1.1 Overview

The Intent-based Migration Controller (IBMC) plays a pivotal role within the NEMO ecosystem, specifically tailored to facilitate the seamless migration of computing workloads across the dynamic landscape of the IoT to Edge to Cloud Continuum. It operates as a critical component that leverages intent-based networking principles, ensuring optimal resource utilization, scalability enhancement, and uninterrupted service delivery throughout the migration process.

In leveraging intent-based networking principles, the IBMC ensures optimal resource utilization, scalability enhancement, and the uninterrupted delivery of services throughout the migration process. This paradigm allows the IBMC to interpret and act upon high-level migration intents, contributing to the adaptability crucial for navigating the complexities of the meta-OS environment.

5.1.2 Background

The development of the Intent-Based Migration Controller (IBMC) is rooted in the intricate landscape of contemporary distributed computing and the evolving meta-OS framework. As computing environments transition towards decentralized architectures, the imperative for efficient and adaptable workload migration mechanisms becomes pronounced.

In the realm of Intent-Based Networking (IBN), the IBMC aligns itself with state-of-the-art technologies that harness the power of automated decision-making driven by high-level user intents. Leveraging advancements in machine learning, artificial intelligence, and orchestration frameworks, the IBMC exemplifies the cutting edge of intelligent workload migration within the meta-OS paradigm.

Examples of such technologies include the use of reinforcement learning algorithms to predict optimal migration paths based on historical data, and the application of containerization technologies like Docker[9] and orchestration platforms like Kubernetes[10] to encapsulate and manage migrating workloads.

Kubernetes, although not inherently intent-based, offers mechanisms for service and workload migration. In Kubernetes, services can be migrated using rolling updates[11], where new instances of a service are gradually introduced while phasing out the old ones. Alternatively, blue-green[12] deployments enable the seamless switch between two environments, ensuring continuous service availability during migration.

[9] https://www.docker.com/
[10] https://kubernetes.io/
[11] https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/
[12] https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment

In the context of microservices, Kubernetes provides tools like Helm[13] charts for packaging, and tools like Istio[14] for service mesh architecture, offering sophisticated traffic management during migration. Alternatives such as Docker Swarm[15] and Apache Mesos[16] also present different approaches to microservices orchestration and migration.

While focusing on Kubernetes seamless migration of services or clusters, the literature brings us into the tool Velero[17], used *de facto*. The approach of this framework is to schedule backups for the desired services that are stored into S3 objects. Then, the user utilizes the restore functionalities to deploy the services into a running targeted cluster. The aim of this backup-restore is to be aligned with the blue-green approach.

The IBMC, building upon these foundations, introduces intent-based principles to migration, allowing users to express high-level objectives, constraints, and requirements. This marks a departure from traditional migration methods, infusing a level of abstraction that aligns with the meta-OS philosophy.

5.1.3 Architecture & Approach

5.1.3.1 Architecture

Situated within the meta-Architecture framework (MAF) stated in D1.2, the architectural design of the Intent-Based Migration Controller (IBMC) seamlessly integrates into the NEMO Kernel and Continuum. At its foundation, the IBMC adopts a modular structure comprising pivotal components working collaboratively to ensure the efficient migration of workloads.

Development View

Figure 27 illustrates the components underneath the Development View of the IBMC.

- Intent Interpretation Module: The IBMC incorporates an Intent Interpretation Module tasked with comprehending and extracting user-defined migration intents. This module leverages advanced techniques such as natural language processing and semantic analysis to translate high-level objectives, constraints, and requirements into actionable instructions.
- Migration Engine: Positioned at the core of the IBMC, the Migration Engine takes the translated migration intents and formulates a comprehensive plan for the migration process. It considers various factors, including workload characteristics, network conditions, resource availability, and latency requirements. This holistic approach ensures informed and strategic decision-making throughout the migration.
- Delivery Module: Dedicated to encapsulating and disseminating information pertinent to the migration process, the Delivery Module plays a crucial role in communicating with other components affected by the migration. It receives data provided by the Migration Engine and establishes communication with the meta-Network Cluster Controller (mNCC). This communication aligns with the Intent-Based Networking (IBN) paradigm, contributing to a cohesive and synchronized migration process.

[13] https://helm.sh/
[14] https://istio.io/
[15] https://docs.docker.com/engine/swarm/admin_guide/
[16] https://mesos.apache.org/documentation/latest/high-availability-framework-guide/
[17] https://velero.io/docs/v1.12/

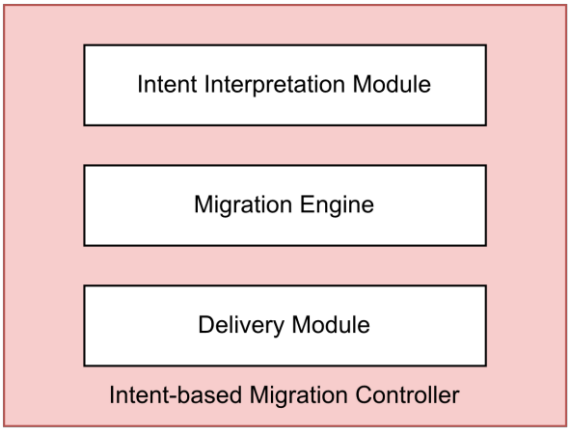


Figure 27: Development view of IBMC

Process View

The Process View is designed to delineate the workflow for a cluster migration within the contextual framework of NEMO. As illustrated in Figure 28, the blue-green approach is proposed for the seamless integration of the Intent-Based Migration Controller (IBMC) into the migration process. The sequential steps involved in migrating the cluster are explicated as follows:

1. The candidate "cluster(1)" initiates a proactive backup of services and configurations, storing them in an S3 bucket.
2. Simultaneously, the PRESS and Policies Enforcement Framework (PPEF) along with the Cybersecure Federated-Deep Reinforcement Learning (CF-DRL) are entrusted with monitoring the cluster. They provide contextual information regarding Quality of Services (QoS) aligned with the Service Level Agreement (SLA).
3. If any Service Level Objective (SLO) either exceeds the threshold of an anticipated value or is predicted to do so, an intent file is triggered and transmitted to the meta-Orchestrator (MO).
4. The MO processes these intents, typically in YAML file format, through the Integration Component. Subsequently, it issues a request to the meta-Network Cluster Controller (mNCC) to retrieve available cluster candidates for service migration.
5. The mNCC provides a cluster ID, enabling the IBMC to initiate the migration of services. Thus, the "cluster(2)" is chosen as the target one.
6. The extraction of the full intent, which includes the cluster ID and metadata, is facilitated through the Intent Interpretation Module.
7. The Migration Engine computes the migration process, necessitating a restore from the S3 bucket to retrieve all information associated with "cluster(1)". This module encapsulates migration-related information, such as SLO violations and metadata, creating a feedback loop to be shared with the CF-DRL module. The objective is to enhance the models for predicting service degradation.
8. Finally, the Delivery Module deploys the services into "cluster(2)". If applicable, "cluster(1)" can subsequently remove the running services.

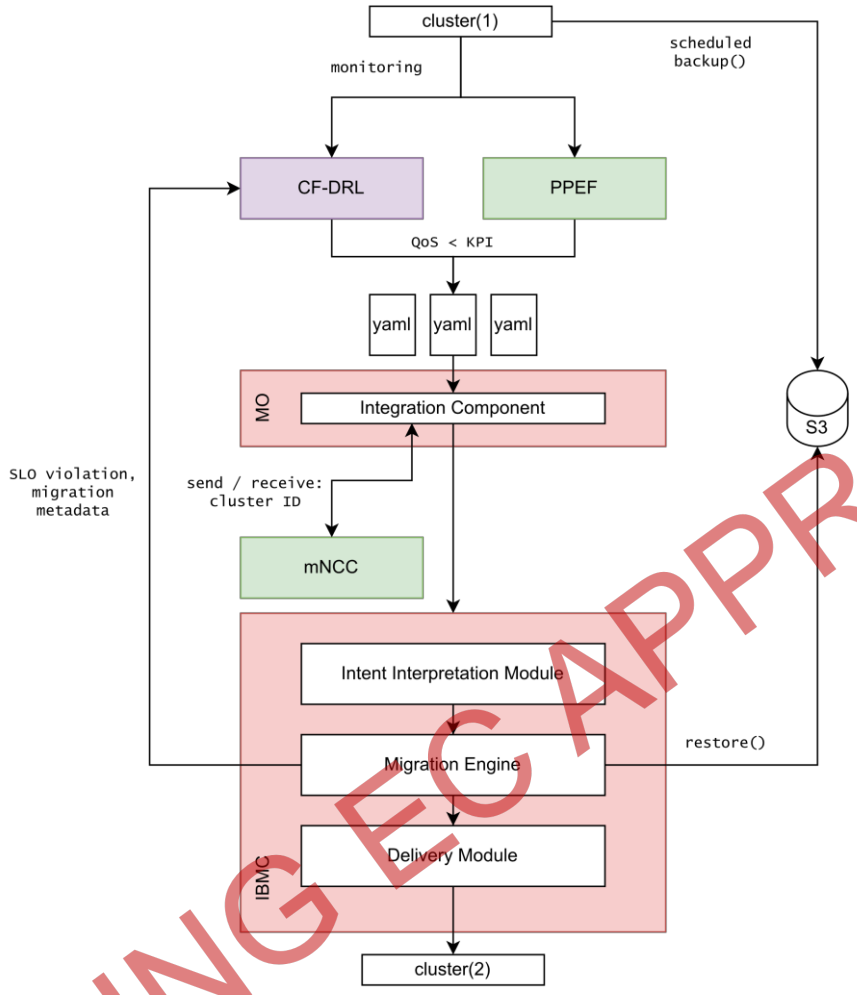


Figure 28: Workflow of a migration of clusters utilizing the IBMC

5.1.3.2 Approach

The approach taken by the IBMC is deeply rooted in the paradigm of Intent-Based Networking (IBN). Unlike traditional migration methods, the IBMC introduces a higher level of abstraction, allowing users to express migration intents at a semantic level rather than delving into intricate technical details. This approach brings about several key facets:

- Intent Abstraction: Users interact with the IBMC through high-level intents, expressing goals, constraints, and desired outcomes. This abstraction shields users from the complexities of migration intricacies, empowering them to focus on strategic objectives.
- Dynamic Decision-Making: The IBMC employs dynamic decision-making, adapting its migration strategies in real-time based on the evolving state of the distributed system. This is achieved through continuous analysis of environmental factors, workload conditions, and historical performance data.
- Feedback Loop Integration: To enhance its adaptive capabilities, the IBMC incorporates a Feedback Loop. This loop collects and analyzes feedback from the migration process, incorporating insights into future decision-making (CF-DRL component). It ensures a continuous improvement cycle, aligning the IBMC with the evolving needs of the meta-OS environment.

5.1.4 Interaction with other NEMO components

5.1.4.1 Cybersecure Federated-Deep Reinforcement Learning (CF-DRL)

The IBMC collaborates closely with the CF-DRL module during the cluster migration process. This collaboration is initiated when CF-DRL, along with PPEF, monitors the cluster and provides contextual information on SLAs. The IBMC, upon encountering SLO violations during the migration, establishes a dynamic feedback loop with CF-DRL. This interaction facilitates the exchange of information crucial for refining models that predict service degradation, contributing to an adaptive and intelligent migration strategy.

5.1.4.2 Meta-Network Cluster Controller (mNCC)

The IBMC's interaction with the mNCC is pivotal in determining suitable cluster candidates for service migration. The IBMC communicates with the mNCC through the Integration Component. This interaction results in the retrieval of a cluster ID, a key identifier enabling the IBMC to initiate the migration process effectively.

5.1.4.3 Meta-Orchestrator (MO)

The meta-Orchestrator (MO) plays a central role in the IBMC's workflow. The MO receives migration intents triggered by SLO violations or predictions. These intents are processed through the Integration Component.

5.1.4.4 PRESS and Policies Enforcement Framework (PPEF)

The IBMC engages with the PPEF to monitor the cluster's health and context. The information provided by PPEF contributes to the overall understanding of the cluster's state, guiding the IBMC in its migration decision-making process. This interaction ensures that the migration aligns with established policies and enforcement frameworks, fostering a secure and compliant cluster transition.

5.1.5 Conclusion, Roadmap & Outlook

The IBMC within the meta-OS framework stands as a significant advancement in the NEMO platform. It contributes to the seamless migration of workloads across the IoT to Edge to Cloud Continuum, maintaining a dynamic equilibrium within the meta-OS environment.

Looking ahead, the roadmap for the IBMC involves continuous refinement and adaptation based on evolving requirements and technological advancements within the meta-OS paradigm. Future directions and potential enhancements are outlined to ensure the IBMC remains at the forefront of migration capabilities within the evolving landscape of meta-Operating Systems.

5.2 Plugin & Applications Lifecycle Manager

5.2.1 Overview

The Plugin & Applications Lifecycle Manager (LCM) is a flexible mechanism for unified, just-in-time plugins and applications life cycle management across the NEMO ecosystem. The Lifecycle Manager (LCM) will be the interface between the NEMO ecosystem and the NEMO users, providing an interface for seamless deployment of workloads (services, applications, plugins) in the NEMO ecosystem. In addition, LCM will check for available updates/bug fixing and install them over the air.

While a workload is running in NEMO meta-OS an event-based mechanism monitors critical events related to the performance of the service. Moreover, a security controller monitors security related events, alerts the user for detected abnormalities, and applies mitigation actions based on specified cyber

threats. Finally, the UI of the LCM will incorporate interfaces to other NEMO components like Intent-based API and CMDT.

5.2.2 Background

While initially we have chosen Jenkins framework as the base for core LCM functionalities, the dominant role of Kubernetes in the NEMO meta-OS revealed the need of choosing a framework that is more appropriate to this collaboration. To this end us we have considered ArgoCD and Flux to automate continuous delivery and lifecycle management of NEMO workloads. Based on the aforementioned frameworks, LCM enables software releasing continuously by interacting with various testing and deployment methods, promoting easy integration and deployment of workloads. In addition, we extend visualization capabilities employing the LCM Visualization component.

The LCM Visualization component builds upon Advanced Visualization Toolkit (AVT[18]), which is a sophisticated set of visualization tools, capable of connecting with state-of-the-art streaming platforms and databases, while it can be tuned to consume different kinds and formats of data, providing essential, interactive and user-friendly visualizations. The AVT is provided as a framework that consists of a web application, accompanied by a dedicated server based on Node.js[19] to handle all the external connections, real time messaging and user management. Both components are provided as docker containers with two separate docker images. The AVT front-end is based on the Angular.io[20] framework in conjunction with a set of visualization libraries that are interchangeably used to cover specific needs and particularities of the analyzed data.

ArgoCD

ArgoCD [10] is a top-tier continuous delivery and GitOps tool designed specifically for Kubernetes. Embracing the GitOps methodology, ArgoCD automates the deployment process by ensuring that the state of applications in a Kubernetes cluster aligns with configurations stored in Git repositories. Its strengths lie in declarative application definition, allowing users to articulate Kubernetes manifests in a version-controlled manner.

ArgoCD offers features such as automated synchronization, rollback capabilities, and support for multi-environment deployments. With an intuitive web interface and native Kubernetes integration, ArgoCD streamlines the deployment lifecycle, fostering collaboration, traceability, and efficiency.

Argo also facilitates visualization and analysis of deployment status, making it an efficient and resource-conscious choice for organizations adopting Kubernetes-centric continuous delivery practices.

In short, some interesting ArgoCD capabilities include:

- Automated deployment of applications to specified target environments.
- Ability to manage and deploy to multiple clusters.
- Multi-tenancy and RBAC policies for authorization.
- Rollback/Roll-anywhere to any application configuration committed in Git repository.
- Health status analysis of application resources.
- CLI (Command Line Interface) for automation and CI integration.
- Webhook integration (GitHub, BitBucket, GitLab)

[18] https://aegisresearch.eu/solutions/advanced-visualization-toolkit/

[19] https://nodejs.org/en/

[20] https://angular.io

- Audit trails for application events and API calls.
- Parameter overrides for overriding helm parameters in Git.

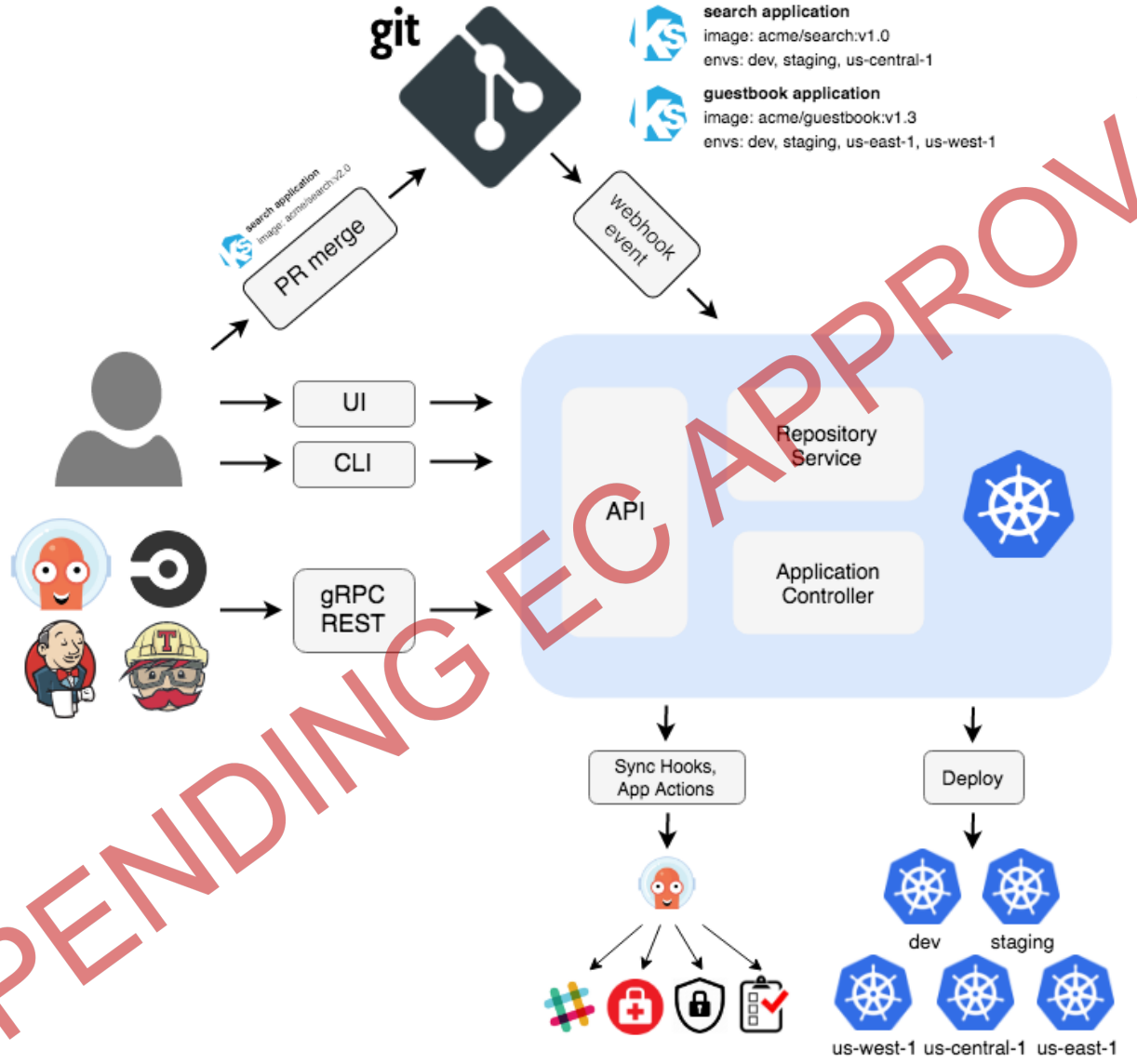


Figure 29: ArgoCD Architecture

Flux

Flux [11] is an open-source tool designed for automating continuous delivery through a GitOps workflow for Kubernetes applications, which has been considered for automating deployments of the NEMO framework per se and has, thus, been introduced in section 3.2.2 from that perspective. The analysis here aims to identify benefits of Flux for the NEMO LCM development.

Flux ensures that the desired state of applications in a Kubernetes cluster aligns seamlessly with configurations defined in declarative YAML files, by leveraging version-controlled manifests stored in Git repositories.

With its commitment to GitOps best practices, Flux enhances collaboration, auditability, and consistency, making it a powerful asset for streamlining the deployment and lifecycle management in complex Kubernetes environments.

Key features include automated synchronization, rollback support, and event hooks. Flux promotes scalability in microservices architectures, facilitating the efficient deployment, update, and scaling of services.

Some interesting features of Flux also include:

- GitOps for both apps and infrastructure.
- Application deployment (CD) and progressive delivery (PD) through automatic reconciliation.
- Compatibility with all major Git providers and container registries.
- Adherence to Kubernetes security policies and tight integration with security tools and best-practices.
- Provision of health assessments and alerting to external systems.
- Flux works with Kubernetes role-based access control (RBAC)

Flux is constructed with the GitOps Toolkit components, which is a set of specialized tools and Flux Controllers, composable APIs, reusable Go packages for GitOps under the fluxcd GitHub organisation for building Continuous Delivery on top of Kubernetes.

Falco

Falco [47] is a cloud-native security tool designed for Linux systems. Generally, it employs custom rules on kernel events, enriched with container and Kubernetes metadata, to provide real-time alerts. Essentially, Falco consumes Linux kernel system calls and enriches these events with information from Kubernetes and the rest of the cloud native stack, thus providing real-time detection capabilities for environments ranging from individual containers to Kubernetes and the cloud.

Falco collects event data from a source and compares each event against a set of rules. An indicative list of Falco's sources includes Linux kernel system calls, Kubernetes audit logs and cloud events, however it is possible to expand Falco data sources via the development of appropriate plugins. As far as the rules are concerned, naturally Falco comes with a thorough set of rules covering container, host, Kubernetes, and cloud security, however, as expected, custom rules can be created as well.

Furthermore, Falco is highly scalable due to its containerized architecture and tight integration with Kubernetes, as realized by Figure 30. Most importantly, it runs as a Kubernetes daemon set, ensuring that every node in a cluster is monitored, while its integration with Grafana and Prometheus allows for the visualization and analysis of the produced alerts at scale. We should also notice that Falco is highly performant and keeps its footprint small via the usage of a minimal set of resources, hence it is not expected to induce cluster costs in terms of resource and energy consumption, as well as efficiency.

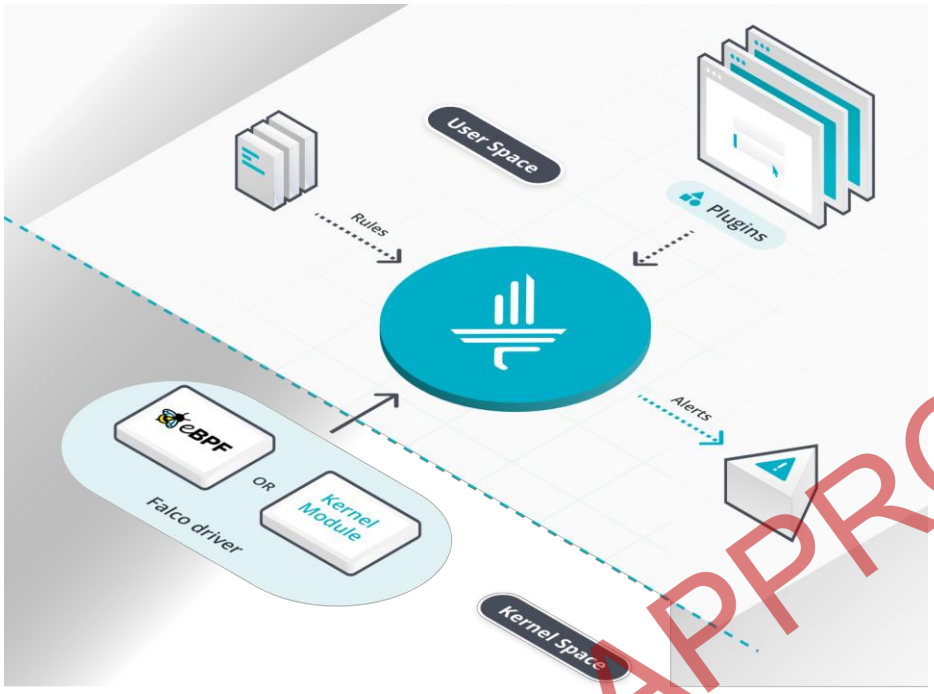


Figure 30: Falco Architecture

Trivy Operator

Trivy Operator [48] is a Kubernetes operator that allows the continuous scanning of a Kubernetes cluster for security issues. The operator constantly monitors Kubernetes for state changes and automatically triggers security scans in response. The operator automatically generates and updates a variety of reports, including, but not limited to:

- **Vulnerability scans.** Automatically scans Kubernetes workloads, control-plane and node components for known vulnerabilities.
- **ConfigAudit scans.** Configuration audits for Kubernetes resources with predefined rules or policies.
- **Exposed Secret Scans.** Scans for exposed secrets with the cluster.
- **Role Based Access Control (RBAC) scans.** It provides information on the access rights of the different resources that are installed.
- **Software Bill of Materials (SBOM)**

The trivy-operator can easily be installed in a Kubernetes cluster with Helm. Figure 31 shows an overview of the trivy-operator's high-level architecture.

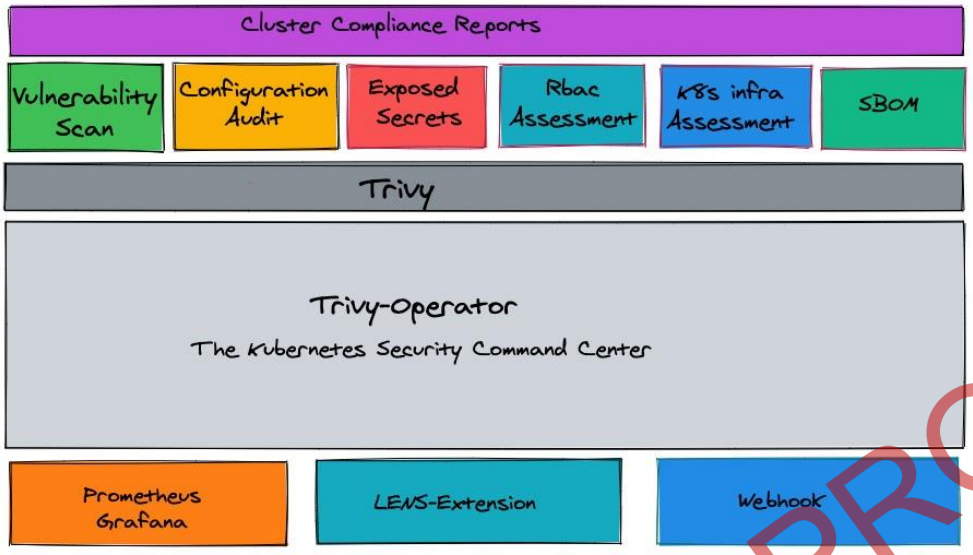


Figure 31: Trivy-operator overview

5.2.3 Architecture & Approach

The role of LCM in high-level architecture of NEMO meta-OS is depicted in Figure 32.

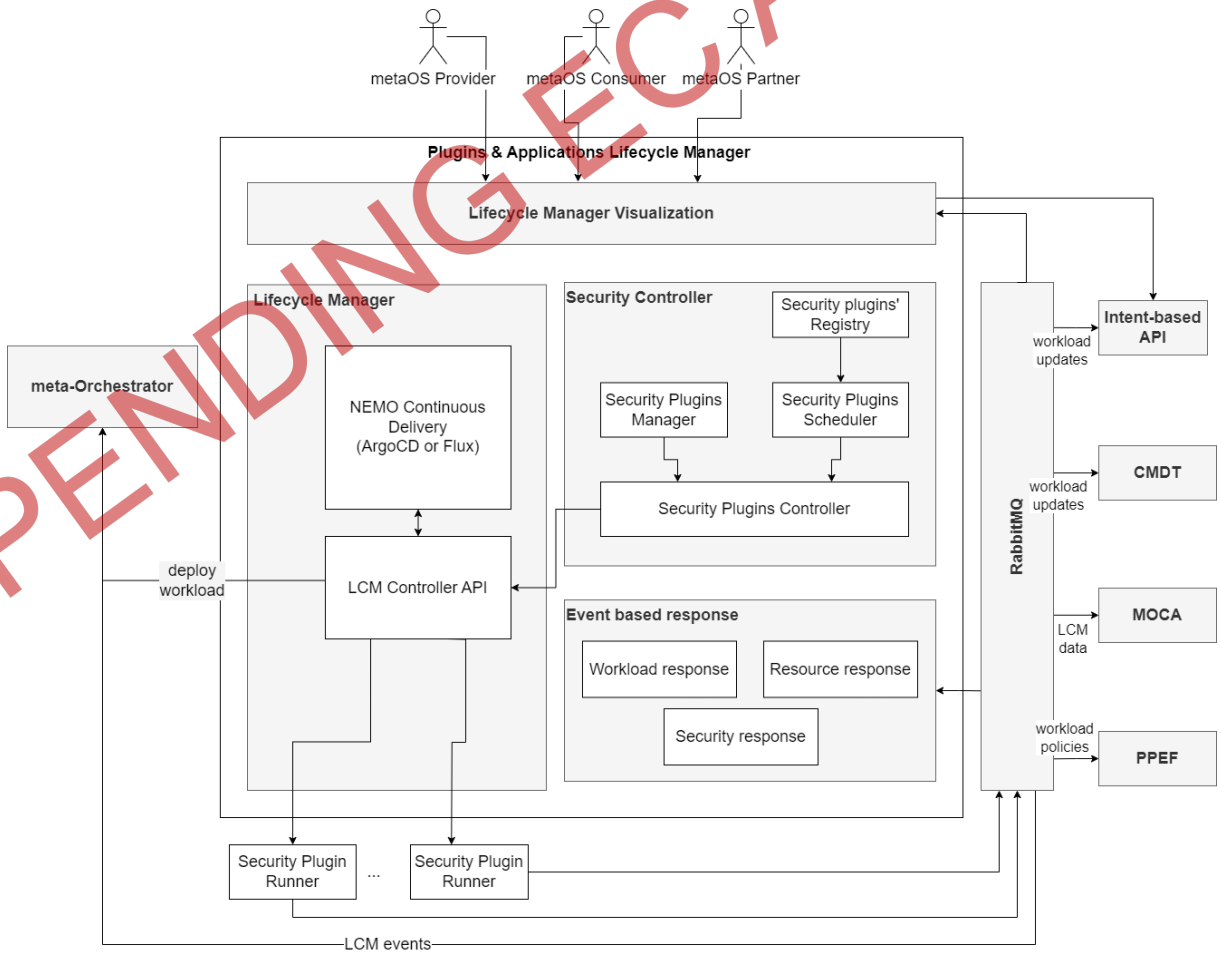


Figure 32: LCM Architecture

5.2.3.1 Lifecycle Manager

LCM employs a continuous deployment framework (Flux or ArgoCD) to monitor the lifecycle and manage a registered workload based on its requirements/manifest. The NEMO LCM will gain ingress access rights, download the necessary plugins and associated dependencies on demand, and install them on the devices while checking for security warnings.

Functionalities of LCM are enhanced by a custom developed controlling mechanism, LCM Controller API, which promotes communication with other LCM submodules as well as with the NEMO ecosystem, providing endpoints to send and receive information (e.g., Register a workload in internet-based API, receive registered workloads list for intent-based API).

5.2.3.2 Security Controller

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

In order to address security and vulnerability scanning needs, the Security Controller follows a plugin based architecture, as depicted in Figure 32 and is composed of the following subcomponents:

- Security Plugins' Scheduler: Schedules security/vulnerability scans, either on demand, or following a schedule.
- Security Plugins' Controller: Orchestrates the spawning of security/vulnerability scanning jobs of specific type (e.g. system calls' scans or image/registry/dependencies' scans)
- Security Plugins' Manager: Maintains a list of scanning plugins, each one scanning NEMO resources for a particular grouped set of vulnerabilities.
- Security Plugin Runner: Performs a security or vulnerability scanning of defined level, e.g. system calls' scanning, container images' scanning, etc.

The security plugins follow common specifications regarding their description, in order to be integrated into the Security Controller. A tentative document for the security plugin descriptor is depicted in Figure 33 in yaml format. Initial plugins to be considered will be based on Falco and Trivy frameworks.

```
apiVersion: nemo.eu/v1
kind: SecurityPlugin
metadata:
  name: {plugin name}
  description: {plugin description}
  version: {plugin version}
  url: {URL of the plugin homepage, if any}
  target-resource-class: {The resource class for which the plugin is targeted
to, if any}
plugin:
  type: {type of security plugin}
  timeout: {Time in ms before a connection gets characterized as timed out}
  scanned-resource: [{List of alert rules}]
  alert-rules: [{List of resources to be scanned}]
  workspace-type: {Location type of the plugin, may be "volume", or "remote"}
```

```
workspace-chroot: {The home directory of the plugin}
flavour: {Programming language of the plugin}
command: {The command to run to execute the scan}
```

Figure 33: Security plugin descriptor document example

Finally, the events identified, or security scan results generated by the running security plugins are communicated to interested NEMO components for being consumed. Specifically, they can be forwarded to the meta-Orchestrator to be considered during decision of workload placement, as well as to the Monetization and Consensus-based Accountability (MOCA) component to be considered for accounting purposes. Moreover, security events or scan results might be of interest to the Cybersecure Microservices' Digital Twin (CMDT) to be recorded in workloads' history, as well as to the PRESS & Policy Enforcement Framework (PPEF) for monitoring compliance to defined policies and SLAs.

5.2.3.3 Event-based Response

The event-based response component aims to apply automated responses to events triggered by user input or identified by other NEMO components, such as the PRESS & Policy Enforcement Framework (PPEF) or LCM's Security Controller. Indicatively, the Event-based Response supports the following activities:

- Workload responses: These include workload installation and automated updates, based on workload owners' preferences.
- Resource responses: These include automated onboarding or releasing of resources, based on resources' owners' preferences.
- Security responses: These refer to automated controls imposed on workload or resources' configuration and management as a result of security incidents identified. Indicatively, container images being assessed of high risk can be automatically blocked from being deployed. Moreover, workloads may be paused or shut down as a result of identified vulnerabilities at runtime.

This subcomponent will follow a modular architecture, similar to the Security Controller, in order to easily integrate additional responses.

5.2.3.4 LCM Visualization

The LCM Visualization is the interface between the end-users and the NEMO meta-OS ecosystem. It grants access to privileged users to manage their workloads and monitor their performance and security.

The internal architecture of a deployed solution is depicted in Figure 34.

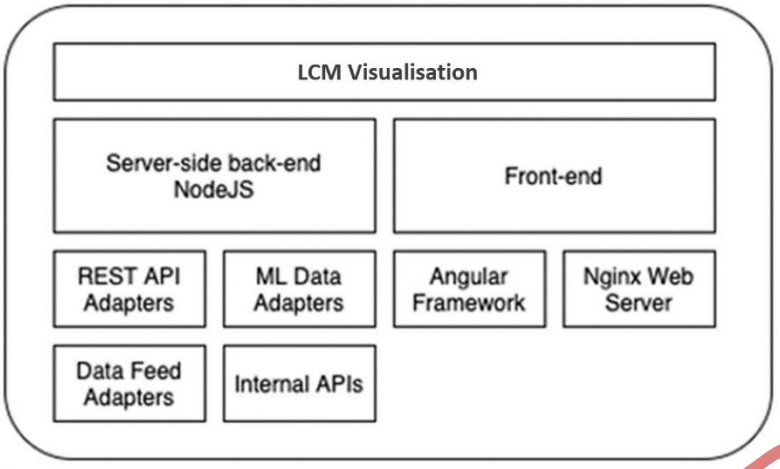


Figure 34: LCM Visualization architecture

The front-end container encompasses the main User Interface (UI) and the complementary web server that serves it. The interface's internal proxy configurations are handled inside this container. The back-end container contains the server-side middleware, which is responsible for handling all the necessary external communications in the deployed environment and the processing activities of the toolkit. Several adapters are integrated in this core component in order to support real-time data streams (e.g., FIWARE, Kafka, etc.), database connections (e.g., MongoDB, MySQL), different kinds of data formats (e.g., JSON files), REST APIs and more. The implementation of the required data queries and the needed pre-processing activities are taking place within this component.

Currently, the visualization of LCM includes screens to register a workload in NEMO meta-OS, to retrieve the lists of registered workloads and interface to deploy and manage workloads.

5.2.4 Interaction with other NEMO components

5.2.4.1 Identity management

The LCM consumes AAA services from the Identity Management component. Indicatively, it will perform entities' authentication and authorization to provide meta-OS consumers access to their workload data.

5.2.4.2 Access Control

The LCM will rely on the Access Control component for controlling access based on requesting user's or service's role.

5.2.4.3 Intent-based API

The LCM is the interface to the end-user, providing access to register a workload in NEMO meta-OS. Thus, any workload registration or deployment requests made through the LCM Visualization are forwarded to the API to be further processed. Moreover, the LCM receives request for executing workload deployment through the meta-Orchestrator. The LCM requires access to the Workload Registry in order to receive information about the workloads registered or running on NEMO.

5.2.4.4 meta-Orchestrator

The LCM interacts with the meta-Orchestrator both as an input and output. As an output, the LCM provides installation and deployment commands such as (install/uninstall, start/restart/stop). As an input, the meta-orchestrator provides feedback and updates regarding the status and progress of the workflow migration in order to track and monitor the migration process.

5.2.4.5 CMDT

As input to the CMDT, the LCM provides workload descriptors and workload lifecycle information, possibly including identified security events and scan results are forwarded to CMDT.

5.2.4.6 MOCA

LCM provides data related to the deployed service and its requirements, which might be interesting for accounting purposes.

5.2.4.7 PPEF

LCM provides the SLA definitions that concern the plugins that will be deployed in NEMO meta-OS.

5.2.5 Conclusion, Roadmap & Outlook

During the previous period we have defined the technologies to be used in the LCM and its subcomponents as well as basic interactions with other NEMO components and the role of LCM in NEMO high-level architecture. Next steps include the functional integration within NEMO ecosystem by implementing structured information exchange and deployment to the working environment.

5.3 Monetization and Consensus-based Accountability

5.3.1 Overview

The Monetization and Consensus-based Accountability (MOCA) component realizes the pre-commercial exploitation of the NEMO platform. This mechanism provides a trusted and secure mechanism for any type of user who offers or consumes resources to the NEMO platform. The main goal of the MOCA component is to provide an accounting service that will quantify reliably and fairly the account of resources that a provider can use over the NEMO continuum, based on the resources that have been offered and on the other hand the “cost” of a deployed service. Although the economic transactions are out of the NEMO context, MOCA implements an accountability mechanism based on “credits”. In the case of an infrastructure owner or a service provider that offers computational resources to the NEMO platform, MOCA provides some “credits” to be used for the deployment of services all over the NEMO continuum. The calculation of the “credits” is based on a sophisticated approach that takes under consideration multiple factors such as the amount of the offered computational resources, the resource demand in specific locations, and the type of offered infrastructure (i.e. edge cloud, 5G RAN, IoT devices, GPUs etc.). From the consumer’s perspective, each deployed service occupies specific computational and network resources that reflect a "cost" for the service owner. This approach enables the creation of new business models allowing volunteers and professionals to adopt the NEMO platform and offer hosting and migration services according to the resources as a service (RaaS) paradigm. MOCA offers a traceable way to build future business trade-offs between providers sharing bundles of computing, memory, storage resources and I/O resources for a short period of time based on DLT-based smart contracts. To achieve its goals, MOCA collaborates with the meta-Orchestrator, the CMDT and the monitoring framework.

5.3.2 Background

MOCA’s main technical functionalities, as defined in D2.1, include a) support of secure transactions resource allocation between stakeholders b) realization of business models based on smart contracts c) support of an accounting mechanism and d) sharing resource utilization information on the infrastructure layer. All these functionalities are based on interdomain transactions and data exchange between the NEMO platform and 3rd party infrastructure owners and service providers. This type of interaction has several challenges that should be taken under consideration during the design and implementation of any system like MOCA. These challenges are summarized as follows:

a) Trust establishment among entities from different domains can be challenging.
b) Data inconsistency in terms of formats, structures and semantics across domains that may lead to inconsistencies and errors in transactions.
c) Interdomain transaction security from threats like data breaches, hacking, and unauthorized access, as well as data integrity and confidentiality of transactions.
d) Scalability to address growing numbers of participants in interdomain transactions, so that the system is able to handle increased transaction volumes without sacrificing performance.
e) The complexity of the automated decision mechanism increases as the systems become smarter and more parameters are taken into account.

MOCA addresses the above challenges by embracing DLT technologies and provides a by-design secure, trusted, transparent, and decentralized approach for interdomain transactions in NEMO. In more detail, contrary to traditional interdomain transactions, where parties often encounter issues related to the lack of trust between entities, the need for intermediaries, and the potential for discrepancies in information, blockchain's decentralized and distributed ledger architecture mitigates these challenges. Furthermore, Blockchain provides a shared, transparent, and immutable ledger that is accessible to all participants, ensuring a single version of the truth. This reduces the risk of disputes and eliminates the need for a central authority to validate and reconcile transactions. Last but not least, blockchain use of cryptographic techniques ensures the security and integrity of data, making it resistant to tampering or unauthorized access. Finally, smart contracts can be part of a sophisticated decision mechanism, which can support self-executing contracts reducing the need for intermediaries and minimizing the risk of errors or fraud.

5.3.3 Architecture & Approach

The MOCA component is responsible for ensuring the integrity of a) transactions between NEMO and 3rd-party stakeholders and b) the accountability mechanism.

The component consists of the following sub-components.

5.3.3.1 IPFS

The Interplanetary File System (IPFS) is a P2P (peer-to-peer) distributed file system that can be used to store and access any type of data (e.g. files, JSON, jpeg etc.). Considering that the information that accompanies/describes a resource allocation transaction or an accounting action may be large (e.g. larger than just a few bytes) and thus not appropriate for being stored in the blockchain, we have decided to integrate an IPFS infrastructure. The different events along with the relevant information will be stored in a private IPFS network and we store in the blockchain a) the ID of the security event/policy and b) the corresponding link in the IPFS system. This way, the required time to store the information is reduced, compared to the case where we store information in the blockchain, and allows for larger data sizes. The events/policies become accessible via a link, which is stored in the blockchain. The IPFS guarantees that if any change happens to the source data, it will be detectable. The addition of the smart contracts component to the NEMO platform allows:

1. The verification that any transaction is handled by NEMO and
2. The guarantee that the details of the transactions are not tampered with (these details are accessed through the link stored in the blockchain).

5.3.3.2 DApps

The Decentralized applications (DApps) sub-component consists of the Smart Contracts that contain the logic for storing the original source of the data (IPFS link) and retrieving it. The DApps component is deployed in a private blockchain network, namely Quorum. This private blockchain network solution uses the IBFT (Istanbul Byzantine Fault Tolerance) consensus mechanism. This mechanism is one of the best regarding performance and transaction speed, therefore making the overall implementation very fast.

5.3.3.3 Event Server

The Relay Server subscribes to the RabbitMQ of the Central Repository and is notified of every new event/policy added. The Server then composes the information in a file to store in the IPFS network.

The Relay Server also exposes REST API endpoints connecting to the DApps, for the components to access the link for the details of the events/policies in the IPFS. This subcomponent is also responsible for keeping track of all the events that are emitted from the smart contracts of the DApps. Thus, instead of querying the blockchain directly to ensure that an event has been triggered (e.g., when a new IPFS link is stored), the log the Event Server creates a log to keep track of the emitted events that can be used to determine whether a call to the DApps was successful and check briefly its most basic details (e.g. the ID of the event). It should be noted that all the above sub-components are designed to work as Kubernetes deployments. The Quorum blockchain platform is available as a Helm chart that will also be deployed in a Kubernetes cluster.

5.3.3.4 Smart Contact Component

The Smart Contracts component and the relevant blockchain network will be deployed in the context of NEMO. However, the blockchain network which is necessary for the NEMO smart contracts component operation will in general be a private blockchain network as is the case for numerous applications/solutions in the market. This means that one or multiple nodes could be deployed within NEMO, another (or another set) on external entities. The NEMO clients do not need to contribute to the maintenance or deployment of the blockchain network. This method ensures that:

1. A decentralized solution is offered since the nodes can be hosted on different premises.
2. The data stored are secure since they cannot be deleted, and any tampering attempts are detectable and
3. There is no single point of failure, since all the nodes hold replicas of the stored data, and one failed node cannot compromise the entire network.

It is worth stressing that: i) the fact that NEMO relies on a private blockchain network does not decrease the value of the decentralized solution and ii) protecting NEMO operations through the integration of blockchain techniques increases the security of the NEMO platform; this does not mean that this is 100% secure as such a security level does not exist; it means that a higher security level is reached and this should be considered keeping in mind the value of the protected data.

5.3.4 Interaction with other NEMO components

MOCA communicates directly, either via RabbitMQ or REST APIs, with the core components of the NEMO platform (to receive this information) and stores it in the blockchain. Figure 35 depicts a high-level architecture of the MOCA component as well as its relations with the NEMO platform.

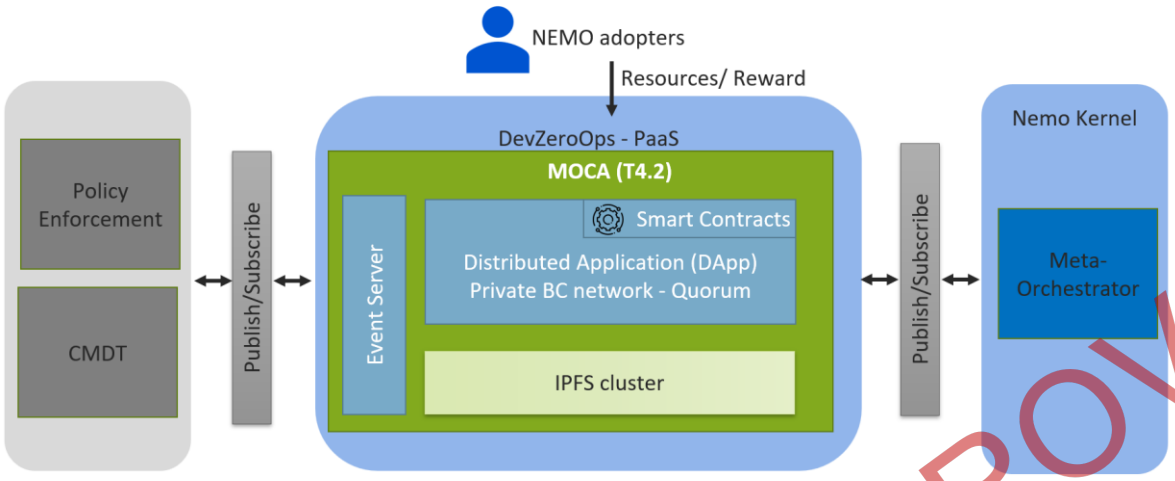


Figure 35: The MOCA component and interactions

5.3.5 Conclusion, Roadmap & Outlook

MOCA provides a significant advantage to the NEMO platform by establishing a distributed secure and trusted framework that accommodates the transactions between NEMO and external stakeholders. Currently, the first version of MOCA has been developed and it's ready for the first round of testing in the stage environment, The next steps involve full integration with the rest of the NEMO platform and the evaluation through the NEMO trials. The evaluation results will drive the second round of development and the release of the final version.

5.4 Intent-based SDK/API

5.4.1 Overview

NEMO will rely on its Intent-based Application Programming Interface (API) and Software Development Kit (SDK) for maximizing the adoption potential by third party entities, including both the meta-OS consumers and meta-OS partners, as well as external applications and (micro-)services. It is aimed to expose NEMO lower-level functionality to the outside world in an easily accessible format, minimizing the effort needed on their side to adapt applications, services and plugins to NEMO-capable ones, but also introducing minimal distraction compared to common practice for proficient (K8s) cluster users.

5.4.2 Background

Kubernetes Admission Controllers

Admission controllers are plugins that can be configured to intercept and potentially modify admission requests to the Kubernetes API server. They are a crucial part of the Kubernetes control plane, helping to enforce cluster-wide policies and ensure that workloads adhere to defined rules before they are admitted to the cluster. Admission controllers operate as watchdogs which are responsible for enforcing custom policies prior to executing requests or persisting objects in etc.

Admission Controllers can be of the following types:

- Mutating Admission Controllers: These controllers may modify (mutate) the content of objects before they are persisted. Examples include the *MutatingAdmissionWebhook*.

- Validating Admission Controllers: These controllers can deny admission based on certain criteria, but they cannot modify the object. Examples include the *ValidatingAdmissionWebhook*.

It should be noted that some Admission Controllers could play the role of both. Moreover, some Admission Controllers are enabled by default. These controllers are responsible for basic validation and defaulting. Examples include *NamespaceLifecycle*, which ensures certain namespace-related policies are adhered to.

Following the admission controller types, the admission control process is executed in 2 phases, namely the mutating and the validating phase. As depicted in Figure 36, authenticated requests to the Kubernetes API pass through the mutating phase, which may augment pods with semantic information useful for governance and configuration management. Then, schema validation for the request object is executed and then the request passes through a set of checks enabled by the validating controllers. In this phase, some requests may not be admitted in the requested cluster or namespace, based on the nature of checks/policies and the implemented admission controller logic.

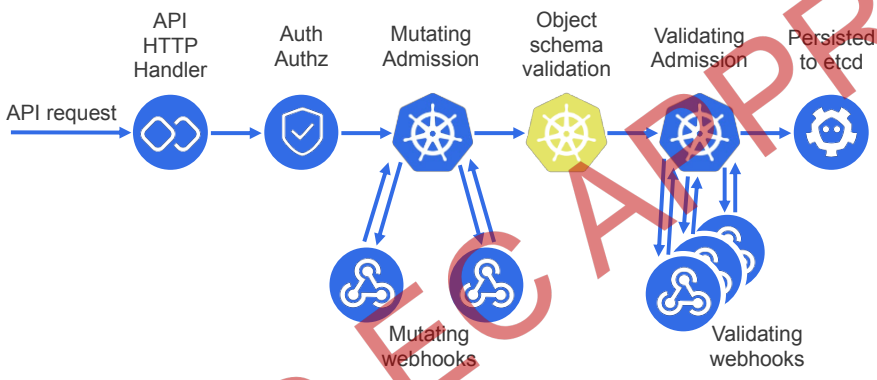


Figure 36: Admission control execution process

Kubernetes provides a list of diverse plugins which are used as admission controllers[21], while custom admission controllers can be created via the *AdmissionReview* API. Some common examples of the available ones include:

- PodSecurityPolicy (PSP): Enforces security policies for pods.
- ResourceQuota: Enforces restrictions on resource usage in a namespace.
- LimitRanger: Provides default values and limits for resources in a namespace.
- NamespaceLifecycle: Enforces policies related to namespace lifecycle.

NEMO may leverage the concept of the Admission Controllers in both validating workloads before they are admitted for registration or deployment in the NEMO meta-OS, as well as for enforcing defined NEMO- or cluster-wide rules before a workload request execution.

Checkov

Checkov [39] is an open-source static code analysis tool for scanning Infrastructure-as-Code (IaC) for misconfigurations that may expose system vulnerabilities and lead to security compliance issues. Checkov includes more than 750 pre-defined policies to check for common misconfiguration issues. Additionally, it supports the creation and contribution of custom policies.

Checkov supports a series of IaC file types, including (but not limited to) Terraform [49], Kubernetes [50], Helm Charts [51], Docker [18], etc. Moreover, on top of the detection of common

[21] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

misconfigurations, Checkov scans for compliance with common industry standards such as the Center for Internet Security (CIS) [52] and Amazon Web Services (AWS) Foundation Benchmark [53].

In the context of NEMO, regarding the IaC file types that need to be supported, Checkov can examine Dockerfiles for 37 built-in policies, and Kubernetes resources for more than 900 built-in policies. Since the installation of a Helm chart eventually leads to the creation of multiple Kubernetes resources, the later applies to the Helm Chart IaC file type. For the sake of completeness, an indicative list of the built-in policies that are examined can be found in Table 5.

Table 5: Indicative Checkov policies

ID	IaC	Entity	Policy
CKV_DOCKER_1	Dockerfile	EXPOSE	Ensure port 22 is not exposed
CKV_DOCKER_7	Dockerfile	FROM	Ensure the base image uses a non-latest version tag
CKV_DOCKER_8	Dockerfile	USER	Ensure the last USER is not root
CKV2_DOCKER_17	Dockerfile	RUN	Ensure that “chpasswd” is not used to set or remove passwords
CKV_K8S_20	Kubernetes	Deployment	Containers should not run with allowPrivilegeEscalation
CKV_K8S_29	Kubernetes	Pod	Apply security context to your pods and containers
CKV_K8S_30	Kubernetes	Pod	Apply security context to your containers
CKV_K8S_35	Kubernetes	Pod	Prefer using secrets as files over secrets as environment variables
CKV_K8S_37	Kubernetes	StatefulSet	Minimize the admission of containers with capabilities assigned

Checkov is going to be utilized in the context of NEMO and the Intent-based SDK / API to suggest the policies that need to be enforced before the registration of a service. After an in-depth examination of the available policies, a collection of the ones that are considered essential will be created. A wrapper around Checkov, would then enforce these essential policies. Furthermore, custom policies can be created if / when considered necessary.

Grype & Syft

Grype [54] is an open-source vulnerability scanner for container images and filesystems. When it runs, a local database of vulnerabilities gathered from a variety of publicly available vulnerability data sources, including (but not limited to) the Alpine Linux SecDB [55], RedHat RHSAs [56], Ubuntu Linux Security [57], and others. Grype is able to scan source code and Docker images. As a side-note, the tool is expected to detect more vulnerabilities in the container images, given that they include packages that are not necessarily included in the source code.

Syft [58] is another open-source command-line interface (CLI) tool and Go library for generating a Software Bill of Materials (SBOM) for containers and filesystems. An SBOM essentially is a list of all the libraries, code packages and other third-party components used to create a particular software application. An SBOM also includes each component’s license type, version and patch status, and the dependencies between the said components in the software supply chain [59]. The importance of an SBOM is underlined by the fact that the software supply chain has become a leading source of software vulnerabilities and breaches, since code reusability is constantly increasing.

Both Grype and Syft are developed by Anchore [60] and integrate seamlessly. Their combination is expected to prove beneficial in enhancing security of the deployed services in the NEMO context.

5.4.3 Architecture & Approach

The Intent-based API implements the logic for exposing the NEMO functionalities to internal or third-party entities. The API will automatically discover and expose resources of registered and deployed workloads in NEMO, enforcing privileged control to their access. Moreover, NEMO API supports the registration and deployment of workloads into the NEMO resources.

Regarding the workload registration, it implements the following workflow. First, NEMO consumers, desiring to deploy their application or service into NEMO, have to register their workloads in the NEMO meta-OS. This is realized by posting a registration request of the workload in NEMO, which includes the *NEMO workload descriptor*. This descriptor includes information useful for discovering the relevant workload from external sources, such as the source code, a container image, a Helm chart, etc., as well as useful information for the workload per se, such as the workload name, the compatible NEMO API version, the workload implementation version, the type of technology the interface is implemented, compatibility with VIM versions, etc. It also includes the definition of intents for the execution of the workload, such as requirements for network, compute and storage resources, for secure execution (e.g., requirement to be executed through containers or unikernels), as well as energy efficiency requirements, such as executing the workload in green-powered servers, etc. Moreover, the NEMO workload descriptor may include information for multi-cluster execution or execution in specific clusters. In addition, policy enforcement rules may be included in this descriptor. Last, but not least, the NEMO consumer may declare resources of their workload desired to be exposed and under which access control criteria. In order to complete the registration into NEMO, the workload must successfully pass the NEMO validation checks. These include compatibility checks between the workload versions and requirements and the NEMO clusters, workload assessment regarding discoverability of said resources, as well as security tests regarding the workload sources. Once the validation is successful, the workload descriptor is augmented with NEMO annotations and the workload is added in the NEMO Registry. The relevant subroles of the Meta-OS Provider (see D1.2) should then validate this registration and a token is created, which allows for deployment of the workload in the NEMO meta-OS. This token is provided to the NEMO Consumer. This workflow is illustrated in Figure 37.

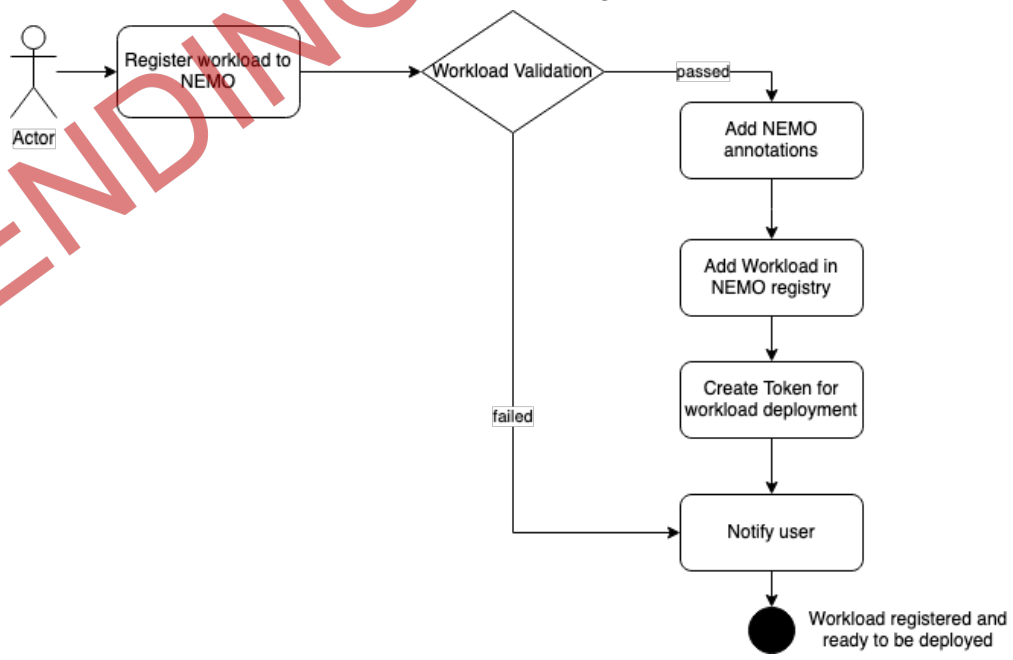


Figure 37: The workload registration workflow

Once the Workload is registered in NEMO, the NEMO Consumer may submit a request for its deployment through the Intent-based API. Before allowing the request to be forwarded for execution, validation and verification tests must have been successfully passed. These include compatibility, performance and security tests, as well. After successful verification and validation, the request for the workload deployment is communicated to the *Applications' & Plugins Lifecycle Manager*, who coordinates the deployment in NEMO via the *NEMO Meta-Orchestrator*, considering both the specified intents and the NEMO resources' capabilities and policies. Upon successful deployment, the API caters for automated workload provisioning. This is realized by providing access to workload resources, following defined Role-Based Access Control (RBAC) rules, ensuring that Ingress and Egress traffic are correctly set up, as well as workload Lifecyle Management is initiated for the deployed workload. This workflow is presented in Figure 38.

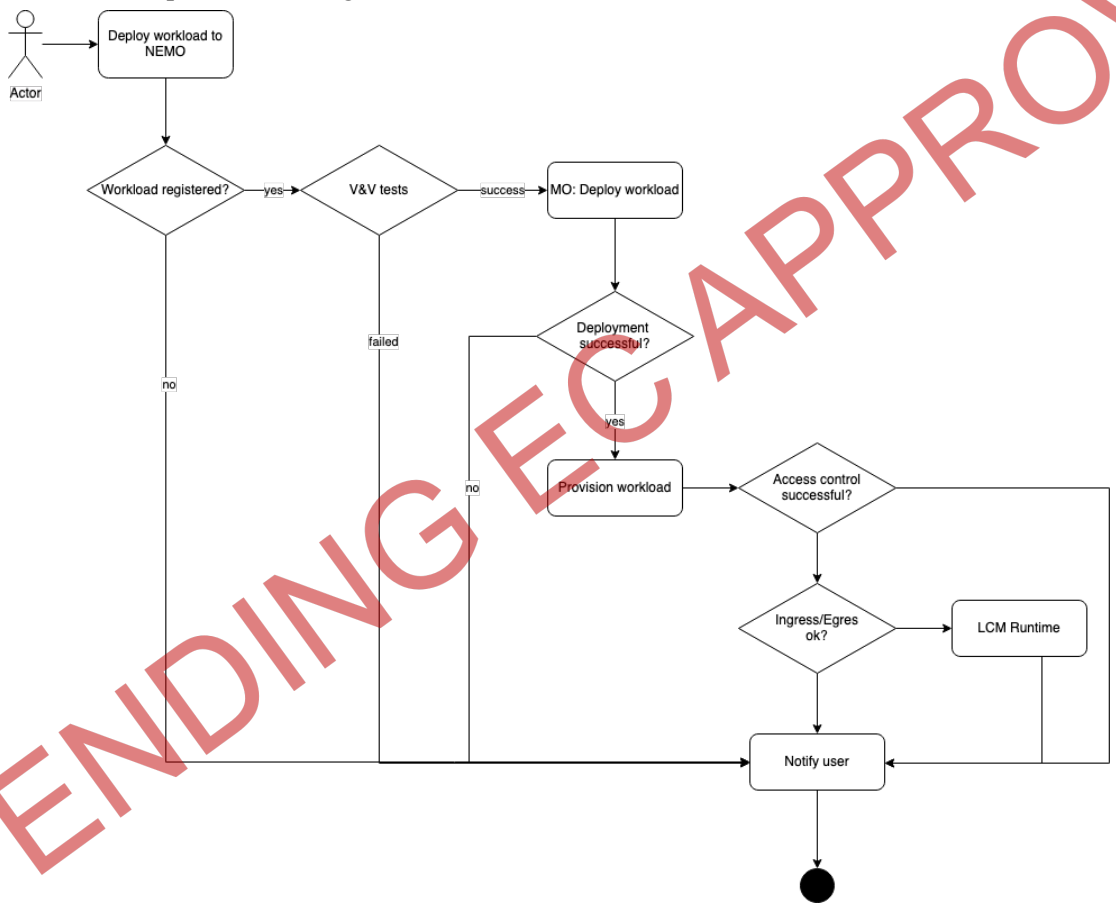


Figure 38: The workload deployment workflow

The high-level architecture of the NEMO Intent-based API is depicted in Figure 39.

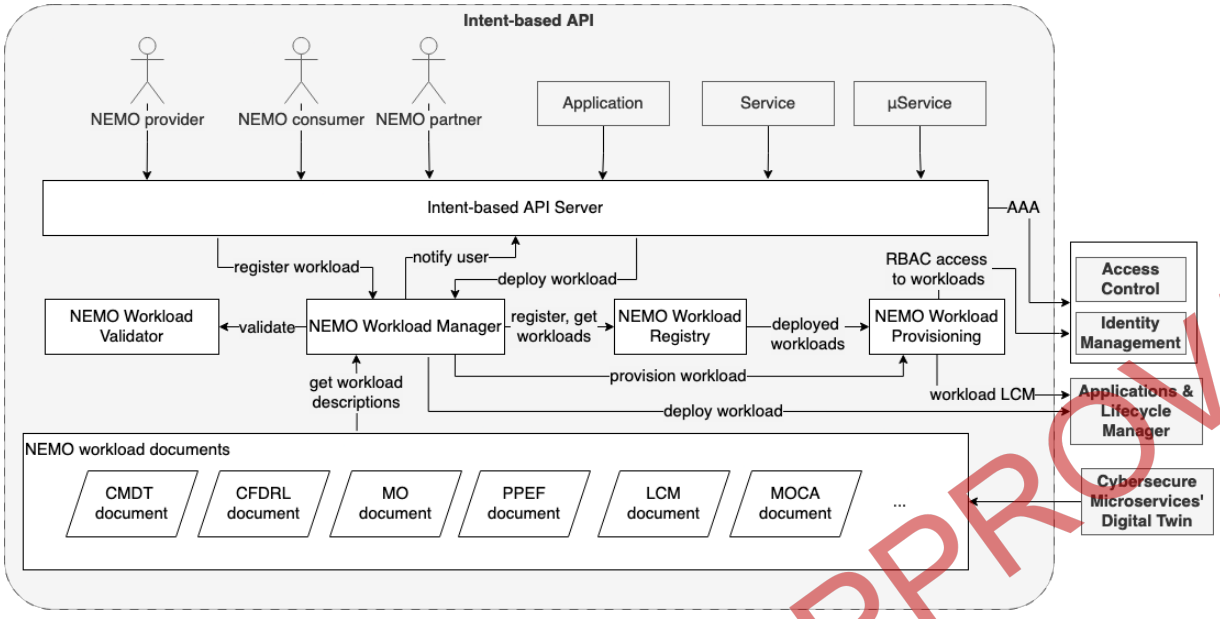


Figure 39: Intent-based API architecture

5.4.3.1 Intent-based API Server

The *Intent-based API Server* is the application that provides NEMO functionality through RESTful interfaces. It allows accessing or managing information and configuration on NEMO workloads and resources, respecting the defined RBAC rules. This API Server communicates with the *NEMO Workload Manager*, in order to register and instantiate workloads, i.e., components, services, applications or plugins, into NEMO.

5.4.3.2 NEMO Workload Manager

The *NEMO Workload Manager* manages the processes for registration/deregistration of workloads, including NEMO annotations, workflow execution, logging and notification of external entities. It also coordinates the workflow for the workload deployment through the LCM and workload provisioning.

5.4.3.3 NEMO Workload Validator

When deploying a workload within the NEMO meta-OS, there are several considerations that need to be addressed. The NEMO Workload Validator aims to apply validations to ensure that the workload runs effectively and efficiently across the distributed environment. Some key aspects to validate include the following.

Compatibility with underlying VIM versions: Ensure that the workload is compatible with the versions of VIM (e.g. Kubernetes) running in each cluster. Different clusters may be running different versions or types of VIMs, and the workload should be tested and validated against these versions.

Resource Requirements: Validate that the workload's intents as defined by its owner, related to the required resources (CPU, memory, storage) are appropriate for the clusters where it might be deployed. Consider variations in cluster sizes and resource availability.

Networking Requirements: Verify that the workload can handle network communication across clusters. This includes validating that the necessary ports are open, and network policies are configured appropriately. Ingress and egress configurations should be tested across clusters and VIMs.

Data Management: If the application relies on persistent storage, ensure that storage solutions are compatible with the multi-cluster environment. This includes validating storage classes, access modes, and persistent volume configurations.

Secrets Management: Validate that secrets used by the workload are managed securely and consistently within the meta-OS.

Security Policies: Validate that the workload adheres to security policies defined for each cluster or VIM.

Integration with VIM Services: Ensure that the workload integrates seamlessly with cluster services such as authentication, authorization, and any other platform-level services.

Compliance with VIM Policies: Validate that the application complies with any specific cluster policies or governance requirements in each cluster.

By thoroughly testing and validating these aspects, candidate workloads are likely to be executed and operate successfully in the multi-cluster meta-OS setting.

The NEMO Workload Validator will leverage the K8s Admission Controllers, as well as Checkov, Grype and Syft in order to implement validation checks, considering the above-mentioned validation aspects.

5.4.3.4 NEMO Workload Registry

The Workload Registry keeps the list of workloads registered in the NEMO meta-OS. It keeps updated with user requests for registration and deregistration of workloads. The Meta-OS Service and Cluster Deployment Manager subrole of the Meta-OS Provider (see D1.2 for definition of the meta-OS user roles) will be able to register the requested workloads into the registry. This will be the initial entry of the workload in the NEMO system.

5.4.3.5 NEMO Workload Provisioning

This component ensures the deployment of workloads and access to relevant services by authorized/eligible entities and roles. This subcomponent will communicate with the Identity Management module to ensure that RBAC is applied for the delivery of the workloads to be provisioned. The component provides automated workload provisioning, by discovering the workload resources to be provisioned in the NEMO workload documents and ensuring that relevant endpoints are made available, imposing defined access controls and policies. Before the workload resources get provisioned, ingress and egress traffic get configured and LCM processes get initiated to ensure workload scanning at runtime.

5.4.3.6 NEMO Workload Documents

These include the *NEMO Workload Descriptors* (e.g., yaml files), which comprise common service description, augmented with NEMO annotations. The Workload Documents refer to workloads already registered in the Registry and are originally provided by the Cybersecure Microservices' Digital Twin (CMDT).

5.4.4 Interaction with other NEMO components

The Intent based API interacts with the following components:

- Identity management.
- Access Control.
- Applications & Lifecycle Manager.

Details on the interaction can be found below.

5.4.4.1 Identity management

The Intent based API interacts with the Identity Management Component for supporting authentication and authorization services for accessing the endpoints of the API Server. Moreover, the Identity Management component will be used to create tokens allowing deployment of registered workloads.

5.4.4.2 Access Control

The Intent based API interacts with the NEMO Access Control for enforcing access control policies to the API Server endpoints, following RBAC rules. Moreover, the API relies on the Access Control component for enforcing access policies during workload provisioning.

5.4.4.3 Application & Lifecycle Manager

The Intent based API interacts with the Application & Lifecycle Manager in order to execute a workload deployment request, after successful validation and verification tests. Moreover, before applying the workload provisioning, the API interacts with the LCM in order to initiate the LCM processes during workload runtime.

5.4.4.4 Cybersecure Microservices' Digital Twin

The Intent based API interacts with the Cybersecure Microservices' Digital Twin for communicating the workload information provided by the NEMO consumer, in order to create, maintain and update the NEMO workload descriptors. Moreover, the API will receive latest updates on such descriptors and, accordingly, updated *NEMO workload documents*, for the deployment of the workloads on NEMO.

5.4.5 Conclusion, Roadmap & Outlook

The Intent-based API plays a significant role in the NEMO meta-OS both functionally and in terms of exploitation. It exposes endpoints to external entities for interacting with NEMO and consuming NEMO functionality. Moreover, it plays a significant role in workload registration and deployment. In addition, it significantly contributes to meta-OS openness towards third parties for further development on top of NEMO in a developer-friendly way.

In its first version, the API focuses on the automated discovery and provisioning of workload resources. Subsequent steps involve the execution of registration and deployment requests, based on a NEMO-wide workload document definition.

6 Conclusions

In this document, structured and rigorous planning is pursued for the NEMO software lifecycle, aiming to maximize the level of automation and thus minimize the time, effort and risks during integration of software modules developed by different teams within the Consortium. Besides development and integration guidelines, as adoption of software practices and tools, the document presents the CI/CD processes of the project, following DevSecOps. Indeed, security is considered in several aspects and stages in the pipeline, which similarly includes different types and points of validation. These practices are to be followed during the project lifetime, which, according to the integration and V&V plan, is split in three phases, escalating NEMO functionality in three releases. The preliminary release is presented in this document, mainly focusing on logical integration and interactions' specification. It also includes an early prototype of the NEMO components. Last, but not least, the document presents the advancements towards technical support of third-party integration and exploitation, through four components in the NEMO *Service Management* layer.

The next integrated release of the NEMO framework, accompanied by new features of the Service Management layer, is planned for the last quarter of 2024, which coincides with the conclusion of the second project phase and will be reported in D4.2.

7 References

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

[2] Argo, "Argo Workflows," 2023. [Online]. Available: https://argoproj.github.io/argo-workflows/.

[3] Apache, "Airflow," 2023. [Online]. Available: https://airflow.apache.org/.

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

[5] 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/.

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

[7] Amazon, "What is Pub/Sub Messaging?," 2019. [Online]. Available: https://aws.amazon.com/pub-sub-messaging/.

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

[9] 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/.

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

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

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

[13] Flux, "Flux multi-tenancy," 2023. [Online]. Available: https://fluxcd.io/flux/installation/configuration/multitenancy/.

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

[15] AsynvAPI Initiative, "Building the future of Event-Driven Architectures (EDA)," [Online]. Available: https://www.asyncapi.com. [Accessed 2024].

[16] Swagger, "Open API Specification," [Online]. Available: https://swagger.io/resources/open-api/. [Accessed 2024].

[17] Swagger, "Swagger Editor," [Online]. Available: https://swagger.io/tools/swagger-editor/download/. [Accessed 2024].

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

[19] GitLab, "The Role of AI in DevOps," 2023. [Online]. Available: https://about.gitlab.com/topics/devops/the-role-of-ai-in-devops/.

[20] T. E. P. a. t. C. o. t. E. Union, "Directive on measures for a high common level of cybersecurity across the Union (NIS2 Directive)," 2022. [Online]. Available: https://eur-lex.europa.eu/eli/dir/2022/2555.

[21] CISA, "Zero Trust Maturity Model," 2023. [Online]. Available: https://www.cisa.gov/sites/default/files/2023-04/zero_trust_maturity_model_v2_508.pdf.

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

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

[24] 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.

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

[26] Istio, "The Istio service mesh," 2023. [Online]. Available: https://istio.io/latest/about/service-mesh/.

[27] HashiCorp, "Consul - Standardize service networking," 2023. [Online]. Available: https://www.hashicorp.com/products/consul.

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

[29] 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.

[30] KongHQ, "Kong Gateway," 2023. [Online]. Available: https://konghq.com/products/kong-gateway.

[31] Envoy, "Envoy Gateway," 2023. [Online]. Available: https://gateway.envoyproxy.io.

[32] 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/.

[33] The Linux Foundation, "Cloud Native Computing Foundation," [Online]. Available: https://www.cncf.io/. [Accessed 12 2023].

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

[35] Selenium, "Selenium," 2023. [Online]. Available: https://www.selenium.dev/.

[36] Robot Framework Foundation, "Robot Framework," [Online]. Available: https://robotframework.org. [Accessed 2024].

[37] Apache, "JMeter," [Online]. Available: https://jmeter.apache.org. [Accessed 2024].

[38] BlazeMeter, "Taurus," [Online]. Available: https://gettaurus.org. [Accessed 2024].

[39] Prisma Cloud, "Checkov," [Online]. Available: https://www.checkov.io/. [Accessed 12 2023].

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

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

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

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

[44] hubblo-org, "Scaphandre," 2023. [Online]. Available: https://github.com/hubblo-org/scaphandre.

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

[46] Flower Labs GmbH, "Flower," open source, [Online]. Available: https://flower.dev. [Accessed 2023].

[47] The Linux Foundation, "Falco," [Online]. Available: https://falco.org/. [Accessed 12 2023].

[48] "trivy-operator," [Online]. Available: https://github.com/aquasecurity/trivy-operator. [Accessed 12 2023].

[49] HashiCorp, "Terraform by HashiCorp," [Online]. Available: https://www.terraform.io/. [Accessed 12 2023].

[50] The Linux Foundation, "Kubernetes," [Online]. Available: https://kubernetes.io/. [Accessed 12 2023].

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

[52] Center for Internet Security, "CIS Center for Internet Security," [Online]. Available: https://www.cisecurity.org/. [Accessed 12 2023].

[53] Amazon Web Services, Inc., "Center for Internet Security (CIS) AWS Foundations Benchmark v1.2.0 and v1.4.0 - AWS Security Hub," [Online]. Available: https://docs.aws.amazon.com/securityhub/latest/userguide/cis-aws-foundations-benchmark.html. [Accessed 12 2023].

[54] Anchore, "anchore/grype," [Online]. Available: https://github.com/anchore/grype. [Accessed 12 2023].

[55] "Alpine Linux SecDB," [Online]. Available: https://secdb.alpinelinux.org/. [Accessed 12 2023].

[56] "RedHat RHSAs," [Online]. Available: https://www.redhat.com/security/data/oval/. [Accessed 12 2023].

[57] Canonical, "Ubuntu Security Team," [Online]. Available: https://people.canonical.com/~ubuntu-security/. [Accessed 12 2023].

[58] Anchore, "anchore/syft," [Online]. Available: https://github.com/anchore/syft/. [Accessed 12 2023].

[59] Circle Internet Services, Inc., "Software bill of materials: What it is and why you need one," [Online]. Available: https://circleci.com/blog/what-is-a-software-bill-of-materials/. [Accessed 12 2023].

[60] Anchore, "Open Source Container Security with Syft and Grype," [Online]. Available: https://anchore.com/opensource/. [Accessed 12 2023].