nephele

A Lightweight Software Stack and Synergetic Meta-Orchestration Framework

for the Next Generation Compute Continuum

D5.1 First Release of NEPHELE Platform, Dashboard and DevOps Environment Setup

---

Document Identification			
Status	Final	**Due Date**	29/02/2024
Version	1.0	**Submission Date**	29/02/2024

Related WP	WP5	**Document Reference**	D5.1
Related Deliverable(s)	D3.1, D4.1	**Dissemination Level (*)**	PU
Lead Participant	SMILE	**Lead Author**	Jonathan Rivalan
Contributors	SMILE, UOM, ECL, ATOS, NTUA	**Reviewers**	Marco Jahn (ECLIPSE)
			Dimitris Spatharakis (NTUA)

Keywords:
Telecom infrastructures, networking, Virtual Object, W3C WoT, OMA LwM2M, CI/CD, DevOps

Document Information

List of Contributors	
Name	Partner
Jonathan Rivalan	SMILE
Hai Long Ngo	SMILE
Saifuddin Mohammad	SMILE
Georgios Papathanail	UOM
Panagiotis Papadimitriou	UOM
Lefteris Mamatas	UOM
Ilias Sakellariou	UOM
Guillermo Gomez	ATOS
Anastasios Zafeiropoulos	NTUA
Dimitrios Spatharakis	NTUA
Nikos Filinis	NTUA
Eleni Fotopoulou	NTUA
Ioannis Dimolitsas	NTUA
Marco Jahn	ECLIPSE
Chiara Lombardo	CNIT

Document History			
Version	Date	Change editors	Changes
0.1	20/11/2023	SMILE (JR)	Document initialisation, table of contents, executive summary
0.2	16/01/2024	SMILE (SM, LN)	Added tables content
0.3	23/01/2024	UOM, SMILE	Refined content and adjusted status
0.4	30/01/2024	SMILE (SM, LN)	Completed tables lists, updated repositories status
0.5	6/02/2024	SMILE (SM, LN)	Additional status updates
0.6	14/02/2024	SMILE	Additional status updates
0.7	20/02/2024	NTUA (DS), SMILE (JR)	Added global infrastructure details and main components description Added CI/CD runners platforms
0.8	21/02/2024	NTUA SMILE (JR, SM, LN)	Added Dashboards mock-up appendix Added Conclusion and infrastructure details Updated repositories list
0.9	27/02/2024	ECL (MJ), NTUA (DS), SMILE (LN, JR)	Review. Accepted with minor comments. Final edits
1.0	28/02/2024	NTUA (AZ, SP)	Final version

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Jonathan Rivalan (SMILE)	28/02/2024
Quality manager	Anastasios Zafeiropoulos (NTUA)	28/02/2024
Project Coordinator	Symeon Papavassiliou (NTUA)	28/02/2024

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
API	Application Programming Interface
CD	Continuous Deployment
CI	Continuous Integration
CI/CD	Continuous Integration/ Continuous Development
CIDR	Classless Inter-Domain Routing
CoAP	Constrained Application Protocol
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
cVO	Composite Virtual Object
Dx.y	Deliverable number y belonging to WP x
EC	European Commission
GCL	Gate Control List
GUI	Graphical User Interface
HDA	Hyper Distributed Application
HDAR	Hyper Distributed Application Repository
HTTP	Hypertext Transfer Protocol
IM	Information Model
K8s	Kubernetes
LwM2M	Lightweight M2M
M2M	Machine-to-machine
MQTT	Message Queue Telemetry Transport
NBI	North-Bound Interface
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualization
NFVCL	NFV Convergence Layer
NFVO	NFV Orchestrator
NSI	Network Service Instance
OMA	Open Mobile Alliance
ONOS	Open Network Operating System
PoC	Proof of Concept
REST	REpresentational State Transfer
TSN	Time-Sensitive Networking
SDN	Software-Defined Networking
SD-WAN	Software-Defined Wide Area Network
SD-WSN	Software-Defined Wireless Sensor Networking

SMO	Synergetic Meta Orchestrator
VO	Virtual Object
W3C	World Wide Web Consortium
WoT	Web of Things
WP	Work Package

Executive Summary

This document depicts the integration efforts and results towards setting up the NEPHELE components as an industrialized platform, which deployment, maintenance, and operational management is made easily understandable, achievable and reproducible.

This document aims to present in a single place multiple aspects of the project integration effort:

- the current functional state at the two milestones [M18, M36] of the NEPHELE prototype components integration and their resulting APIs
- source code availability and source code management principles
- the associated documentation elements supporting integration within, and usage for, the resulting platform
- Open-Source quality requirements
- DevOps guidelines

As such, the current document provides knowledge about the collaboration within the partners and WPs towards a working prototype, and the open-source access to its components, documentation and tests. Infrastructure-wise, D5.1 provides the full details about made-available resources, from external source code managers to private CI/CD platforms and project-level evaluation infrastructures and testbeds.

As of the M18 milestone, the NEPHELE integration is comprised of three main results: 1. working prototypes from all WPs made available publicly through the Eclipse Research Labs Gitlab, 2. CI/CD integration effort enabling automated builds and test runs, 3. DevOps bootstrap documentation including code examples and guidelines to support microservices-oriented developments within the project.

Aside from the direct results, prototypes were audited as standalone solutions and their integration within a single tenant was studied and discussed. A contribution is designed in the form of a nomenclature, detailing components support for CI/CD, and microservices deployment.

From the first prototypes published in M18, a complete platform prototype seems achievable in the coming months. The next internal iteration is set on M24.

In conclusion, NEPHELE's integration effort is following the path described initially in the technical annex. Open-Source results are made available in a DevOps approach, and platform setup is ongoing.

1 Introduction

1.1 Purpose of the document

This document is aimed at reporting the integration status of NEPHELE components as single standalone components and as an overall integrated platform. It corresponds to the deliverable D5.1. Two versions are expected, this initial version at the milestone M18, and another version at the milestone M36.

The current version M18 pursues the presentation of the first round of the integration effort, the overview of public repositories and quality-oriented requisites (docker files, tests, licences), made available APIs and their respective formats, proposed integration tools, and execution platforms. It also depicts functional blocks at a global architecture level and emphasises over-testing principles.

1.2 Relation to other project work

D.5.1 is meant to report the results in integrating the NEPHELE platform, at various stages of the developments:

- The first stage includes a first working prototype with part of the WP3-WP4 mechanisms integrated and is expected by M18
- the second stage includes a full working prototype with all the mechanisms integrated and is expected by M24
- a third stage release includes all the updates based on feedback received from the project use cases providers and is expected by M24

1.3 Structure of the document

This document is structured in 6 major chapters and an appendix.

Chapter 1 (the current chapter) **and Chapter 6** present the document outline, plan, and outcomes. Respectively as an introduction chapter and a conclusion one.

Chapter 2 details the integration status. It first provides an overview of project results from the other WPs and then depicts their integration effort as a single platform.

Chapter 3 details the overall integration support. It depicts the various infrastructures and solutions provided within the project along with their capabilities (source code management, compute support, evaluation environments).

Chapter 4 details the standard architecture components. It details functional components as well as their relations to the platform features and usage.

Chapter 5 details the documentation produced. Documentation is comprised of integration guidelines for the microservices and open-source results.

2 Single components integration status

This chapter presents the NEPHELE platform integration status under 2 perspectives. First, an overview of results as standalone components, second the current state of the overall integration.

2.1 Single components status [M18]

2.1.1 Reported progress

In this section, we introduce the main repositories associated with components of the NEPHELE project, which are hosted on the Eclipse Research Labs GitLab[1]. Specifically, the discussion encompasses concise descriptions, integration maturity assessments, correlations with the technical annex, authorship attributions, and pertinent annotations for each repository. The repositories under consideration are VO-WOT, wot-py-2, nephele-HDAR, VO-LwM2M, VO-Security, VO-TSN, and VO-SDN. This comprehensive overview aims to offer insights into the foundational facets of these repositories within the broader context of the NEPHELE project's framework. Table 1 synthesises the results as reported by the different responsible partners involved in the development and maintenance of these repositories.

Table 1: NEPHELE project's repositories summarization

Repository name	**Description**	**Integration Maturity (Poc/Proto/Final)**	**Relation to technical annex [WPx-Tx-Dz]**	**Partners involved**	**Notes**
VO-WoT	This repository is a fork of the original WoTPy[2] repository. VO-WoT is an experimental implementation of a W3C WoT Runtime[3] and the W3C WoT Scripting API[4] in Python, inspired by the exploratory implementations located on the_Thingweb GitHub page[5].	In development - PoC	WP3 - T3.1 - D3.1	NTUA	Device abstraction based on W3C-Web of Things
wot-py-2	This repository is an updated version of WoTPy. WoTPy is an experimental asynchronous implementation of a W3C Web of Things runtime	In development - Prototype	WP3 - T3.1 - D3.1	Siemens	
nephele-HDAR	This multi-project repository hosts the commitments of T4.1. Dev environment and HDAR including Information Model (IM)of artifacts.	In development - Prototype, Beta version	WP4 - T4.1 - D4.1	ATOS	

[1] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project

[2] https://github.com/agmangas/wot-py

[3] https://github.com/w3c/wot-architecture/blob/main/proposals/terminology.md#wot-runtime

[4] https://github.com/w3c/wot-architecture/blob/main/proposals/terminology.md#scripting-api

[5] https://github.com/thingweb

VO-LwM2M	Device abstraction based on OMA-LwM2M standard. The environment surrounding a Virtual Object (VO) is multifaceted, encompassing various components that facilitate its operation and interaction within the computing continuum. This section delves into these primary components, shedding light on their roles and functionalities.	In development - Prototype, Beta version	WP3 - T3.1 - D3.1	CNIT	
VO-Security	This is a demo for the Security and Trust processes between objects from the VOStack. The repository tests the OpenID Connect for Verifiable Presentations (OIDC4VP), e.g., the verification of Verifiable Credentials and the generation of an Access Token using the Holder, Verifier and PEP-Proxy components.	In development - Prototype	WP3 - T3.1 - D3.1	OdinS	
VO-TSN	This repo contains a prototype implementation of a Centralized TSN control plane for the configuration of TSN-enabled devices, such as IoT Gateways and TSN bridges, for low-latency communication. The TSN control plane features a TSN schedule engine and a Gate Control List (GCL) controller. Communication with the TSN bridges is established using NETCONF.	In development - Prototype	WP3 - T3.2 - D3.1	UOM	Runs as a separate container and communicates via a custom REST API.
VO-SDN	This repository contains the prototype implementation of Reactive Routing component of VO stack. Reactive Routing maintains network connectivity between wireless IoT nodes and the IoT gateway. We adopt the Software-Defined Wireless Sensor Networking (SD-WSN) approach, which enables dynamic routing adjustments based on a global view of the network, e.g., handling routing changes due to mobility or signal interference. Furthermore, it provides logically-centralized and programmable routing control, allowing easy integration with the distributed control across the compute continuum, fostering enhanced network intelligence and monitoring capabilities.	In development - Prototype	WP3 - T3.2 - D3.1	UOM	Configurable protocols via a custom REST API, also supporting a cVO GUI.
VO-Discovery-Server	This repository contains the code for the Discovery Server that will be used to register Virtual Objects and make them discoverable.	In development - Prototype	WP3 - T3.1 - D5.1	NTUA	REST API where VO's can be registered and discovered.
SMO	This repository contains the code for the Synergetic Meta-Orchestrator (SMO). The SMO is a component in the NEPHELE ecosystem responsible for managing	In development - Prototype	WP5 - T5.1 -	NTUA	

	centralized information about application deployments and the overall orchestration of the resources spanning both the network and the computing domains. The SMO also interprets user-defined Intent declarations and constructs a deployment plan according to the user's request and the available resources.		D5.1		
NEPHELE-Dashboard	This repository contains the code for the NEPHELE Dashboard. The Dashboard is the front-end of the NEPHELE ecosystem where developers can construct Hyper Distributed Applications, deploy and monitor them.	In development - Prototype	WP5 - T5.2 - D5.1	NTUA	Web application operated by a UI.
NFVCL-NG	The NFVCL is a network-oriented meta-orchestrator, specifically designed for zeroOps and continuous automation. It can create, deploy and manage the lifecycle of different network ecosystems by consistently coordinating multiple artifacts at any programmability level (from physical devices to cloud-native microservices).	In development - Prototype	WP3 - T3.1 - D3.1	CNIT	Works through REST API calls

2.1.2 Blind evaluation

Within this section, we show the language distribution of each repository constituting the project as well as their statuses concerning building, working, and relevant notes. Table 2 serves as a comprehensive presentation of these components, offering insights derived from CLOC (Count Lines of Code) code analysis, inclusive of details pertaining to build and execution statuses.

Table 2: Language distribution and building and execution status of all repositories

Repository Name	Language Distribution (CLOC)	Build	Test	Additional notes
nephele-HDAR	Go 86.86% Smarty 11.1% Python 2.04%	Working	Working	
VO-LwM2M	Java 95.33% C++ 2.64% Python 1.67% PowerShell 0.34% Dockerfile 0.03%	Working	Working	
VO-Security	Kotlin 81.92% Python 15.43% Shell 0.79% JavaScript 0.64% Java 0.64%	Working	NA	
VO-WoT	Python 99.49% Shell 0.29% Dockerfile 0.19%	Working	Working	Already few tests in the repository

	Makefile 0.03%			
wot-py-2	Python 99.46% Shell 0.38% Dockerfile 0.16%	Working	Working	Already few tests in the repository
VO-SDN	GSC 62.7% Makefile 35.1% C 1.5% Dockerfile 0.4% Shell 0.3%	Working	Working	
VO-TSN	Prolog 93.6% Python 3.0% Dockerfile 2.7% Shell 0.7%	Working	Working	Tests found in run. Tests working locally but fail in a remote setup
VO-Discovery-Server	NA	Under development	NA	
SMO	NA	Under development	NA	
NEPHELE-Dashboard	NA	Under development	NA	
NFVCL-NG	Python 97.0% Jinja 2.8% Other 0.2%	NA	NA	Only APIs test found

2.1.3 Exposed APIs

In the present section, we expound upon the API (Application Programming Interface) details of the repositories, encompassing their presence, categorization, employed protocols, and access points to related documentation. Table 3 consolidates and succinctly encapsulates this array of information, providing a summarised overview of the repositories' API characteristics. The tabulated presentation aims to furnish a comprehensive reference for stakeholders seeking a clear understanding of the API-related aspects within the context of the examined repositories.

Table 3: Repositories' API information

Repository Name	***API***	**API type (language level, system level (comm and line), Web, I/O, single output, other protocols)**	**Public / Rest API (is it web exposable?)**	**API documentation**
nephele-HDAR	No	-Web	- REST (HTTP)	README[6][7]
VO-LwM2M	Yes	- Web - Other protocols	- REST (HTTP + CoAP) - MQTT	README[8]

[6] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-hdar/-/blob/main/hdar/README.md?ref_type=heads

[7] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-hdar/-/blob/main/hdar/api/docs/swagger.yaml?ref_type=heads

[8] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-security/-/blob/main/README.md?ref_type=heads

VO-Security	Yes	- Web	- REST (HTTP)	README[9]
VO-WoT	Yes	- Web - Other protocols	- REST (HTTP + CoAP) - WebSockets - MQTT	./docs[10]
wot-py-2	Yes	- Web - Other protocols	- REST (HTTP + CoAP) - WebSockets - MQTT	./docs[11]
VO-TSN	Yes	- Web	- REST (HTTP)	./documents[12]
VO-SDN	Yes	- Web	- REST (HTTP)	./docs[13]
VO-Discovery-Server	NA	NA	NA	NA
SMO	NA	NA	NA	NA
NEPHELE-Dashboard	NA	NA	NA	NA
NFVCL-NG	Yes	- Web	- REST (HTTP)	http://NFVCL_IP:5002/docs

2.1.4 Integration and Open Source quality requisites

This section explores the integration and open source quality requisites associated with the repositories within the NEPHELE project. Our investigation focuses on determining the repository's support for Docker, its capability for Continuous Integration/Continuous Deployment (CI/CD), and the necessity for dependency management. Additionally, we elucidate the type of testing employed and the repository's open-source licensing. Table 4 provides a succinct summary of the available assets at a repository level that contribute to the facilitation of DevOps, integration processes, and fulfilment of open source-oriented requisites.

Table 4: Integration and Open-Source quality requisites

Repository name	**Dockerfile**	**Available tests**	**Tests**	**CI/CD**	**Dependency Management**	**Licence**
VO-WoT	Yes	Functional	Yes	Smile Runners	No	MIT License
wot-py-2	Yes	Functional	Yes	Smile Runners	No	MIT License
nephele-HDAR	Yes	Functional	Yes	GitHub workflow, Smile Runners, Eclipse runners	Yes (go.mod)	Apache License
VO-LwM2M	Yes	Functional	Yes	Smile Runners	No	MIT License

[9] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-security/-/blob/main/README.md?ref_type=heads
[10] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-wot/-/tree/main/docs?ref_type=heads
[11] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/wot-py-2/-/tree/main/docs?ref_type=heads
[12] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-tsn/-/tree/main/documents?ref_type=heads
[13] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-sdn/-/tree/main/docs?ref_type=heads

VO-Security	Yes	NA	Yes	Smile Runners	Yes (pytz==2019.2)	Apache License
VO-TSN	Yes	Functional	Yes	Smile Runners	Yes (requests~=2.31.0, lxml~=4.9.3)	MIT License
VO-SDN	Yes	Functional	Yes	Smile Runners	No	MIT License
VO-Discovery-Server	NA	NA	NA	NA	No	MIT License
SMO	NA	NA	NA	NA	No	MIT License
NEPHELE-Dashboard	NA	NA	NA	NA	No	MIT License
NFVCL-NG	NA	NA	NA	NA	Yes (OpenStack instance, Ubuntu (22.04 LTS) instance, OSM 14, Python 3.11)	NA

2.1.5 Repositories documentation

This section shows the resources of documentation for repositories regarding their installation and usage guides. The investigation aims to ascertain the accessibility and comprehensiveness of documentation for each repository. Table 5 summarises the availability for each repository of installation documentation and usage documentation.

Table 5: Integration and Open-Source quality requisites

Repository name	Installation Documentation	Usage documentation	Notes
VO-WoT	Yes (Readme[14])	Yes (Readme[15])	
wot-py-2	Yes (Readme[16])	Yes (Readme[17])	
nephele-HDAR	Yes (Readme[18])	Yes (Readme[19])	
VO-LwM2M	Yes (Readme[20])	Yes (Readme[21])	

[14] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-wot/-/blob/main/README.md?ref_type=heads
[15] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-wot/-/blob/main/README.md?ref_type=heads
[16] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/wot-py-2/-/blob/main/README.md?ref_type=heads
[17] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/wot-py-2#development
[18] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-hdar/-/blob/main/README.md?ref_type=heads
[19] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-hdar/-/blob/main/README.md?ref_type=heads
[20] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-hdar/-/blob/main/README.md?ref_type=heads
[21] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-lwm2m/-/blob/main/README.md?ref_type=heads

VO-Security	Yes (PEP-Proxy/Readme[22])	Yes (Readme[23])	
VO-TSN	Yes (Readme[24])	Yes (Reame[25] & /agent/Readme[26])	
VO-SDN	Yes (Readme[27])	Yes (Readme[28])	
VO-Discovery-Server	Yes (Readme[29])	Yes (Readme[30])	
SMO	Yes (Readme[31])	Yes (Readme[32])	
NEPHELE-Dashboard	Yes (Readme[33])	Yes (Readme[34])	
NFVCL-NG	Yes (Readme[35])	Yes (Readme[36])	

[22] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-lwm2m/-/blob/main/README.md?ref_type=heads
[23] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-security/-/blob/main/README.md?ref_type=heads
[24] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-tsn/-/blob/main/README.md?ref_type=heads
[25] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-tsn/-/tree/main/schedulerSRC?ref_type=heads
[26] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-tsn/-/tree/main/agent?ref_type=heads
[27] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-sdn/-/blob/main/README.md?ref_type=heads
[28] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-sdn/-/blob/main/README.md?ref_type=heads
[29] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-discovery-server/-/blob/main/README.md?ref_type=heads
[30] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-discovery-server/-/blob/main/README.md?ref_type=heads
[31] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/smo/-/blob/main/README.md?ref_type=heads
[32] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/smo/-/blob/main/README.md?ref_type=heads
[33] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-dashboard/-/blob/main/README.md?ref_type=heads
[34] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/nephele-dashboard/-/blob/main/README.md?ref_type=heads
[35] https://github.com/s2n-cnit/nfvcl-ng/blob/master/README.md
[36] https://github.com/s2n-cnit/nfvcl-ng/blob/master/README.md

3 Overall integration status [M18]

3.1 Management tools

3.1.1 SCMs

In this section, we provide the overview of the source control management (SCM) options available for hosting the repositories within the NEPHELE project. The focus is on detailing the platforms designated for SCM and provided to project partners. Table 6 depicts available SCMs platform accessible to the project's collaborators.

Table 6: Source control management

Platform	Access	Gitlab-CI support	Notes
Gitlab (ZHAW)	Private	No	Meant for holding internal documentation
Gitlab (ECLIPSE)	Public	Yes	Meant for holding open source repositories

3.1.2 CI/CD

This section presents an overview of the available SCM solution used in the execution of the repositories within the NEPHELE project, specifically within the context of CI/CD processes. Table 7 depicts available CI/CD solutions provided to project partners.

Table 7: CM with CI/CD available for partners

Platform	Access	Gitlab-CI support	Notes
Gitlab (ECLIPSE)	Public	Yes	Meant for holding open source repositories
Gitlab (SMILE)	Private	Yes	Used as a remote alternative CI/CD provider

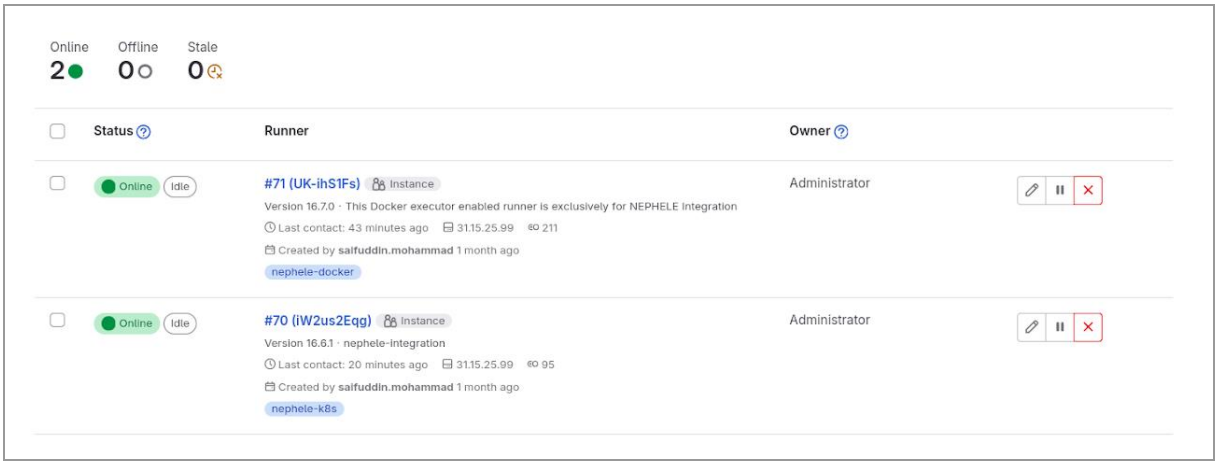


Figure 1: NEPHELE-Integration: Two CI/CD execution platforms are made available: one for the Docker applications, one for the Kubernetes applications.

3.2 Available infrastructures

3.2.1 Computation and evaluation environments

In this section, we show the overview of the available environments dedicated to computing and evaluating the repositories within the NEPHELE project. Table 8 depicts available infrastructures for computational and evaluative purposes.

Table 8: Computation and evaluation environments

Platform	**Access**	**Availability**	**Provider**	**Notes**
CI/CD Runners	Private	M18	SMILE	
CI/CD Runners	Private	M18	ECLIPSE	
CI/CD Runners	Local	NA	Partners	Gitlab-CI can support runners provided by users.
Micro-services platform (components execution)	Private	M18	SMILE	K8S Cluster (1 controller 2 worker nodes): amd64/linux 4 cores, 2.4 GHz 8 GiB, 50 GB Docker Executor: amd64/linux 4 cores, 2.4 GHz 8 GiB, 50 GB
Micro-services platform (VO applications execution)	Private / Public	M18	SMILE	K8S (3 nodes)
NEPHELE infrastructure (applications execution)	Private	M18	NTUA	OpenStack Platform 72 vCPUs, 234 GiB
Micro-services platform (components execution)	Private/Public	M18	UOM	K8S (1 master - 2 worker nodes w/ 4GB RAM and 4vCPUs)
Micro-services platform for network management (components execution)	Private/Public	M18	CNIT	OpenStack Platform

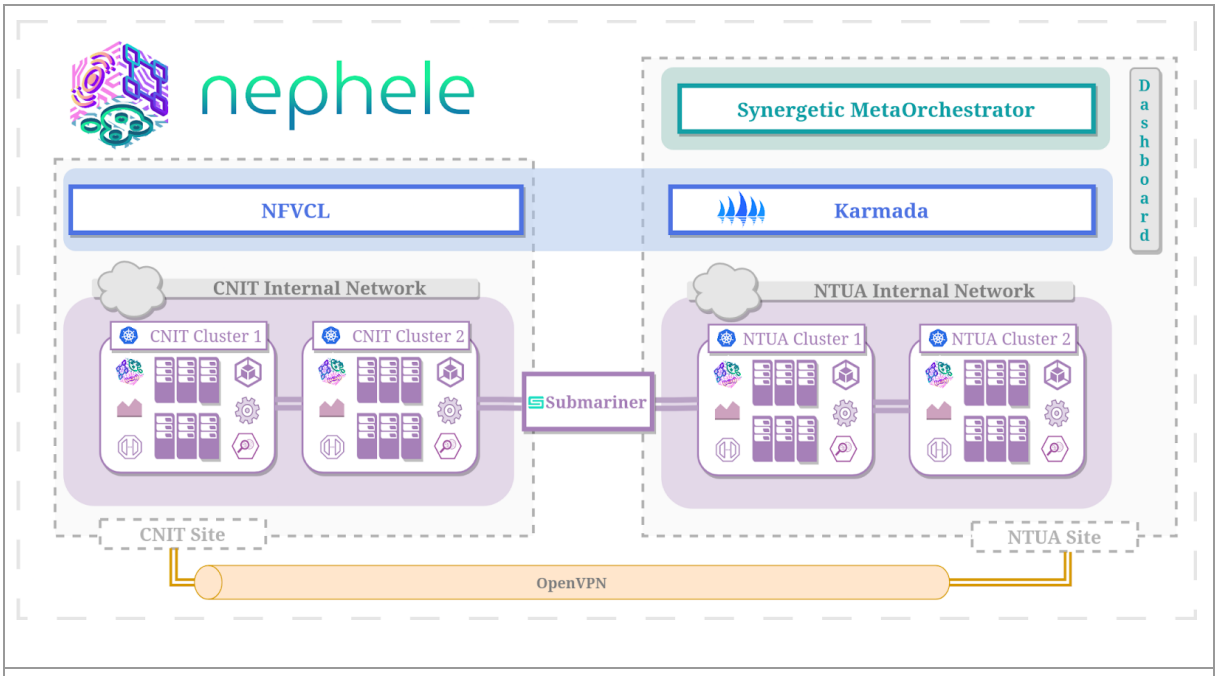


Figure 2: NEPHELE infrastructure Setup, the network overlay between partners provided infrastructures is connected through the Submariner solution.

In Figure 2 we depict the NEPHELE testbed on which Hyper Distributed Application Graphs can be instantiated. More specifically the testbeds between CNIT are connected through an OpenVPN tunnel. Kubernetes clusters are hosted on each testbed with the interconnection being achieved through the use of Submariner that handles the exposing of services across clusters. Karmada has been selected to orchestrate application graphs across multiple clusters propagating all the necessary resources to the desired cluster(s). The NFVCL component hosted on the CNIT testbed is part of the Network Manager of the infrastructure and handles Kubernetes cluster creation. Lastly, the Synergetic Meta-Orchestrator supervises and orchestrates the communication between the various components of the infrastructure.

3.3 Features coverage / Integration tests

This section presents a comprehensive overview of the integration tests associated with the NEPHELE project. The discussion encompasses detailed descriptions and statuses of the integration tests. Table 9 depicts overall integration tests.

3.3.1 Functional and Unit Tests for VO

The existing tests for the Virtual Object (VO) functionality can be divided into the following categories as presented in the below table.

These tests check the message exchange between client/server objects for the specific protocol. The tests include checking the message flows, states, proper protocol attribute values, and payload exchange. The tests can also be run against these Python versions 3.8, 3.9, 3.10, 3.11.

Table 9: Integration tests description and status

Test Category	Description	Related components	Status	Notes
Communication protocol: coap	These tests check the message exchange between client/server objects for the specific protocol. The tests include checking the message flows, proper protocol attribute values, and payload exchange.	Client/server	Implemented	
Communication protocol: http		Client/server	Implemented	
Communication protocol: mqtt		Client/server	Implemented	
Communication protocol: WebSocket		Client/server	Implemented	
Test all protocols	Checks Protocol bindings work as expected when multiple servers are combined within the same Servient	Client/server	Implemented	
Json codec	This category asserts the proper serialisation and deserialization of codec objects from json files in the forms of Unicode, byte and dictionary	Json codec	Implemented	
Thing functionality	Tests for checking client subscriptions to property change events, event emissions and actions, writing properties and proper interaction through the map-like interface	Thing Object	Implemented	
Consumed Thing functionality		Consumed Thing Object	Implemented	
Exposed Thing functionality		Exposed Thing Object	Implemented	
Servient functionality	Tests for creation and manipulation of catalogues of things and interaction with things	Servient Object	Implemented	

Thing Description (TD) functionality	Tests for creation and manipulation of Thing Description objects and interaction with Things and TD documents	Thing Description Object	Implemented	

Testing Methodology

Regarding the integration and testing/piloting phases a two-phased approach will be applied with respect to the pilots running.

The methodology that is considered to be followed, is the agile testing, as depicted in Figure 3. Briefly, this methodology considers testing and development as two intertwined phases and not sequential (i.e. testing following development).

With respect to the practical, separate, and integrated module testing, as proposed in [2], in Figure 3 we see a generic perspective of the integration, testing, and validation processes, which will be adopted by the NEPHELE team for the modules that will be integrated to produce the final NEPHELE platform (the numbers refer to steps, as described below, after the figure):

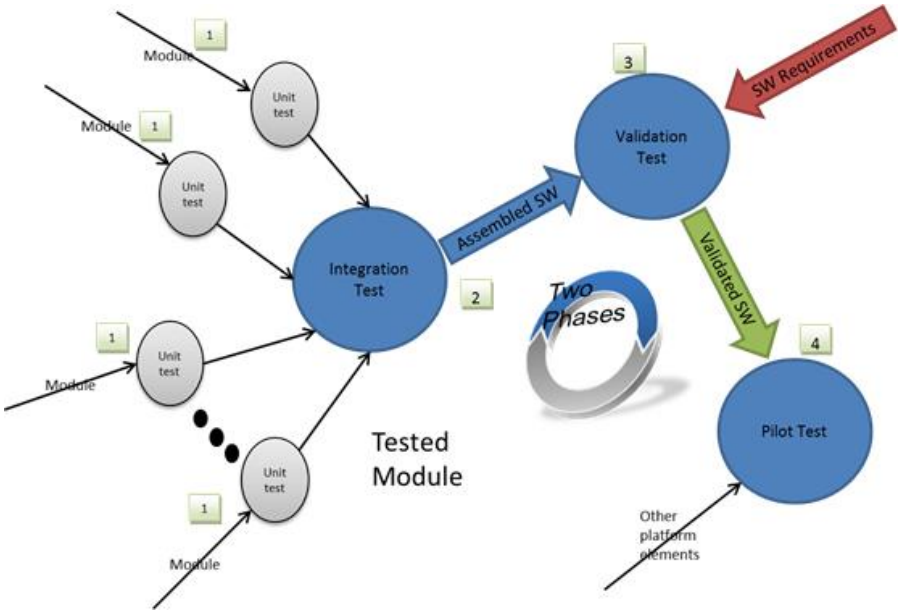


Figure 3: Testing-Methodology: Agile Testing for NEPHELE components/modules.

According to this generic methodology, our plan is composed of the following major steps:

1. Individual unit testing.
2. Integration of individual units to implement the NEPHELE platform.
3. Validation test of the integrated platform against the requirements.

4. Integration of the NEPHELE platform with PoCs
5. First PoC phase of integrated NEPHELE platform
6. Feedback to developers and implementation of corrective measures – quick individual unit testing against reported problems.
7. Integration of new individual unit modules.
8. Second round of final platform testing.

In Figure 4 we see the testing process for each integrated module:

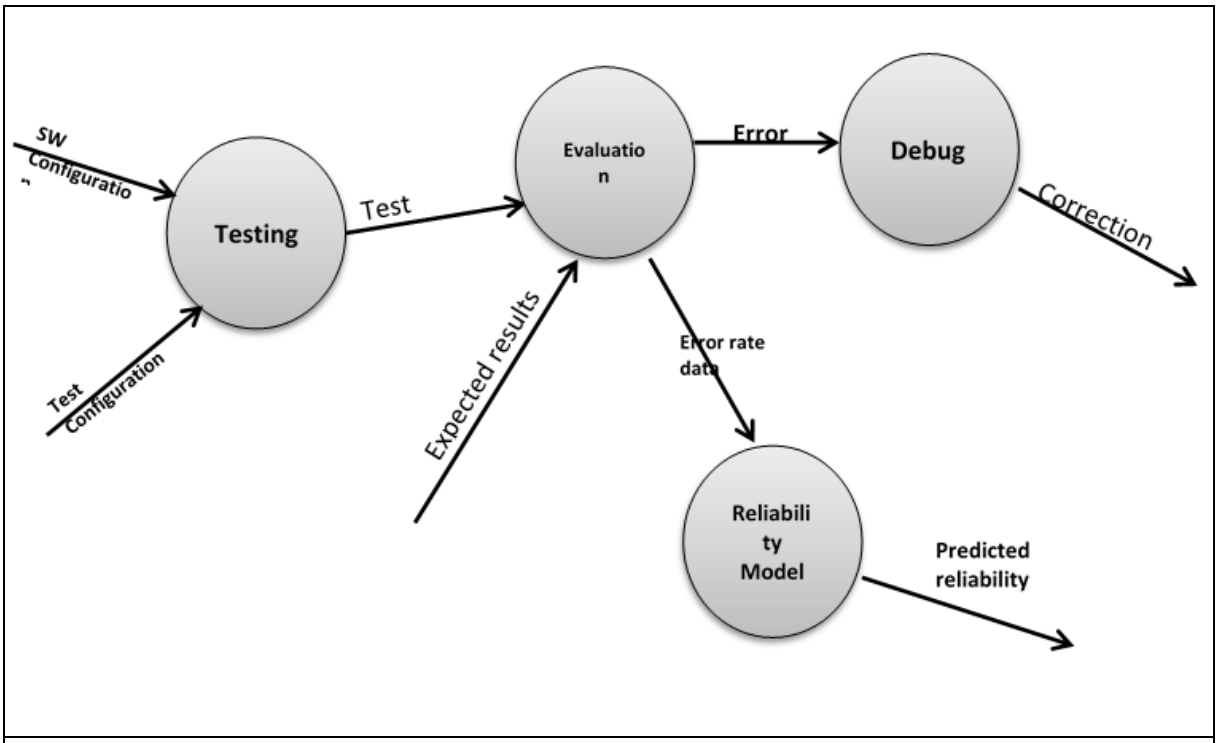


Figure 4: Testing-Process: Testing Process for each integrated Module.

With respect to system testing, there are generally two main methodologies: (i) Incremental testing, and; (ii) Non-incremental testing. We, briefly, discuss these two methodologies in what follows and state, with justification, our adoption decision. In the context of our description, the term low-level component refers to self-contained modules that perform a specific, relatively simple, computation and the term high-level component refers to modules that perform more complex operations and require other modules for their operation.

1) Incremental system integration

- Top-down testing: This approach requires the integration and testing to start from the highest-level components. This enables the testing of high-level system parts and the involved data flows and interfaces. This approach, usually, minimises the need for drivers, i.e. test modules that invoke others, since most modules are in place (at least the high-level ones). However, since lower-level modules are missing, stubs are necessary (perhaps numerous) to feed with inputs to the higher-level component which is under testing, until the real lower-level components become available. In addition, the lower-level components are tested late in the integration phase leaving limited time for system-level corrective actions. The exact forms and functionality

of the stubs and drivers, since they form very low code-level components of the testing phase, depend heavily on the actual code of the tested modules, which is not available at the time of the writing of this deliverable, and will be decided later by the involved partners.

- Bottom-Up testing: Contrary to top-down testing, this testing strategy starts from the lower-level system components. This, also, minimizes the need for stubs and identifies low-level problems early, before the integration begins. However, more drivers are needed to invoke, appropriately, the lower level components because the higher level components (which invoke the lower level ones) are missing.

- Sandwich Testing: This is also called Hybrid Integration Testing, and combines the two approaches discussed above. According to this approach, lower-level modules are tested in parallel, while their integration also proceeds along with testing and is tested itself too. Accordingly, the need for stubs and drivers is minimized. However, this approach is a little less systematic than the top-down and bottom-up approaches and may need more coordination effort.

2) Non-Incremental system testing:

- Big-bang testing: In this approach, all (or a large portion) of the modules that compose the system are integrated and tested. According to this testing methodology, the testing starts from the final system and not the individual components level. However, in this testing strategy, if an erroneous behavior is detected while a test is performed, it is very difficult to isolate the components that fail and lead to this kind of behavior, since attention is not paid to the component operation or component interface level during the testing.

After consideration of these options, the development teams are opting for the "sandwich approach". This methodology is deemed more suitable, due to the fact that partners, in parallel, develop their own modules according to the specifications set in the architecture deliverables (most notably D2.2). Thus, the module developers use the bottom-up approach to test their low-level modules and then Task T5.1 (the platform integration task) uses the bottom-up approach to follow and support the piloting phases.

Moreover, the selected testing methodology will utilize the CI/CD paradigm, which facilitates the automation of testing stages, by incorporating them in CI/CD pipelines enabling to speed up the testing procedure into a standardized repetitive cycle. This cycle facilitates the faster finding of possible bugs and faster delivery of SW in general.

4 Integration proposal and prototyping

4.1 Description of the resulting architecture and packaging

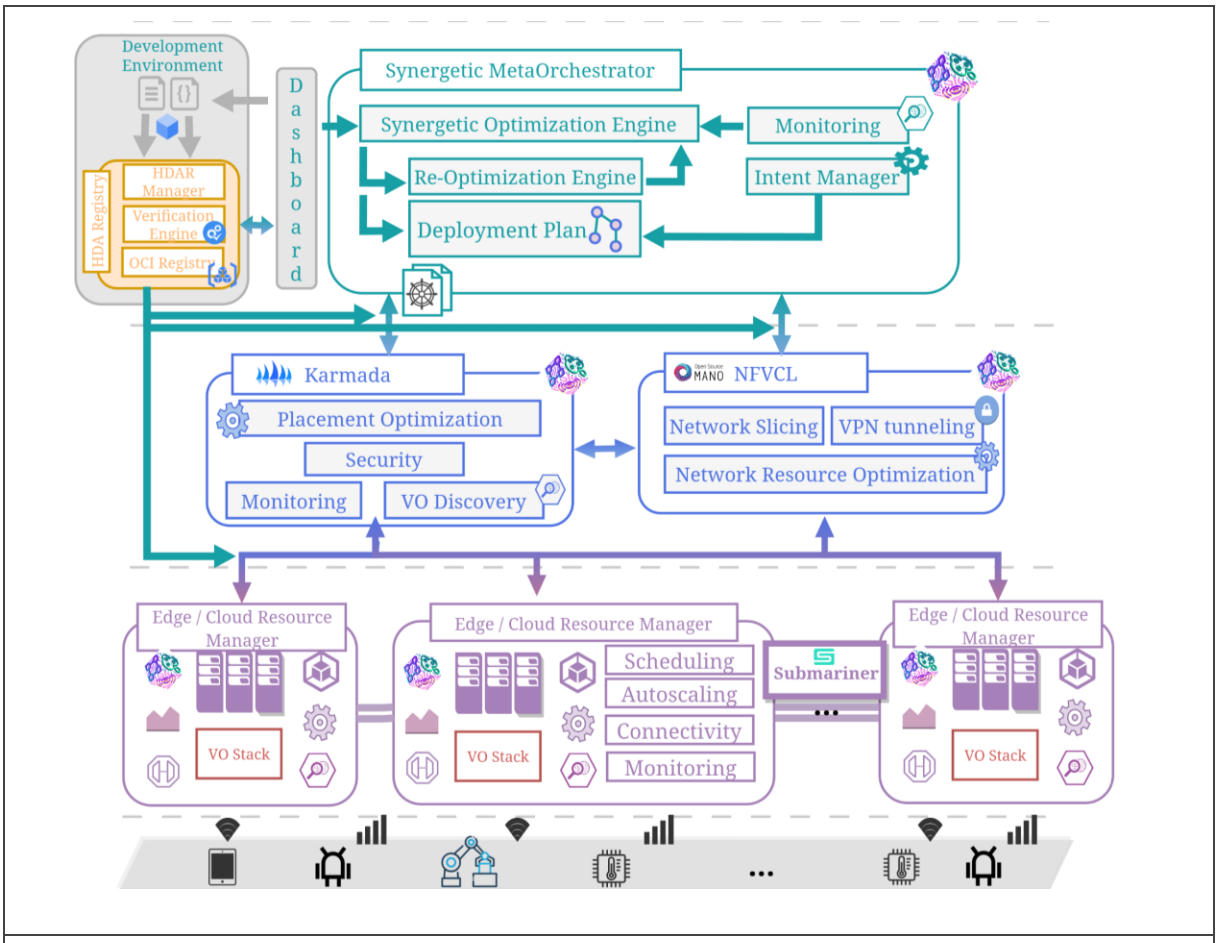


Figure 5: NEPHELE-Platform-Architecture: The overall applicative platform is managed through a dashboard-type UI. It supports the orchestration of operations through the Synergetic Meta Orchestrator.

The architecture of NEPHELE was initially presented in deliverable D2.1 "Requirements, Use Cases Description and Conceptualization of the NEPHELE Reference Architecture" in month 9 and updated in its final version in deliverable D2.2 "NEPHELE Reference Architecture Final Specification" in month 18. In this section we provide a short description of the NEPHELE architecture as illustrated in Figure 5, to facilitate the presentation of the initial version of the first integrated NEPHELE Platform.

The overall architecture incorporates multiple components that interact with each other to achieve the desired functionality. The NEPHELE Dashboard is composed of a front-end website that is interactable and allows the deployment of Hyper Distributed Applications and a Python back-end that interfaces with and makes calls to other components. Also, an SQL database is used to store information related to application deployments. Before application deployment, the developer uses the development environment to create software artifacts of the packaged applications that are then pushed to a Distribution OCI registry. The back-end of the NEPHELE dashboard is able to communicate with the

OCI artifact registry to fetch artifact information as well as with the SMO to enforce deployment plans. The SMO creates deployment plans by sending requests to both the NFVCL Network Manager and the Karmada Mutli-Cluster Manager. After instantiation, inter-cluster communication is achieved with the Submariner tool that seamlessly allows service discovery across clusters.

SMO:
The Synergetic Meta Orchestrator (SMO) is a layer that oversees the deployment of Hyper Distributed Application Graphs by orchestrating and communicating with the various other layers of the architecture to create and propagate a deployment plan. More specifically the SMO is tasked with receiving a Hyper Distributed Application Graph descriptor with the specification of the application graph along with an Intent formulation describing the desired application performance levels and constraints. The SMO interfaces with the Network Manager and the Multi-Cluster Manager to make network-related and infrastructure-related decisions respectively. Moreover, the SMO translates the given Intent into enforceable deployment plans leveraging the information provided by the Network and Multi-Cluster Managers as mentioned previously. The SMO is currently under development and we will use a Flask framework to build the necessary Rest API for communication with all other components. All mechanisms described in Deliverable D2.2 and D4.1 will be included as backend mechanisms here.

Dashboard:
The NEPHELE Dashboard is a developer-facing frontend of the NEPHELE project allowing the interaction with the underlying infrastructure and the deployment of Hyper Distributed Applications. The Dashboard is currently under development using Vue.js as frontend and Python Flask as backend. The developer can create Hyper Distributed Application Graphs by combining various application components that use already created artifacts hosted on a Hyper Distributed Application Registry. Afterward, the developer can request the deployment of the created Graph which forwards the request to the SMO constructing and enforcing a deployment plan. Moreover, the dashboard can be used to monitor the deployment of the graph and view metrics related to its performance. To ensure that the Thing Description JSON files provided by the developers are valid a validator has been used whose JSON schema can be found here[37]. The validator ensures that during the deployment of the VO, the given Thing Description is valid. Sample Mockups of the NEPHELE dashboard and a description of the functionalities can be found in the Appendix section of this deliverable.

Development Environment:
It is the developer-facing component of the NEPHELE project's architecture. Apart from the development sandbox built directly over GitLab, there is the need to develop and deploy a system that takes care of the storage, distribution, and verification of all the artifacts produced by the developer. The HDA Registry is a system that is built following a micro-service architecture combining and extending functionalities around the official OCI Registry microservice (distribution[38]). Regarding the artifacts, two contributions have been addressed. First, the NEPHELE-specific developer CLI (hdarctl) was developed to interact with the HDAR in a similar way as other CLI tools from the CN ecosystem (Helm, docker). Second, the HDAR is a system deployed in K8s along with the rest of the platform component, which provides a verification pipeline that ensures the correctness and integrity of the artifacts, not only based on the syntax and composition of files but also based on the semantic validation of the different links and values in the descriptors. From the perspective of the NEPHELE platform, the HDAR exposes

[37] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-wot/-/blob/8418e272dc8e7719bb0568d9e89cc5f0455dd6a6/wotpy/wot/validation.py
[38] https://github.com/distribution/distribution

a series of custom REST APIs that aim to assist with the artifact handling needs of other components, mainly the Dashboard. The available swagger is continuously updated here[39].

Multi-Cluster Resource Manager (Karmada):
The Multi-Cluster Resource {Manager/Orchestrator} component within the NEPHELE ecosystem manages multiple (Kubernetes) clusters across diverse environments and providers, offering a centralized and unified interface for administration and operation. This centralized control simplifies the management process, reduces the complexity of handling disparate systems, and minimizes the potential for errors, enhancing overall efficiency. This level of orchestration enables scalability, resilience, and cross-environment compatibility. Among the available multi-cluster resource {managers/orchestrators}, Karmada is one of the best candidates. As an open-source project under the Cloud Native Computing Foundation (CNCF), Karmada ensures transparency and flexibility. Also encourages widespread adoption and community-driven enhancements. Being part of the CNCF, Karmada benefits from a robust ecosystem, contributing to its credibility and the assurance of continued evolution and support.

Karmada Architecture:

Karmada orchestrates (Kubernetes) clusters by leveraging a comprehensive suite of components designed for efficient multi-cluster management across diverse environments. The architecture includes the Karmada-Apiserver which serves as the gateway to the Karmada control plane, offering compatibility with the Kubernetes API. The Karmada-Aggregated-Apiserver extends the Kubernetes API to manage and access resources across multiple clusters. The Kube-Controller-Manager and Karmada-Controller-Manager manage standard and custom resources across member clusters. The Karmada-Scheduler optimizes resource distribution across clusters based on scheduling constraints. Karmada-Webhook components enforce custom policies through request validation and modification. Add-ons like the Karmada-Scheduler-Estimator and Karmada-Descheduler refine scheduling accuracy and resource allocation, while Karmada-Search offers global search capabilities across the clusters. CLI tools such as karmadactl and kubectl provide command-line management options, streamlining cluster administration and policy application. Clusters can join the multi-cluster in both Push or Pull modes, with Push mode directly connecting Karmada to the cluster's Kube-Apiserver, while Pull mode employs Karada-Agent within the cluster as the mediator between Karmada and the cluster. The overall architecture is visualized in the following Figure 6:

[39] https://nephele-platform.netmode.ece.ntua.gr/hdarapi/swagger/index.html

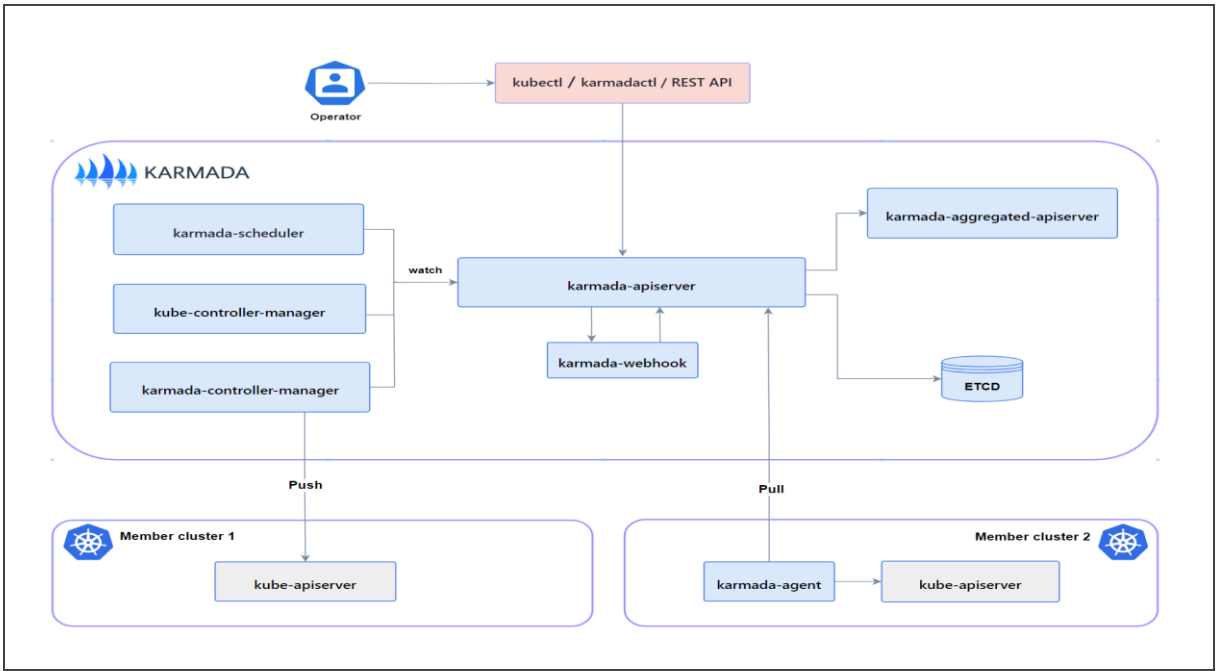


Figure 6: NEPHELE Multi-Cluster Resource Manager technical architecture: At a system level, operational actions are managed through a mix of APIs, including the native Kubernetes, cli i.e., kubectl.

Inter-Cluster communication:

Karmada uses Submariner to enable direct networking between Pods and Services across different Kubernetes clusters. It's designed to be network plugin agnostic, ensuring broad compatibility and seamless operation within the Kubernetes ecosystem. Submariner's architecture includes several core components that work together to facilitate this connectivity:

- Gateway Engine: This is deployed in each participating cluster and manages the establishment of secure tunnels between clusters for direct pod-to-pod and service-to-service communication.
- Broker: Serves as a central information exchange hub for clusters, handling the distribution of connectivity meta-data and service discovery information.
- Service Discovery (Lighthouse): Enables the discovery of services across cluster boundaries, making it possible for services in one cluster to find and communicate with services in other clusters.
- Globalnet Controller: Provides a solution for overlapping CIDRs between clusters by ensuring that cross-cluster communication is possible even when IP address spaces conflict. Karmada is not compatible with Globalnet.
- Network Plugin Syncer: Ensures that the network configuration is consistent across clusters, adapting to the specific requirements of different network plugins.
- Route Agent: Facilitates the routing of cross-cluster traffic by configuring the necessary routes on each node in participating clusters.

For inter-cluster communication, Submariner utilizes the ServiceExport and ServiceImport Custom Resource Definitions (CRDs) to specify which services should be made available across clusters. The ServiceExport CRD is used to mark a service for export to other clusters, while the ServiceImport CRD represents an imported service in a consuming cluster, enabling seamless service discovery and connectivity across the multi-cluster environment. The overall architecture of Submariner is visualized in the following image:

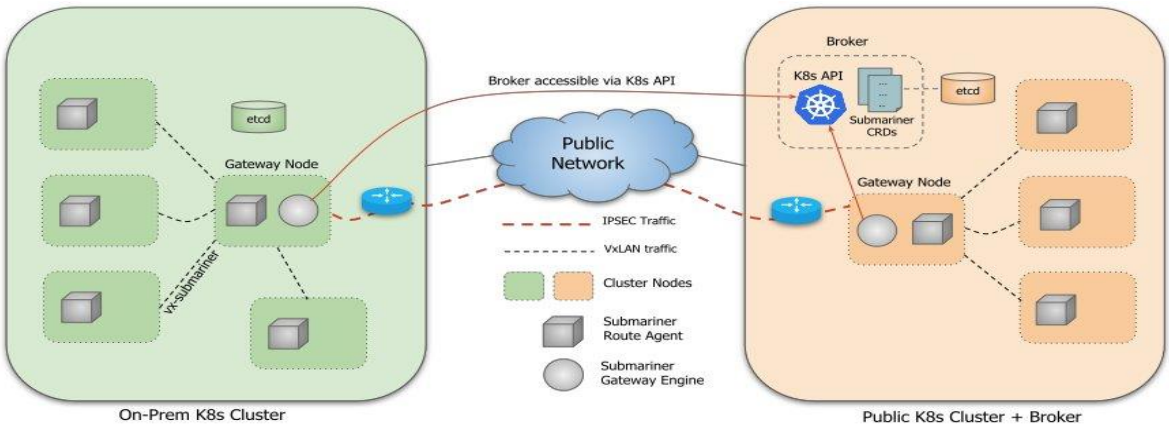


Figure 7: High-level representation of distributed controller principles: Network control of private and public clusters is enabled through the public network exposure of both Submariner and Kubernetes interfaces.

Network Resource Manager (NFVCL):

The NFVCL is a network-oriented meta-orchestrator, specifically designed for zeroOps and continuous automation. It can create, deploy, and manage the lifecycle of different network ecosystems by consistently coordinating multiple artifacts at any programmability level (from physical devices to cloud-native microservices).

A network ecosystem is meant to be a completely functional network environment, such as a 5G system, an overlay system for network cybersecurity, or a simple application service mesh.

For their nature, these environments are realized through heterogeneous Network Functions (xNFs – i.e., Physical NF, Virtual NF, and cloud-native Kubernetes NF), which are usually to be realized over highly distributed infrastructures.

As defined in the ETSI NFV standard, xNFs are managed by the NFVO through end-to-end Network Service Instances. Every Network Service can include one or more xNF instances, and it is meant to be deployed over a single geographical facility, which may correspond to a computing facility and/or a physical device (e.g., a gNodeB, an O-RAN Radio Unit, a P4 switch, etc.).

The key feature of the NFVCL is process automation (ZeroOps): when a blueprint is instantiated (Day0-1), it does not need interaction with the user, a description of the topology template and initial configuration parameters are sufficient for a blueprint to be deployed. This feature also includes post-initialization configuration (Day2) and blueprint deletion (DayN). The relevant live documentation and swagger can be found here[40].

[40] https://nfvcl-ng.readthedocs.io/en/latest/index.html

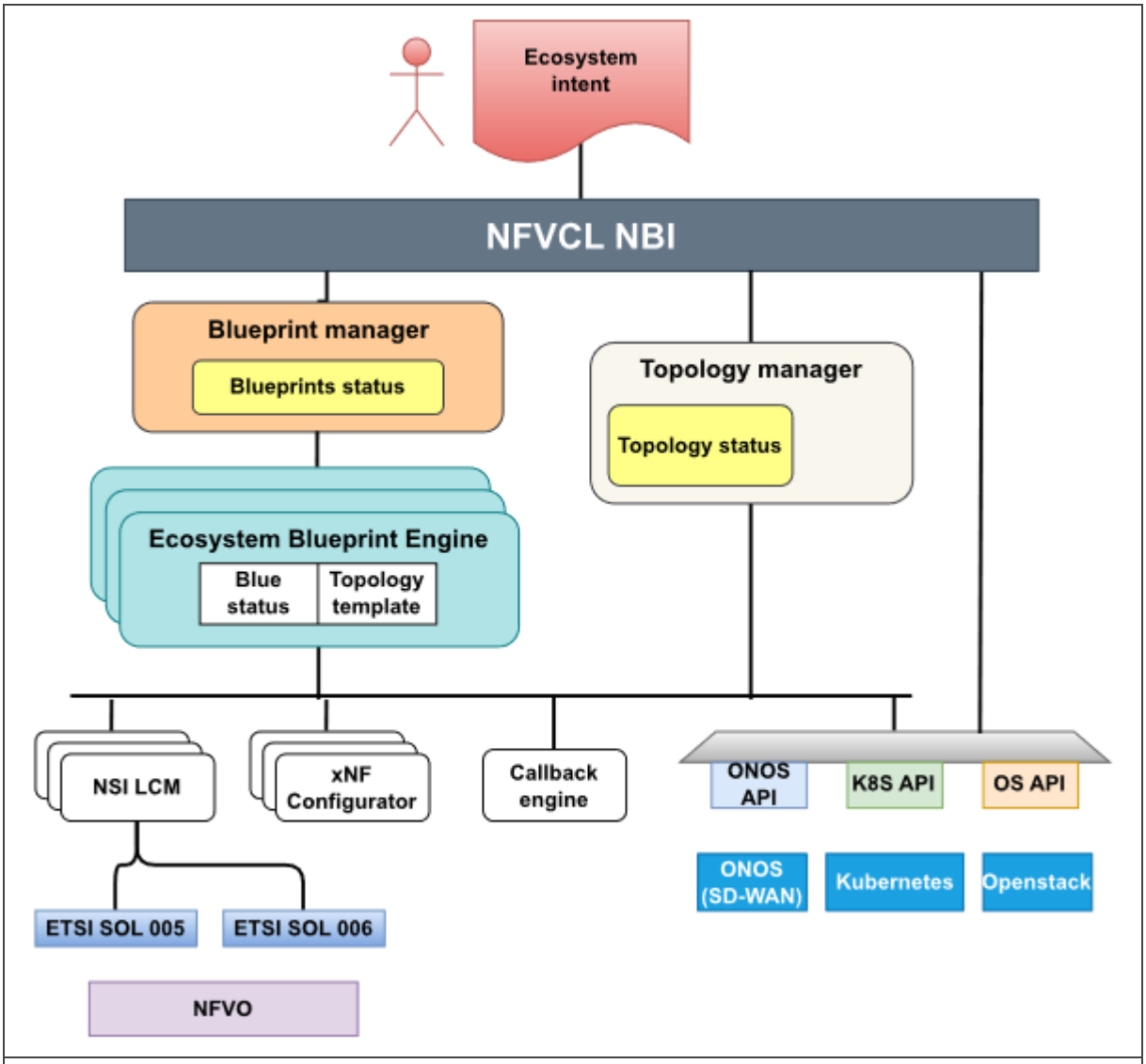


Figure 8: NFVCL reference architecture - The NBI approach enables a distribution of concerns between the various logical and physical components of the SDN. Stakeholders can leverage blueprints and topologies to define fine-grained dynamic network resource configuration and access.

Virtual Object:

The Virtual Object (VO), is the virtual counterpart of IoT devices deployed on the premises of Edge/Cloud Clusters. The VO is a lightweight software stack[41], based on two different specifications, i.e., W3C Web of Things[42] and OMA LwM2M[43], that stores the necessary information produced by IoT devices and therefore acts as a broker between the devices and an application graph. In NEPHELE we

[41] https://netmode.gitlab.io/vo-wot/

[42] https://www.w3.org/WoT/

[43] https://technical.openmobilealliance.org/index.html

maintain and develop two different repositories with live documentation for each specification. Complete versions will be delivered in D3.2 on Month 24. The Web of Things repo can be found here[44] (documentation available here[45]), while the OMA LwM2M is here[46].

[44] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-wot

[45] https://netmode.gitlab.io/vo-wot/

[46] https://gitlab.eclipse.org/eclipse-research-labs/nephele-project/vo-lwm2m

5 Documentation

5.1 NEPHELE integration bootstrap [M18]

5.1.1 Motivations

Considering microservices-oriented development and later deployment, more than one approach can be chosen from research and development teams. These various approaches may bring specifics within the project toolkit, limiting or delaying the seamless integration effort between partners.

To unify the development effort, while supporting non-experts in providing their results towards integration, a specific bootstrap documentation has been conceived. It is aimed to provide information about microservices development and CI/CD usage while exposing integration-related technical choices to the consortium discussion.

It has been delivered at M18.

5.1.2 Content

The integration bootstrap documentation is made of a git repository, holding:

- text documentation
- code examples

The documentation is split into two main parts: CI/CD principles and configuration means, and microservices development and packaging. The first part is meant for users not aware of the possibility of hosting CI/CD operations configuration along the source code management, within the Gitlab platform. A working example of a gitlab-ci.yml file is provided along with documentation for its structure. The second part holds techniques and approaches to ease microservices build and deployment, notably packaging as a single image or as an operator.

Code is meant to support documentation over demonstrating builds and tests ran within the CI/CD pipelines. Files assets are made available to showcase microservices build, locally and remotely, and tests, through two frameworks, Robot framework and Gherkin, enabling functional testing.

5.1.3 Availability

The NEPHELE integration bootstrap is made available over the project's private Gitlab platform under an MIT-free software license.

6 Conclusions

Halfway into the NEPHELE project duration, public, open source, achievements can be observed, evaluated, and measured. The main architectural components supporting the deployment and execution of VOs, one of the core technical innovations proposed in the project roadmap, are made available with their relevant documentation and test suites.

Continuous integration has been set for all the public code repositories with respect to the DevOps best practices. Continuous deployment is being discussed along with the provision of project-dedicated infrastructures.

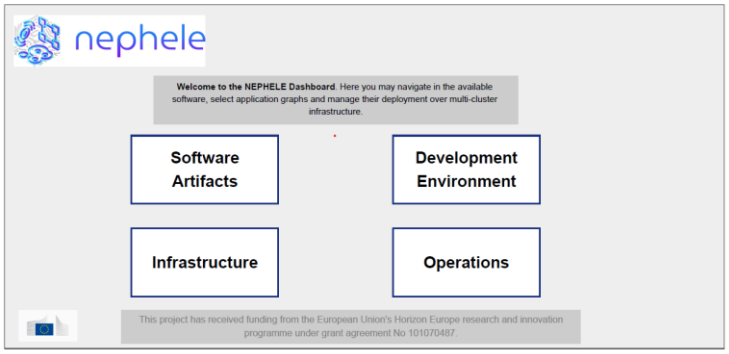
A standard architecture, enabling the integration of individual results within a single platform prototype is proposed. The main existing and upcoming functional blocks are detailed.

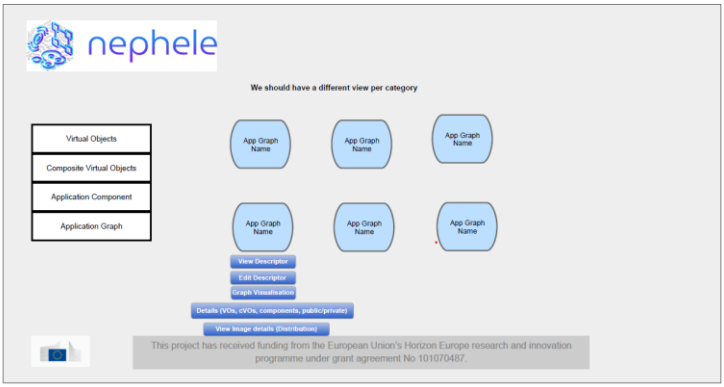
Information held in this document will be used as a follow-up base to report and share the various integration levels' statuses. Some aspects may contribute to the development or the improvement of architectural components, such as tests, guidelines, and quality constraints.

As of the M18 milestone, future steps are oriented towards adding up iteratively newly developed components as well as their overall integration support. The resulting status will in turn be depicted to provide at M24 and M36 a clear understanding of the NEPHELE platform integration achievement, in terms of maturity, infrastructure(s) deployment, and functional support.

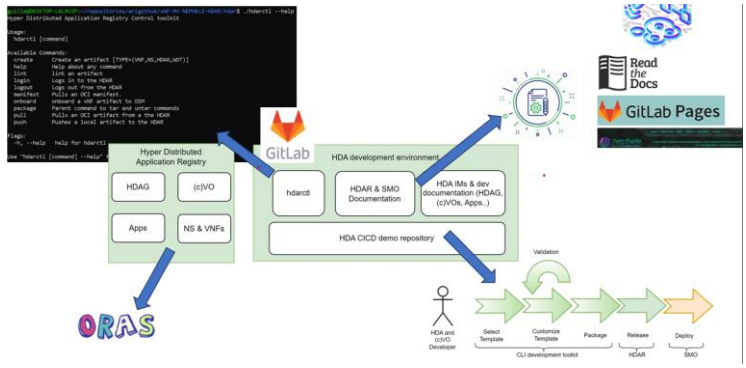
7 Appendix

7.1 Dashboard Initial Mock-ups

Landing page 	The landing page serves as the landing page of the NEPHELE Dashboard where users can authenticate themselves and log into the dashboard.
Home page 	The home page allows the user to navigate the various pages of the dashboard and in particular the Software Artifacts, the Development Environment, the Infrastructure and the Operations pages. These pages are explained below.
Software artifacts 	The software artifacts pages list all the various Software Artifacts that are available on the platform. These include the Virtual Objects, the Composite Virtual Objects, the Application Components and the Application Graphs.

Software artifacts - VO 	This view of the Software artifacts page shows a list of available VOs where more information about the Descriptor files or the used Image file can be requested.
Software artifacts - Application Graph 	This page shows a listing of Application Graphs. An Application Graph's descriptor can be viewed and edited while more information can be solicited including a Visualisation of the Graph.
Infrastructure 	The Infrastructure page lists all the underlying Infrastructure components including information about their specifications, location and their availability.
Mockup in construction.	The Operations page includes a list of the instantiated application graphs and VOs/cVOs along with a Grafana view showing live metrics about these deployments.

7.2 Development environment



The Development Environment contains information and instructions on how to package software artifacts and push them to the Hyper Distributed Application Registry.

8 Bibliography

[1] L. Crispin and J. Gregory. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional, 2009.

[2] G.J. Myers and C. Sandler. The Art of Software Testing. Wiley, 3rd Edition, 2011.

[3] R. Pressman and B. Maxim. Software Engineering: A Practitioner's Approach. McGraw-Hill Education, 8th Edition, 2014.