NebulOuS

A META OPERATING SYSTEM FOR BROKERING
HYPER-DISTRIBUTED APPLICATIONS ON
CLOUD COMPUTING CONTINUUMS

D4.2

NEBULOUS SECURE CROSS-CLOUD AND FOG APPLICATIONS DEPLOYMENT & ORCHESTRATION BASED ON SMART CONTRACTS

[17/06/2025]

Funded by
the European Union

Grant Agreement No.	101070516
Project Acronym/ Name	NebulOuS - A META OPERATING SYSTEM FOR BROKERING HYPER DISTRIBUTED APPLICATIONS ON CLOUD COMPUTINGCONTINUUMS
Topic	HORIZON-CL4-2021-DATA-01-05
Type of action	HORIZON-RIA
Service	CNECT/E/04
Duration	36 months (starting date 1 September 2022)
Deliverable title	NebulOuS Secure Cross-Cloud and Fog Applications deployment & Orchestration based on smart contracts
Deliverable number	D4.2
Deliverable version	V1.0
Contractual date of delivery	31 May 2025
Actual date of delivery	17/06/2025
Nature of deliverable	OTHER
Dissemination level	Public
Work Package	WP4
Deliverable lead	ACTIVEEON
Author(s)	Ankica Barisic (ACTIVEEON), Moritz von Stietencron (BIBA), Nikos Papageorgopoulos (UBI), Sarantis Kalafatidis (UBI) & Geir Horn (UiO)
Abstract	The introduction of smart contracts in the deployment mechanism to support approvable service levels (T4.3) and report on the final iteration of all the WP4 mechanisms for providing a secure deployment and orchestration of applications on heterogeneous cross-clouds and fog resources.
Keywords	cloud-edge continuum, fog computing, orchestration, networking, vpn, security, access control, kubernetes.

DISCLAIMER

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Directorate-General for Communications Networks, Content and Technology. Neither the European Union nor the granting authority can be held responsible for them.

COPYRIGHT

CONTRIBUTORS

Name	Organization
Ankica Barisic	AE
Moritz von Stietencron	BIBA
Nikos Papageorgopoulos, Sarantis Kalafatidis	UBI
Geir Horn	UiO

PEER REVIEWERS

Name	Organization
Yiannis Verginadis	ICCS
Paweł Skrzypek	7bulls

REVISION HISTORY

Version	Date	Owner	Author(s)	Comments
0.1	10/03/2025	AE	Ankica Barisic	Preliminary draft
0.2	07/04/2025	BIBA	Moritz von Stietencron	Chapter 6
0.3	17/04/2025	UBI	Giannis Ledakis	Chapter 4 & 5
0.4	25/04/2025	AE	Ankica Barisic	First draft
0.5	05/05/2025	UBI	Nikos Papageorgopoulos, Sarantis Kalafatidis	Chapter 4 & 5
0.6	06/05/2025	AE	Ankica Barisic	First draft
0.7	12/05/2025	UiO	Geir Horn	Chapter 2
0.8	15/05/2025	EUT	María Navarro	Version for Review
0.9	30/05/2025	ICCS, 7bulls	Yiannis Verginadis, Paweł Skrzypek	Review
1.0	16/06/2025	EUT	María Navarro	Final

TABLE OF ABBREVIATIONS AND ACRONYMS

Abbreviation/Acronym	Open form
AE	Activeeon
ABAC	Attribute-Based Access Control
AMD64	AMD 64-bit x86 instruction set architecture
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARM	Advanced RISC Machines
AWS	Amazon Web Services
BQA	Brokerage Quality Assurance
CI	Continuous Integration
CD	Continuous Delivery or Deployment
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CPU	Central Processing Unit
CRD	Custom Resource Definitions
CRUD	Create, Read, Update, Delete
DLTs	Distributed ledger technologies
DNS	Domain Name Service
DT	Digital Twin
DTO	Data Transfer Object
EC2	Amazon Elastic Compute Cloud

EU	European Union
FGW	Fabric Gateway
FPGA	Field-Programmable Gate Array
GCE	Google Compute Engine
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure-as-a-Service
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPSec	Internet Protocol Security protocol suite
JSON	JavaScript Object Notation format
K8s	Kubernetes
LDAP	Lightweight Directory Access Protocol
(Meta-)OS	(Meta-)Operating System
NAT	Network Address Translation
NPM	Node Package Manager
OAM	Open Application Model
ONM	Overlay Network Manager
OPA	Open Policy Agent
PA	ProActive
PERM	Policy, Effect, Request, Matchers
PKI	Public key infrastructure

PoS	Proof-of-Stake
PoW	Proof-of-Work
QoS	Quality of Service
R&D	Research and Development
RAM	Random Access Memory
RBAC	Role-Based Access Control
RDBMS	Relational Database Management System
REST	Representational State Transfer
RM	Resource Manager
RTT	Round-trip Time
SAL	Scheduling Abstraction Layer
SC	Smart Contract
SCE	Smart Contract Encapsulator
scp	Secure Copy
SLA	Service Level Agreement
SLO	Service Level Objective
SSH	Secure Shell protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VM	Virtual Machine
VPC	Virtual Private Cloud
VPN	Virtual Private Network
WG	WireGuard

WP	Work Package
XACML	eXtensible Access Control Markup Language
YAML	Yet Another Markup Language

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Deliverable D4.2 of the NebulOuS project presents the final iteration of Work Package 4 (WP4), focused on secure deployment and orchestration of cross-cloud and fog applications through smart contract-based mechanisms. Building on the foundational concepts and preliminary results reported in D4.1, this document demonstrates the evolution and integration of WP4 technologies into the NebulOuS Meta-Operating System, delivering enhanced capabilities for brokering hyper-distributed applications across heterogeneous infrastructures.

This deliverable D4.2 is focused on

- The introduction of smart contracts in the deployment mechanism to support approvable service levels
- Orchestration of applications on heterogeneous cross-clouds and fog resources using deployment manager and network
- Secure deployment of applications on heterogeneous cross-clouds and fog resources
- Digital twin of deployment optimization

1 INTRODUCTION

D4.2 main goal is the introduction of smart contracts in the deployment mechanism to support approvable service levels (T4.3) and report on the final iteration of all the WP4 mechanisms for providing a secure deployment and orchestration of applications on heterogeneous cross-clouds and fog resources.

In particular, D4.2 addresses four key innovations:

1. **Executionware Architecture Enhancements** – Advancing the Deployment Manager (SAL) and Execution Adapter (ProActive) to support fully automated, reproducible, and scalable deployment pipelines across cloud, edge, and hybrid environments.

2. **Smart Contract-Enabled SLA Management** – Introducing blockchain-based service level agreement enforcement mechanisms, bridging the gap between declarative SLAs and runtime smart contract execution.

3. **Privacy-Preserving Network Orchestration** – Expanding the secure overlay networking capabilities of the Overlay Network Manager (ONM) to facilitate encrypted communications, NAT traversal, and cross-node integration through WireGuard and Headscale/Tailscale solutions.

4. **Security and Policy Framework** – Implementing privacy-by-design strategies and runtime observability via integration of OPA Gatekeeper, Cilium, and Tetragon to manage access control, enforce network segmentation, and ensure runtime compliance across deployment targets.

The report is structured to provide a comprehensive technical overview of these mechanisms, while also highlighting their alignment with the broader NebulOuS objectives of automation, security, and trustworthiness in distributed cloud-edge deployments. In particular:

- **Section 2** details the architecture, lifecycle, and operation of the enhanced Executionware stack.

- **Section 3** explains the SLA-to-smart contract transformation process and outlines the technical infrastructure enabling automated SLA enforcement.
- **Section 4** covers secure networking through ONM, with specific focus on overlay creation, device onboarding, and Kubernetes integration.
- **Section 5** presents the privacy and security framework implemented across the deployment lifecycle.
- **Section 6** introduces the orchestration of digital twins in cloud-edge continuums as a key enabler for adaptive and intelligent deployments.
- **Section 7** summarizes the results and outlines the next steps toward the final system integration.

Overall, this deliverable demonstrates how NebulOuS translates complex orchestration and compliance requirements into automated, trustworthy, and scalable deployment workflows. The developed technologies lay the foundation for the platform's second release, enabling dynamic, secure, and SLA-aware deployment of applications in a cloud-edge continuum.

2 DEPLOYMENT & ORCHESTRATION IN HETEROGENEOUS ENVIRONMENTS

In the context of *Task 4.1 "Deployment & Orchestration in heterogeneous environments"*, NebulOuS has developed **Executionware** components that enable seamless deployment and orchestration of applications across the cloud-edge continuum. These components—namely, the **Deployment Manager (i.e., SAL)** [2],[3] and the **Execution Adapter (i.e., ProActive)** [4][5][6] are responsible for translating optimized deployment plans into actionable operations across a wide array of target infrastructures.

NebulOuS supports deployment on public cloud platforms such as AWS, Azure, and Google Cloud, as well as private clouds based on OpenStack and edge devices spanning AMD, ARMv7, and ARMv8 architectures. These heterogeneous resources are onboarded and managed through an integrated system, where the Execution Adapter interfaces directly with the Deployment Manager to register or deregister resources. This process can be initiated either via the NebulOuS GUI for cloud resources or through the NebulOuS Resource Manager for edge devices.

Once resources are available, the NebulOuS Optimizer component evaluates all candidate nodes, across cloud and edge, to determine the most suitable setup. Based on this optimal configuration, the *Deployment Manager* triggers the deployment or reconfiguration of *Kubernetes* clusters and applications. During this process, key NebulOuS components are automatically instantiated within the target cluster.

Kubernetes [7][8] remains the backbone for container orchestration within each cluster, handling the local deployment and lifecycle management of containerized workloads. However, the infrastructure layer beneath Kubernetes, provisioning, scaling, and cluster formation, is fully managed by the Executionware stack.

In this deliverable, we present the advancements made in Executionware for managing multi-cloud and cloud-to-edge deployments, enabling NebulOuS to operate over dynamic, distributed, and heterogeneous infrastructure environments. The next section provides an in-depth description of the

Deployment Manager and Execution Adapter, detailing their architecture and implementation. We then present the NebulOuS Deployment Scenario, outlining the full lifecycle of resource and application management through dedicated REST API endpoints and execution sequences. This includes resource registration (both cloud and edge), candidate node discovery, cluster and application deployment, and reconfiguration operations. We also cover the role of NebulOuS deployment scripts in supporting k3s- and k8s-based clusters [7][8], resource deregistration, and mechanisms for persisting state across operations. Finally, we highlight the framework's automated testing capabilities and the fully automated deployment pipeline, which ensure reproducibility and robustness, key aspects of the support provided in the NebulOuS platform.

2.1 EXECUTIONWARE OVERVIEW

The Deployment Manager and Execution Adapter form the backbone of the NebulOuS Executionware layer, acting as the key enablers for translating optimized deployment plans into actual infrastructure provisioning and orchestration actions. These components ensure that application deployment in heterogeneous environments, spanning public clouds, private data centers, and edge devices, is both automated and robust.

The NebulOuS deployment and orchestration capabilities are built on Activeeon's ProActive Workflows and Scheduling technology. Specifically, the functionality of the Execution Adapter and Deployment Manager is realized through two core components of the ProActive stack:

- **The Execution Adapter** [4][5][6], implemented using ProActive's core engine and its IaaS Connector microservice.
- **The Deployment Manager** [2],[3], implemented via the Scheduling Abstraction Layer (SAL), a Java-based orchestration and resource coordination layer.

2.1.1 Execution Adapter (ProActive)

The Execution Adapter [4][5][6] *(see* Figure 1*)* is responsible for direct interaction with infrastructure APIs. It communicates with various cloud, on-premise, and edge providers to deploy, configure, and manage computing resources. This abstraction is enabled through the ProActive node model, which provides a unified representation of infrastructure resources—regardless of the underlying platform—enabling consistent management of heterogeneous environments.

At the heart of this functionality is the ProActive **IaaS Connector [9]**, a microservice that exposes RESTful endpoints for performing lifecycle operations on infrastructure resources. The IaaS Connector supports a broad spectrum of providers, including AWS EC2, Google Cloud, OpenStack, VMWare, and Docker, and is built on top of the Apache jclouds toolkit. The main models exposed by the IaaS Connector include:

Infrastructure: defines authentication and management endpoints.

Instance: represents machine configurations (e.g., image, hardware, network).

NodeCandidate: represents resource metadata (e.g., region, price, architecture).

These models are backed by a caching mechanism that optimizes performance across multi-provider environments. ProActive nodes are automatically deployed over the allocated compute resources using SSH-based installation of lightweight Java agents (`node.jar`), ensuring minimal overhead and high portability.

The Execution Adapter provides the admin interface for backend and development support, including:

- *Automation Dashboard*, which is used to monitor the execution of cluster deployments.
- *Workflow Studio*, which is used for testing the NebulOuS scripts.
- *Scheduler*, where all deployment jobs can be monitored.
- *Resource Manager*, visually presenting resources and their usage.

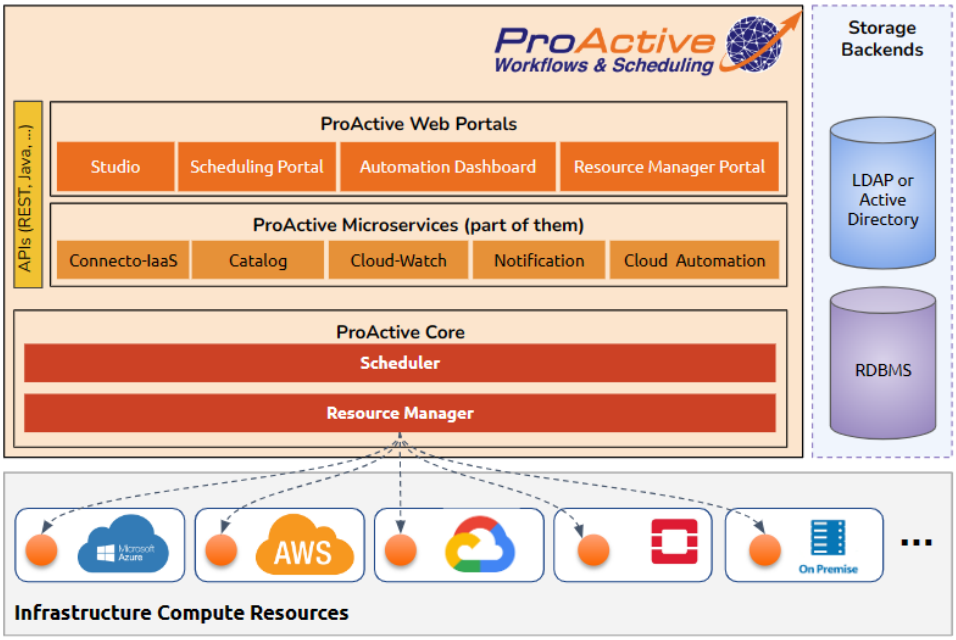

Figure 1. Execution Adapter Architecture

2.1.2 Deployment Manager (SAL)

The Deployment Manager [2][3], implemented using Activeeon's Scheduling Abstraction Layer (SAL), serves as the coordination layer that orchestrates the provisioning and cluster setup processes. It interacts with the Execution Adapter through the ProActive REST API and is responsible for:

- Triggering infrastructure resource retrieval according to user-defined constraints (CPU, RAM, region, etc.).
- Constructing deployment jobs and defining per-task node selection strategies.
- Managing node candidate caching and reuse.
- Initiating cluster setup and workload deployment.

The Deployment Manager exposes a comprehensive set of REST endpoints that define the lifecycle of a deployment, from registration of cloud providers to application orchestration. These endpoints are documented and integrated with the NebulOuS platform's GUI, enabling users to register cloud infrastructures and edge devices and trigger deployments seamlessly.

The Deployment Manager begins its operation by establishing a connection with the Execution Adapter through an authentication endpoint, acquiring a `sessionId` based on user credentials. Once connected, users can register new cloud providers using dedicated REST APIs and proceed to resource allocation and application deployment. The architecture of Executionware solution is presented in Figure 2.

In the following sections, we describe the main operations supporting the NebulOuS scenario using these endpoints.

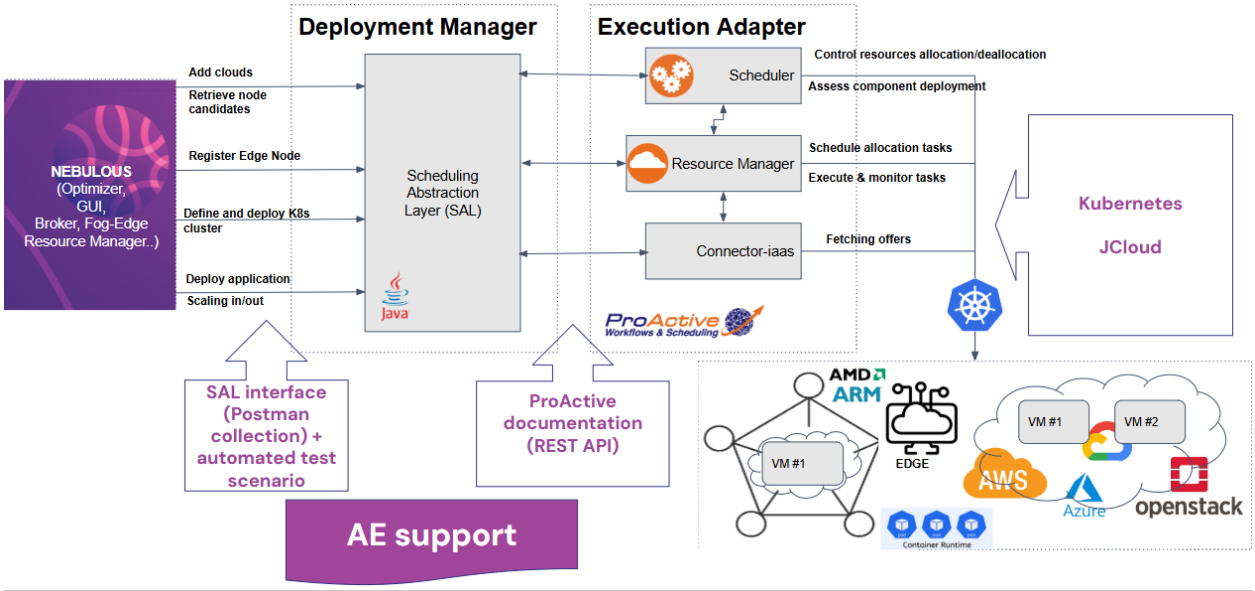

Figure 2. Executionware architecture

2.2 NEBULOUS DEPLOYMENT SCENARIO WITH EXECUTIONWARE

2.2.1 Cloud Resources registration and deregistration

Cloud registration within the NebulOuS platform is initiated through the `AddCloud` endpoint by NebulOuS GUI [10]. In this process, the upper layers of NebulOuS request the Deployment Manager to define a new cloud infrastructure by providing a unique cloud name. This name must not match any existing registration, as conflicts can cause failures during configuration updates or leave resources orphaned.

To ensure a clean environment, proper deregistration must be performed using the `RemoveClouds` endpoint. Correct deregistration guarantees that all resources are released on the provider side, avoiding additional costs from lingering virtual machines (VMs) and ensuring internal states are cleared.

After a cloud is successfully registered, the Deployment Manager automatically triggers an asynchronous discovery process to retrieve available cloud images and node candidates. The platform monitors this operation through the `isAnyAsyncNodeCandidatesProcessesInProgress` endpoint, polling until completion. Once the discovery phase ends, the `GetCloudImages` endpoint is used to validate the cloud authentication and fetch the available images.

If cloud credentials are misconfigured or insufficient, errors will occur during image retrieval, with additional details accessible via Execution Adapter logs. It is important to note that SSH credentials, though required later during cluster deployment, are not involved in this initial registration flow. This discovery phase ensures that cloud infrastructures are accurately validated and ready for subsequent orchestration operations within the NebulOuS ecosystem.

Integration with cloud providers is handled as follows:

- Google Compute Engine (GCE) [11], Amazon Web Services (AWS EC2) [12], and OpenStack [13] are integrated through JClouds [14],
- Microsoft Azure [15] is integrated directly through the Azure API [16].

Because of differences across providers, specific fields must be populated during cloud registration. The required fields for each cloud provider are summarized in Table 1.

Table 1. Overview of the information needed to register Cloud 2Providers

SAL Field Name	GCE	Azure	AWS	Open Stack
cloudId	{{cloud_name}}			
	Cloud ID placeholder to identify the cloud configuration. String value need to represent valid cloud name.			
cloudProviderName	*"google-compute-engine"*	*"azure"*	*"aws-ec2"*	*"openstack "*
	Cloud provider identifier			
cloudType	*"PUBLIC"*	*"PUBLIC"*	*"PUBLIC"*	*"PRIVATE"*
securityGroup	null	null	*{{aws-securityGroup}}*	*{{os-securityGroup}}*
			Name of the AWS EC2 Security Group to apply to the instances.	Name of the OpenStack security group assigned to VMs.
sshCredentials.username	*"ubuntu"*			
	Defines the SSH username for connecting to Ubuntu 22.04 VM instances.			
sshCredentials.keyPairName	null	null	*{{aws-keypair}}*	*{{os-keypair}}*
			AWS EC2 key pair name used to access the instances via SSH.	OpenStack key pair name used to access the instances via SSH.
sshCredentials.publicKey	*{{gce-publickey}}*	*{{azure-publickey}}*	null	null
	SSH public key to grant user access to created instances.	Public SSH key used for authentication into Azure VMs.		
sshCredentials.privateKey	*{{gce-privatekey}}*	*{{azure-password}}*	*{{aws-privatekey}}*	null
	SSH private key used to connect to instances.	Private key or password to connect via SSH to Azure VMs.	Private key to connect to the AWS EC2 instances.	
endpoint	null	null	null	*{{os-auth_url}}*
				Authentication URL for OpenStack identity service (Keystone).
scope.prefix	null	null	null	*"project"*
				OpenStack scope prefix (commonly

				project for project-scoped tokens).
scope.value	null	null	null	*{{os-projectName}}* OpenStack project name for scoping the authentication.
identityVersion	null	null	null	*{{os-identity-api-version}}* Identity API version used (usually "3").
defaultNetwork	null	null	null	*{{os-defaultNetwork}}* OpenStack default network name or ID for the VM instances.
credentials. user	*{{gce-user}}* GCP service account user email used for authentication.	*{{azure-user}}* Username or client ID of the Azure service principal.	*{{aws-user}}* AWS access key ID (part of authentication credentials).	*{{os-user}}* OpenStack username for authentication.
credentials. secret	*{{gce-secret}}* Secret content of the GCP service account key (JSON).	*{{azure-secret}}* Password or secret key for the Azure service principal.	*{{aws-secret}}* AWS secret access key (part of authentication credentials).	*{{os-secret}}* OpenStack user's password or secret for authentication.
credentials. domain	null	*{{azure-domain}}* Active Directory Tenant ID.	null	*{{os-domain}}* OpenStack domain name (used in Keystone v3).
credentials. subscriptionId	null	*{{azure-subscription_id}}* subscription ID under which resources are managed.	null	null
credentials. projectId	*{{gce-project-id}}* Project ID where instances are created	null	null	null

While the registration procedure through SAL is standardized, each cloud provider introduces operational and technical particularities that must be considered for correct resource discovery and orchestration.

For instance, Google Compute Engine (GCE) [11] organizes resources around zones rather than regions, as seen with AWS or Azure. GCE's lack of strict regional partitioning during node candidate retrieval can significantly increase synchronization time, especially if a project contains numerous images or instances across zones. To maintain efficiency, it is important that only relevant and active resources are kept within the project.

Similarly, OpenStack [13] deployments typically do not use region-based partitioning unless explicitly configured by the administrator. Consequently, image and node discovery can be slower in OpenStack environments. Additionally, OpenStack platforms are often customized, resulting in variations in API behavior and available metadata depending on the specific distribution (e.g., vanilla OpenStack, OpenTelekomCloud, CityCloud).

By contrast, Amazon Web Services (AWS) [12] structures resources explicitly within regions, leading to faster and more scoped discovery operations. However, AWS instances must be tagged with a specific label (by default, `proactive=true`) to be visible to NebulOuS. Without this label, virtual machines will not be considered eligible candidates, which ensures that only intentionally provisioned resources are exposed to orchestration workflows.

Azure [15] also relies on strict regionalization but differs in its credential and authentication handling. Azure requires a service principal (Application ID and Secret) associated with an Active Directory tenant, alongside the subscription ID. For VM access, Azure supports both SSH keys and password-based authentication, unlike most other providers that mainly rely on SSH key pairs.

Across all cloud providers, differences in metadata exposure and API capabilities exist. Some providers offer detailed specifications and rich metadata natively, while others return minimal information, necessitating additional normalization during the discovery phase. To address these inconsistencies, NebulOuS implements *cloud-specific handlers* to standardize the resource information model.

The cloud registration process within NebulOuS, orchestrated through the Deployment Manager, establishes a structured mechanism for integrating cloud infrastructures across multiple providers. Despite following a standardized workflow for cloud addition, asynchronous resource discovery, and image validation, each provider introduces specific challenges related to credential management, regionalization, metadata retrieval, and API behavior. Understanding these variations and properly managing provider-specific requirements—such as AWS instance tagging, OpenStack's customizable environments, GCE's zoning structure, and Azure's service principal authentication is crucial to achieving seamless and efficient multi-cloud orchestration within the NebulOuS ecosystem.

2.2.2 Edge Resources registration and deregistration

Edge device management in NebulOuS plays a crucial role in enabling seamless application deployment across the cloud-edge continuum. The Executionware provides the necessary interfaces and automation to register and deregister edge nodes, allowing edge resources, spanning AMD and ARM architectures [17][18][19], to be brought into the orchestration pipeline as candidates for optimized workload deployment.

The registration process is initiated through a dedicated REST API by NebulOuS Resource Manager [20], enabling users to onboard new edge nodes via the `RegisterNewEdgeNode` endpoint. This action results in the assignment of a unique Device Identifier and a Candidate Node ID, which are critical for subsequent deployment and orchestration processes.

While the `RegisterNewEdgeNode` endpoint captures essential metadata and integrates the device into the NebulOuS system, validation of edge node availability occurs later, during the actual cluster deployment phase. The Deployment Manager ensures that the registration information accurately persisted and made accessible for cluster planning and execution.

The available list of registered edge devices can be retrieved using the `GetEdgeNodes` endpoint, offering transparency and control over accessible resources. Edge devices can be deregistered using the `DeleteEdgeNode` endpoint, ensuring clean removal and maintaining internal system consistency.

When registering a new edge device, the `RegisterNewEdgeNode` endpoint expects a complete EdgeDefinition object to be provided to the Deployment Manager. This object encapsulates all relevant metadata about the edge resource, including:

- Network accessibility (public and private IPs)
- Authentication credentials (SSH username and password or key)
- Hardware specifications (CPU, memory, disk, GPU/FPGA availability)
- Operating system details
- Geolocation (latitude and longitude)
- Architecture type (e.g., AMD64, ARMv7, ARMv8)

This information enables the Execution Adapter to accurately instantiate a corresponding Node Candidate in the platform, which serves as a logical representation of the physical device. To ensure compatibility across diverse hardware types, the Execution Adapter leverages architecture-specific execution agents, self-contained `.jar` files, that are downloaded and executed on the edge device during registration.

These `node.jar` files are tailored to the device's architecture (AMDx64 [16], ARMv7 [18], or ARMv8 [19]) and are responsible for establishing communication between the physical edge node and the ProActive Resource Manager. Agents are accessible through the portal, and the appropriate `jarURL` must be used during registration. Successful execution of the agent finalizes the registration process and confirms the device's readiness for deployment.

During registration, a corresponding Node Candidate is added to the orchestration system. During deregistration, the Node Candidate is removed, ensuring that only active devices are considered for workload deployment. This tight coupling guarantees system cleanliness, robustness, and dynamism across the NebulOuS orchestration environment.

2.2.3 Filtering Node Candidates

To support intelligent and adaptive deployment decisions, the Deployment Manager provides the `findNodeCandidates` endpoint for NebulOuS Optimizer [21] to query the available pool of resources. This filtering mechanism is based on placement requirements and constraints defined either by the application developer or inferred from optimization strategies. The results are node candidates, representations of physical or virtual machines, that meet the specified conditions and are eligible for deployment.

Filtering is done by composing a list of requirements, which may include node type (IAAS for cloud resources or EDGE for physical edge devices), cloud provider, price, operating system, location, and specific hardware capabilities such as memory, cores, GPU, or FPGA availability. For example, a deployment targeting a cloud region in Bergen (bgo) may require IAAS-type nodes with Ubuntu 22, 8GB RAM, and 4 CPU cores. These criteria are structured as typed attributes (e.g., `AttributeRequirement`, `NodeTypeRequirement`) and passed as filters to the Deployment Manager. The resulting node candidates are then referenced by their unique IDs for scheduling and resource reservation.

To accommodate cloud-specific constraints and limitations, Executionware includes predefined, provider-specific queries hardcoded within the search logic. This is particularly important because not all metadata is retrievable through public APIs across cloud vendors. For instance, when dealing with AWS, operating system information is not exposed for instances. As a workaround, the required OS version (e.g., Ubuntu 22.04) must be explicitly defined when creating cloud configurations on the

provider's side. Similarly, hardware attributes such as GPU or FPGA availability are calculated using the latest provider data, rather than being directly queried.

Azure integration adds an additional layer of complexity. Successful deployment has been achieved on a single Azure instance, F4s standard, that utilizes a third-party Ubuntu image. Other potential Azure instances have been identified during discovery, but many are incompatible with standard Execution Adapter deployment requirements. On another hand, during testing deployment with different instances of other cloud providers, some instances are found also not to be supported, often duet to limitations imposed by JCloud adapter. These unsuitable instances are captured and registered in the Optimizer's exclusion list to prevent faulty deployments. This ensures that only validated and reliable instances are considered in future filtering operations.

For edge devices, node candidates are registered during the initial registration process. If a deployment needs to target a specific device, NebulOuS can retain the node candidate ID at registration time or define a unique name identifier for the edge device based on their owner, that can later be searched using a `hardware.name` attribute filter.

Together, this flexible filtering mechanism ensures that deployments are both context-aware and infrastructure-compliant, balancing performance, availability, and compatibility constraints across heterogeneous environments.

2.2.4 Deploying the Cluster and Application

The deployment of Kubernetes clusters in NebulOuS is orchestrated through a clear and flexible workflow, offering the NebulOuS platform full control over configuration and monitoring. The process begins with the `DefineCluster` endpoint, which allows the Optimizer to define a cluster by specifying its name, the associated node candidates, environmental variables, and the type of cluster, either `k8s` (Kubernetes) [7] or `k3s` (lightweight Kubernetes) [8], with `k8s` set as the default. This setup is supported by predefined deployment script templates [22], which can be customized for specific installation needs, including integration with network components on public cloud environments. Environmental variables can also be passed at this stage to guide the installation of application components or to configure service-specific behavior.

Once clusters are defined, the `DeployCluster` endpoint is used to execute the deployment. This endpoint transforms the cluster definition into an Execution Adapter workflow (see Figure 2) and launches it on the target infrastructure, whether cloud or edge. As a result, all setup actions, from node provisioning to Kubernetes bootstrapping, are orchestrated seamlessly.

Cluster deployment is monitored through Execution Adapter interfaces available to NebulOuS developers and use case partners during the development and integration phases. These admin users can test and adjust deployment scripts using the Workflow Studio, which provides a detailed and editable view of the deployment logic (see Figure 3). Workflow scripts [22], namely *PRE_INSTALL_SCRIPT.sh* , *INSTALL_SCRIPT.sh* , *POST_INSTALL_SCRIPT.sh*, *STOP_SCRIPT.sh*, *UPDATE_SCRIPT.sh* are integrated as part of the task Implementation of the task **D**. Script *START_SCRIPT.sh*, is integrated as part of the task Implementation of the task **E**. Tasks A, B, C, F, G are ProActive-defined tasks handling internal mechanisms.

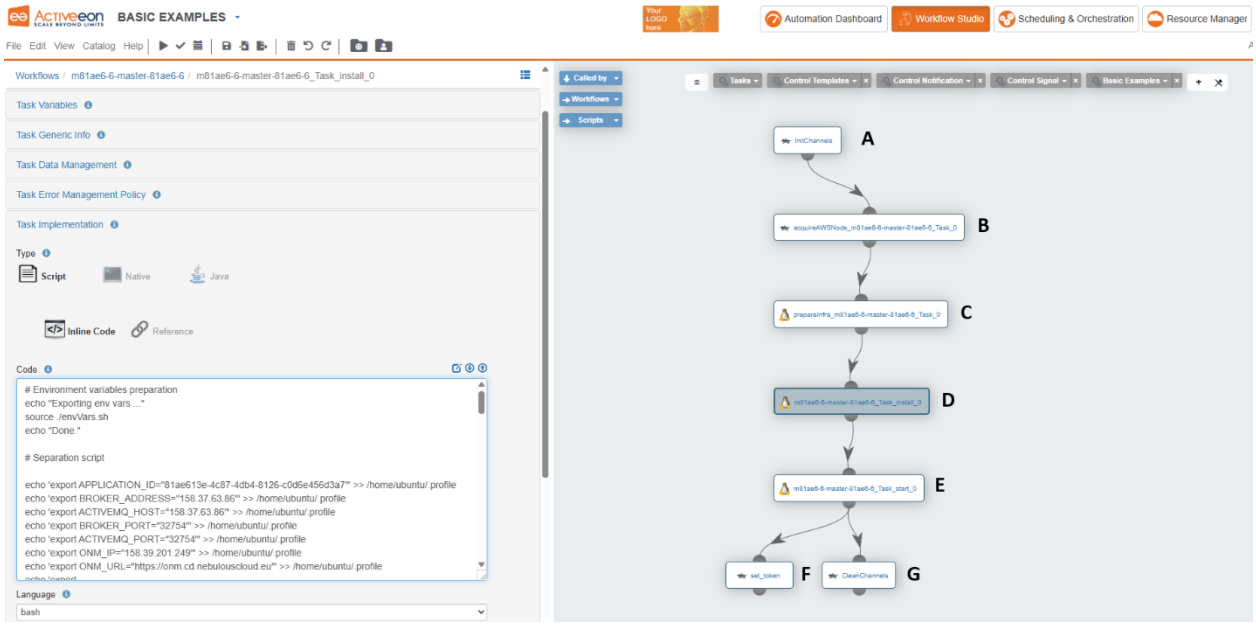

Figure 3. Generated deployment workflow for master node in Execution Adapter workflow studio

Throughout deployment, the cluster's status can be monitored automatically in real time using the `GetCluster` endpoint or manually by leveraging tools like the Execution Adapter Dashboard, Scheduler, and Resource Manager (see Figure 4.). These tools give visibility into individual node states, resource consumption, and task execution, helping users identify any bottlenecks or failures. If the deployment fails, in addition to the output in Execution Adapter, logs from the Deployment Manager and iaas-connector components can be inspected to troubleshoot issues such as missing credentials or unavailable nodes. It's important to ensure node availability and valid SSH credentials as these are validated only at execution time.

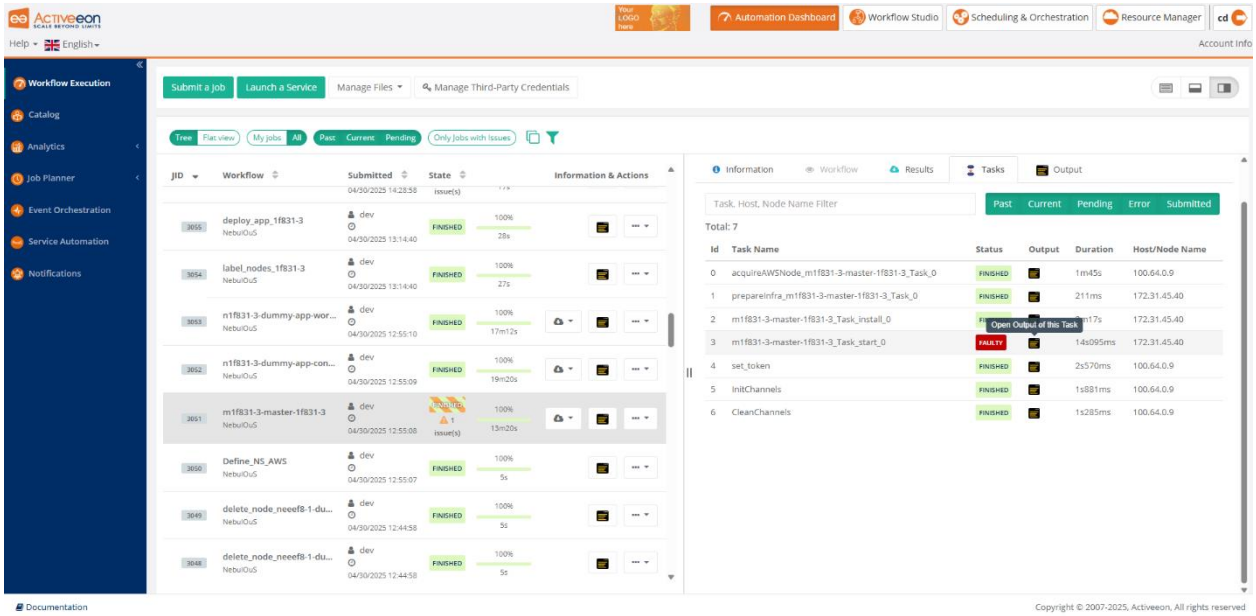

Figure 4. Execution Adapter Dashboard view of the workflow execution

Once a cluster is no longer needed, users can fully dismantle it with the `DeleteCluster` endpoint. This step undeploys all nodes, shutting down provisioned machines on cloud provider side, and

removes the cluster definition from the system, preventing unnecessary resource usage. This final action is especially useful during iterative testing or when clusters are meant for temporary demonstration or experimentation purposes.

Altogether, NebulOuS offers a modular and developer-friendly environment for deploying Kubernetes clusters across heterogeneous infrastructures. By combining flexible definition mechanisms, script-based customization, and comprehensive monitoring tools, the Deployment Manager deployment process ensures both adaptability and traceability for complex distributed systems.

Once a Kubernetes cluster has been successfully deployed, users can seamlessly proceed to application deployment through the `ManageApplication` endpoint. This endpoint allows developers and use case partners to define application configurations using a YAML-based descriptor (`appFile`) and deploy them using tools such as KubeVela [23]. The deployment definition includes specifications like container images, CPU/memory allocations, environment variables, scaling traits, and deployment workflows. With a simple POST request, the Deployment Manager engine wraps the application definition into an Execution Adapter job and submits it to the targeted cluster, returning a unique job ID for status tracking.

The deployment is fully observable through the Execution Adapter Dashboard, where the submitted job can be monitored in real time using the returned Job ID via the `getJobState` endpoint. This ensures full traceability from cluster setup to application execution. Moreover, since the cluster nodes are labelled during this process (based on the `app_component_name` in the YAML), resource targeting and application placement within the cluster are automatically handled. This approach provides a clean and declarative way to manage applications post-deployment, combining infrastructure-level automation with fine-grained control over service behavior, making the entire DevOps loop transparent and manageable through the Deployment Manager framework.

2.2.5 Cluster Reconfiguration

Cluster reconfiguration in the Deployment Manager enables dynamic and responsive management of Kubernetes clusters and deployed applications. These operations support both scaling out and scaling in the cluster and are orchestrated through dedicated Deployment Manager endpoints. The process fully leverages Kubernetes-native capabilities, such as label-based scheduling, declarative replica management, and node provisioning, while abstracting their complexity within Execution Adapter workflows.

When scaling out an application, the first step is to dynamically expand the Kubernetes cluster using the `ScaleOut` endpoint. This adds new worker nodes based on existing node templates, increasing the overall processing capacity of the cluster. Once deployed, these new nodes are labeled using the `LabelNode` endpoint, which marks them as eligible for scheduling application workloads. Labels follow a structured key-value format and allow fine-grained control over which nodes are targeted by specific application components. Finally, the `ManageApplication` endpoint is used to increase the number of application replicas. This ensures that the application takes full advantage of the newly added resources by deploying instances on the freshly labeled nodes.

Conversely, when scaling in an application, the process begins with updating the labels on specific nodes to indicate that they should no longer be targeted for new application workloads. This label change, performed via the `LabelNode` endpoint, prepares the nodes for safe removal. The application replicas are then reduced accordingly through the `ManageApplication` endpoint, which ensures

that workloads are drained from the targeted nodes. Once the nodes are no longer hosting active application components, the `ScaleIn` endpoint is invoked to remove them from the cluster entirely.

Throughout this process, the Deployment Manager ensures compatibility with Kubernetes conventions and exploits its built-in features to manage workload distribution and node lifecycle. The orchestration of these operations is handled through Execution Adapter workflows, which enable automation, logging, and safe execution. As a result, platform benefit from a high-level, cloud-agnostic interface for cluster and application reconfiguration, minimizing manual intervention while maximizing control and scalability. This makes the Deployment Manager particularly well-suited for managing dynamic, multi-cloud, or edge environments where resources and workloads are in constant flux.

2.3 EXECUTIONWARE DEPLOYMENT ARCHITECTURE AND LIFECYCLE MANAGEMENT

In the NebulOuS platform, the Execution Adapter and the Deployment Manager are automatically deployed as Kubernetes pods to simplify installation, maintenance, and scaling.

2.3.1 Fully Automated NebulOuS Deployment Pipeline

During the initial development phase of NebulOuS, the Execution Adapter was deployed as a separate server installation. This Linux-based setup, installed on an Ubuntu 22.04 server, served as a shared execution backend for multiple NebulOuS sandbox environments. Deployment Manager was deployed in this environment as a Kubernetes pod, establishing connection to the Execution Adapter from each instance. However, following feedback from Open Call participants, who found the standalone server setup too complex, the final NebulOuS release shifted to a dynamic Kubernetes-based deployment of Execution Adapter which is to be provided with a second release of NebulOuS platform.

In this improved deployment, Execution Adapter is installed as a set of Kubernetes pods (see Figure 5). Ready-to-use Kubernetes configuration was provided, allowing automated deployment on on-premises Kubernetes clusters. The key components deployed as pods include:

- (Optional) PostgreSQL database pod [24]
- (Optional) Static nodes pods [25]
- The ProActive server pod [26]
- (Optional) Static nodes pods [27]
- (Optional) Dynamic nodes pods [27]
- (Optional) PostgreSQL database pod [27]

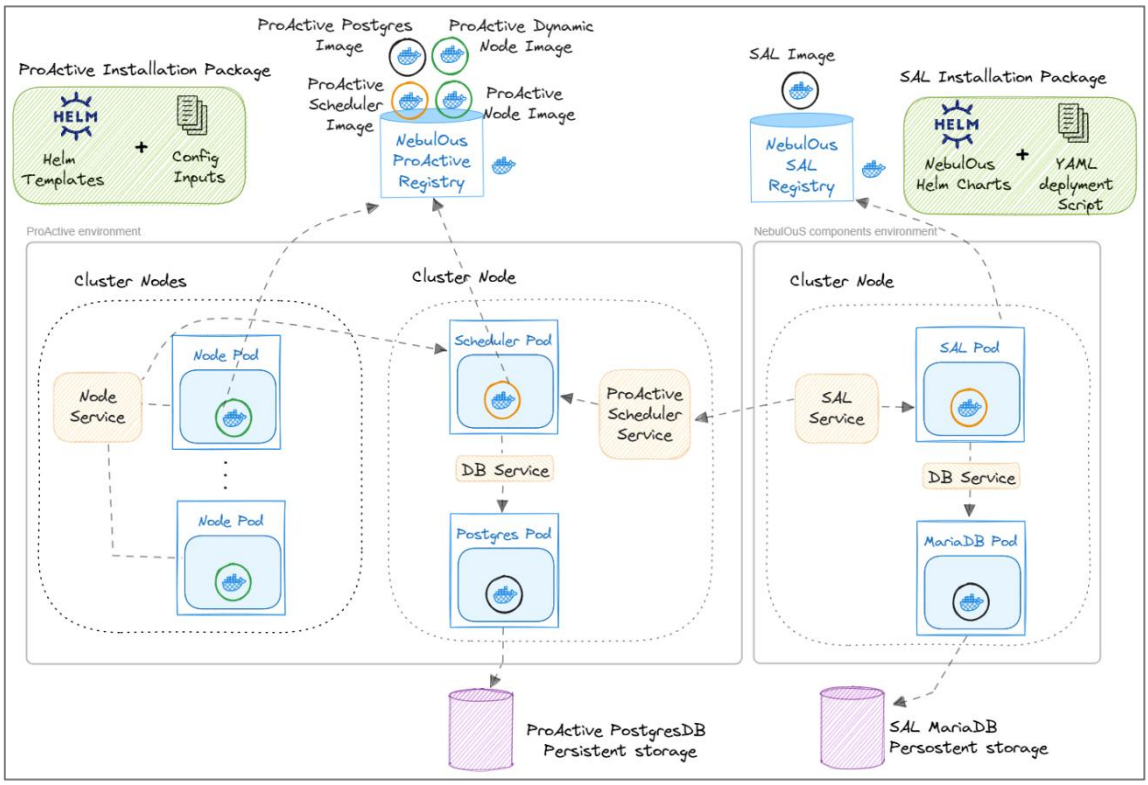

Figure 5. Execution Adapter (ProActive) and Deployment Manager (SAL) deployment in NebulOuS Kubernetes

The installation is managed via Helm [28], a Kubernetes package manager. A dedicated Helm chart is configured using a `values.yaml` file where users specify deployment parameters, such as cluster type, image repositories, credentials for the private Docker Registry, resource limits, and persistence volumes. The images were moved regularly for the testing purposes to the dedicated Nebulous repositories [24][25][26][27] To ensure proper operation, particularly on-premises, persistent volumes are configured for the server and database pods, with storage paths predefined on Kubernetes nodes.

To upgrade ProActive, users simply update the `values.yaml` with the new image versions and run a Helm upgrade command, with the pods automatically recreated to reflect the new release. This Kubernetes-based dynamic deployment simplifies setup and maintenance, eliminates manual server provisioning, and aligns with cloud-native practices expected by users of the NebulOuS platform.

The Deployment Manager is packaged as a Docker image. During the development phase the image was automatically published in the public ActiveEon repository [29] at each new release. The Nebulous CI/CD pipeline detects new date-tagged images and automatically upgrades the Nebulous sandbox environment by deploying the updated SAL pod. Developers can also manually deploy the latest Deployment Manager image to a Kubernetes cluster by applying the provided `sal.yaml` manifest [2], which defines the deployment, service exposure, environment variables, and persistent volumes necessary for operation. For releases and well tested SAL version, corresponding SAL image was moved to SAL Nebulous repo [30].

All runtime configuration for Deployment Manager, including the connection to the Execution Adapter, is dynamically injected into the Kubernetes deployment via environment variables defined in Nebulous Helm Charts [31]. This setup enables secure, flexible connectivity without hardcoding endpoints, credentials, or environment-specific parameters into the application image or deployment

manifests. Developers can easily update connection settings or override them temporarily by editing the environment variable configuration in the deployment.

Developers have full control over Deployment Manager lifecycle operations in Kubernetes. They can restart the Deployment Manager service via a rolling deployment update (`kubectl rollout restart`), manually delete and recreate pods, or scale the deployment to zero and back up for a complete reset. Logs are accessible both at the Kubernetes pod level and inside the container's filesystem, providing visibility into scheduling activity, runtime errors, and internal processing states. Kubernetes' built-in tools (`kubectl top pods`, `kubectl describe pod`) also allow developers to monitor resource utilization, pod health, and diagnose deployment issues.

The Deployment Manager deployment exposes a dedicated debugging service, configurable through `sal.yaml`. Developers can use Kubernetes port-forwarding to map the internal debugging port to their local machine, enabling remote JVM debugging. This allows setting breakpoints, inspecting variables, and tracing execution paths live inside the running Deployment Manager instance. In parallel, developers are encouraged to monitor the Deployment Manager runtime logs for additional context during debugging sessions.

2.3.2 Persistence and Cleanup Mechanisms

The Deployment Manager includes a set of dedicated endpoints for managing persistence and ensuring the integrity of its internal state across deployments, particularly during development and maintenance activities. These persistence operations are primarily intended for NebulOuS developers and project mentors who need to validate that resources such as clusters, clouds, and edge devices have been properly undeployed and removed from both the Deployment Manager and the underlying Execution Adapter server or cloud providers.

To support this, the Deployment Manager exposes several endpoints [3]: `CleanAll`, `CleanAllClusters`, `CleanAllClouds`, `CleanAllEdges`, and `CleanSALDatabase`. Each of these targets specific resource types, and collectively they serve to eliminate inconsistencies between the Deployment Manager's internal state and the actual infrastructure. The most comprehensive of these is the `CleanAll` endpoint, which performs a full system cleanup, removing all registered resources from the Deployment Manager, the Execution Adapter server, and any external environments to which the Deployment Manager has deployed nodes. This is particularly useful after testing scenarios or when resetting a development environment.

The Deployment Manager database itself is backed by a Persistent Volume (see Figure 5), ensuring that data persists across Deployment Manager restarts. However, there are cases where this persistence can lead to issues, particularly when the database schema evolves or an unrecoverable error occurs, such as a Hibernate exception or SQL-level corruption. In such scenarios, a restart of the PVC may be necessary to reset the persistence layer.

An important nuance is that the Deployment Manager's persistence management also intersects with the Execution Adapter's own execution and resource metadata. For instance, the Deployment Manager maintains its state independently, but resources like cloud providers and cluster nodes also have corresponding entries within the Execution Adapter server. If only the Deployment Manager database is wiped, without invoking full cleanup operations, those Execution Adapter-side records remain intact. This can result in misleading and potentially hazardous behaviour. For example, if a cloud provider was previously added via the Deployment Manager, but then the Deployment Manager's database is reset without cleaning the Execution Adapter state, re-adding a cloud with the same name may connect to the original Execution Adapter resource, regardless of whether the new

configuration differs. This leads to unpredictable behaviour and may result in applications being deployed to an unintended environment.

Similar complications may arise with clusters. If the Deployment Manager attempts to deploy a cluster node but fails, due to cloud connectivity issues, for example, the node might be partially registered or misrepresented in the Execution Adapter. If not properly removed, subsequent operations involving that cluster could fail silently or produce errors that are difficult to trace. These challenges illustrate the importance of using the resource deregistration endpoints consistently, especially during testing or error recovery workflows.

In sum, the Deployment Manager's persistence operations are designed not only to support long-term state retention, but also to ensure that environments can be safely and consistently cleaned up. Proper use of these endpoints prevents state divergence between the Deployment Manager and Execution Adapter, ensures predictable deployment behaviour, and supports the broader system goal of secure and fault-tolerant orchestration across distributed cloud and edge environments.

2.3.3 Automated Testing Support

The Deployment Manager in the NebulOuS platform is supported by an automated testing workflow [31] designed to validate its behaviour under a realistic, repeatable deployment scenario. These tests are implemented using Postman collections [32], which differ from traditional API test sequences by incorporating scripted logic through pre- and post-request scripts. These scripts are used not only to validate the response of each API call but also to dynamically control the flow of the test based on the results received, enabling fully automated execution without manual intervention.

To execute these automated tests from the command line, the NebulOuS platform relies on Newman [33], a CLI utility specifically designed to run Postman collections. This allows the test suite to be integrated into local development environments or automated CI/CD pipelines. Newman requires a JavaScript execution environment and thus depends on the installation of Node.js [34] and the Node Package Manager (npm) [35]. These tools can be installed from the official website, after which developers can install Newman globally via npm. Once installed, Newman is accessible from the terminal and can be used to execute test runs against a deployed Deployment Manager instance.

At present, the test suite focuses on a single-cloud deployment for each cloud, a multi-cloud and cloud-to-edge scenario, covering the full lifecycle of resource and cluster operations in a controlled environment. The workflow begins with the registration of a resources, followed by the discovery of node candidates based on specified criteria. A cluster is then defined and deployed alongside an application using one replica. The cluster is subsequently scaled to accommodate two application replicas, then scaled back down to a single replica, simulating realistic elasticity scenarios. After these operations, the application is removed, the cluster deleted, and the cloud provider deregistered, completing the full deployment lifecycle and cleanup phase.

Before executing the test, users must edit the provided postman collection for testing [32]. This file includes all necessary API calls and the logic that drives the test. In particular, users need to ensure that the `AddCloud` or RegisterNewEdgeNode request contains the appropriate cloud parameters, and that the `FindNodeCandidates` calls are updated with any necessary filters for master and workers node selection. The `DefineCluster` call includes a set of pre-defined environment variables that must be correctly configured to reflect the target NebulOuS test environment. These include parameters such as application ID, message broker configuration, and AMPL license keys, all injected dynamically via the associated Postman environment file.

The Postman environment file serves as the context for test execution and contains the actual values for environment variables referenced throughout the test collection. Testers are advised to update this file to match their specific test environment, including credentials and host details. To avoid conflicts during collaborative testing, it is recommended to rename all variables to user-specific values, particularly when multiple testers operate on shared infrastructure.

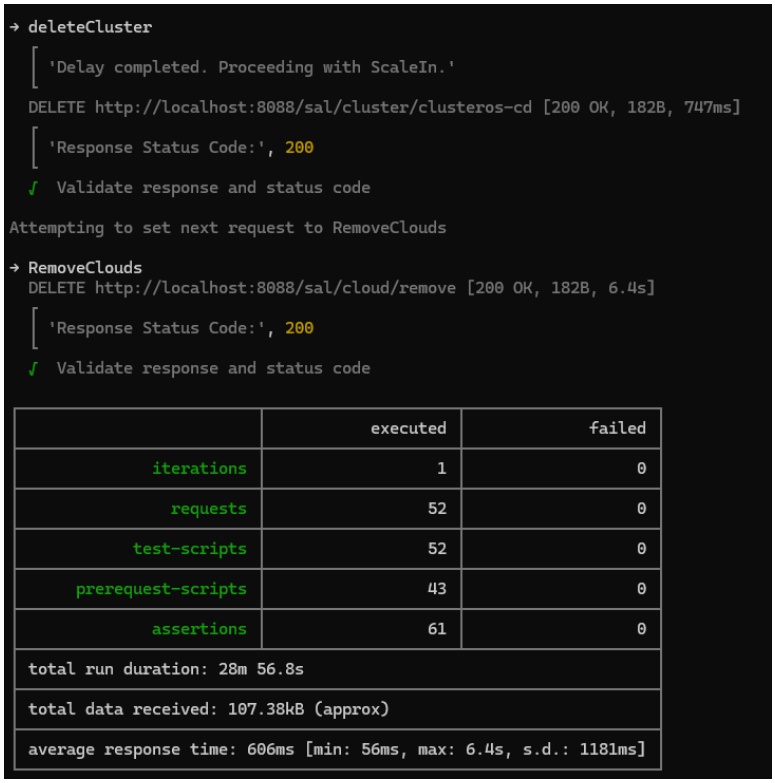

Figure 6. Results and interface of automated test successful execution

Once the configuration is complete, the test is executed by first establishing a connection to the Deployment Manager, typically through a local port-forward setup. Newman is then used to run the test collection with the selected environment configuration. Developers are free to rename the test files to suit their workflows, if the names are correctly referenced in the Newman execution command. This automated testing setup offers a reliable, repeatable method to validate Deployment Manager behaviour under real-world usage patterns and plays an essential role in maintaining the robustness of the NebulOuS platform as it evolves.

3 FROM SLA TO SMART CONTRACTS

3.1 BACKGROUND

Blockchain is a distributed ledger, which consists of a series of chronologically ordered blocks that are appended to the ledger and connected with each other in a linked-list data-structure. To provide integrity and immutability of data in the ledger, the blockchain prevents any updates in the committed blocks. To ensure this, each block contains the hash of the previous block, and the ledger is replicated across peers that participate in the network. A block usually contains a set of timestamped

transactions that are bundled together. To ensure adequate security, blockchain systems adopt various cryptographic primitives such as hashing algorithms, digital signatures, and Public key infrastructure (PKI) protocols. In the blockchain systems, there are two key types of participants: those that generate the transactions, and those that validate and store them in the ledger.

A blockchain network runs on a peer-to-peer topology where each node is expected to store the same copy of the ledger. The network consists of a set of nodes or organizations that do not have a preexisting trust relationship among them. Therefore, to ensure that each peer node has the same copy of the ledger at any given time, the new valid block that will be appended in the ledger is selected by executing a consensus mechanism. In particular, a consensus mechanism (e.g., *Proof-of-Work (PoW)*, *Proof-of-Stake (PoS)*, or *Practical Byzantine Fault Tolerance*) is a protocol that ensures synchronization among all network peers (i.e., nodes that maintain the ledger and might also process the transactions) about the transactions that are valid and that are about to be added to the blockchain [36][37].

Therefore, the mentioned consensus mechanisms are pivotal for the correct functioning of blockchain and need to be tested properly before their use in real-world applications. The key components and functionalities of a blockchain system enable some unique features including immutability, decentralization, consensus, provenance, and finality, which makes it a promising solution in many application domains [38][39][40][41][42].

Typically, blockchains are categorized based on their permission model. Based on this categorization, blockchain can either be permissionless or permissioned and can also be divided into public, private or consortium ledgers. In relevant literature, public and permissionless blockchains are considered equivalent and used interchangeably. However, these two categories are concerned with different authentication and authorization mechanisms.

A permissionless blockchain (e.g., Bitcoin[1] and Ethereum[2]) allows anyone to become a participant and perform activities such as taking part in a consensus mechanism, sending new transactions throughout the network, and maintaining the ledger state. In a permissioned blockchain (e.g., Hyperledger Fabric[3]), on the other hand, the participation is constrained, and only the pre-verified parties with an established identity are allowed to join the network. Permissioned blockchains require a minimum level of trust among the participants of the consortium and hence, nodes need identities and mutual authentication to participate in the network.

Different types of blockchains do not inherently offer advantages over one another. Each type can be optimized for performance and usability based on the specific requirements and the implementation environment.
required number of transactions per second, transaction commit latency, and service availability [43]. There are other benefits and drawbacks for each of the blockchain types, apart from the trust among the participants, e.g., scalability [44], security and privacy [45], and degree of decentralization. These should be taken into account when making a choice to utilize the efficient blockchain platform.

[1] Bitcoin is a decentralized digital currency and the first implementation of a permissionless blockchain. It allows anyone to participate in the network, validate transactions, and maintain the ledger through a consensus mechanism known as Proof of Work.
[2] Ethereum is a permissionless blockchain platform that supports more than just a cryptocurrency. It enables the creation and execution of smart contracts—self-executing agreements with terms written in code.
[3] Hyperledger Fabric is a permissioned blockchain framework designed for enterprise use. It restricts participation to pre-verified entities, allowing only trusted parties to join the network, validate transactions, and access the ledger.

3.1.1 Smart Contracts

The term *Smart Contract (SC)* was coined in 1990s by cryptographer Nick Szabo [46]. He defined SC as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises". However, practical applications of SCs did not emerge until the evolution of distributed ledger technologies (DLTs) such as Bitcoin and Ethereum, in which the immutable and distributed nature of the blockchain and consensus protocols made it feasible to implement SCs. Generally speaking, a SC can be seen as a computer program that digitally allows the verification and enforcement of contracts between parties in a blockchain system.

Typically, SCs are deployed on and protected by blockchain, and they possess certain unique characteristics and provide a number of advantages. First, since the SCs are deployed and verified on blockchain ledger, the code implementing the SCs is immutable due to the tamper-resistant feature of blockchain. Second, the execution of SCs is done by consensus nodes without mutual trust in a decentralized manner. Third, an SC enables automation of tasks. For instance, it could automatically initiate a transfer of digital assets between involved parties when certain predefined conditions specified in the contract are met or a trigger is sent via a transaction.

Bitcoin was the first cryptocurrency to facilitate the use of SC for sending and receiving bitcoins via a simple scripting language. However, Bitcoin's scripting language has limitations concerning the logical, arithmetic, and cryptographic operations that it supports, which are not suitable for expressing complex business logic.

Ethereum became the first public blockchain platform that supports SCs with advanced and customized logic by using its Turing-complete *Ethereum Virtual Machine*. In Ethereum, SCs can be seen as accounts which are controlled by program code, unlike the user accounts which are controlled by user's private key. Both contract and user accounts can hold and send/receive Ether. Ethereum supports development of SCs in several high-level languages such as Solidity and Serpent, and regardless of the language, the SC code is compiled to create the corresponding Ethereum virtual machine bytecode which is then deployed for execution on the underlying blockchain. The blockchain along with the SCs provides a suitable platform for the design of various types of *Decentralized Applications*, e.g., games, gambling, supply chain management, voting, and crowdfunding.

Since the SCs usage are at early stages, and these contacts deal with the asset management and transfer, they are a promising target for cybercriminals. Hence, advanced techniques and tools are required to ensure that the SCs are tested for various security vulnerabilities before their deployment on a blockchain platform. Apart from Ethereum, there are many open-source popular blockchain platforms such as Hyperledger-fabric and Corda[4], that support the execution of complex SCs and facilitate the creation of decentralized applications.

3.1.2 SLA to Smart Contract Transformation

The Smart Contract Encapsulator (SCE) plays a central role in the SLA lifecycle within the system. It is assumed that an SLA has been formally agreed upon between the service provider and the

[4] Corda is an open-source permissioned blockchain platform designed for enterprise use, particularly in industries like finance. It supports smart contracts, known as CorDapps, which automate business processes and manage assets. Unlike permissionless platforms like Ethereum, Corda restricts participation to authorized parties, enhancing privacy and security.

consumer, and that the SLA generator module has successfully generated the SLA. In NebulOuS, the transformation of SLAs into smart contract functions is performed in an automatic manner.

The SCE is responsible for retrieving new SLAs from the AMQP server, converting them into smart contracts by invoking the generic SLA chaincode with extracted parameters, and committing these smart contracts to the Hyperledger Fabric ledger via the Fabric Gateway (FGW) for secure storage. In terms of compliance monitoring, the SCE processes real-time monitoring data received from the AMQP server. Upon receiving such data, the SCE invokes predefined functions to process the relevant actions. It then publishes these events back to the AMQP server, enabling other services in the system to consume and act upon them as needed.

Figure 7 illustrates the architecture of the SCE. The SCE is composed of several key components, each responsible for handling different aspects of SLA management, blockchain interaction, and system communication.

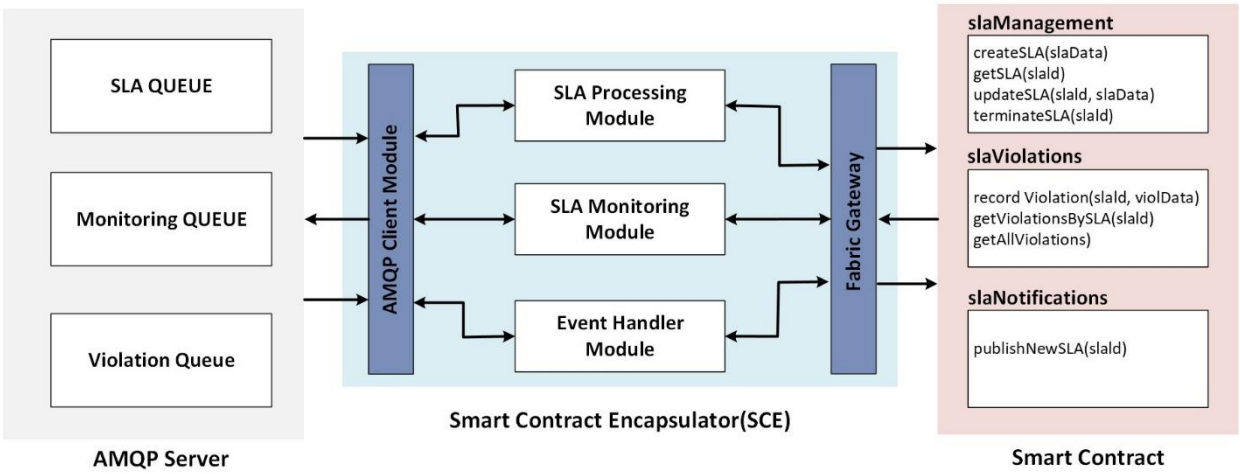

Figure 7. Smart Contract Encapsulator (SCE): Workflow and Module Communication

The AMQP Client Module is responsible for communication between the SCE and the AMQP server. It subscribes to two main queues: the SLA Queue, which receives messages containing new SLAs, and the Monitoring Queue, which receives performance data messages. In addition to receiving data, the module also publishes SLA-related updates. When the SCE processes violation or penalty data, it sends these updates to the AMQP server, where they can be accessed by other services for further processing.

The Blockchain Gateway Client handles the interaction with the blockchain network, specifically Hyperledger Fabric. It uses the Fabric Gateway to establish a secure connection, enabling the SCE to create and store SLA smart contracts on the ledger.

The SLA Processing Module parses and validates incoming SLA data, formats it appropriately, and then sends it through the Fabric Gateway to smart contract functions such as createSLA in the slaManagement module.

The SLA Monitoring Module processes incoming monitoring data from the AMQP server's Monitoring Queue, which includes SLA violation or penalty data from external sources. Upon receiving such data, it invokes predefined smart contract functions in the SLA violation handling chaincode to process the relevant actions associated with the affected SLAs on the blockchain.

3.1.2.1 Off-chain Service Monitoring

Smart contracts operate on the principle of determinism - always producing the same output given the same input and state. This characteristic guarantees transparency, consistency, and security, preventing tampering or manipulation of their execution. However, this deterministic nature imposes limitations, as smart contracts are confined to the data and events within the blockchain network.

Oracles act as vital bridges, where smart contracts get access to real-world information or off-chain data to trigger predefined actions [47]. Technically, an oracle is a middleware service or component that facilitates communication between a blockchain and external data sources. It is responsible for fetching data from off-chain systems (such as RESTful APIs, web services, IoT devices, or databases), validating its integrity and authenticity, and securely transmitting it to the on-chain environment. Cloud service providers implement monitoring systems that collect real-time data from their infrastructure. The monitoring system, through the oracle, transmits the collected data to smart contracts deployed on the blockchain. Oracles can also perform processing or aggregation steps. For instance, if the SLA involves a requirement for average response times, the oracle aggregates this data before sending it back to the smart contract.

Oracles can appear in different types:

Input Oracles: The most widely recognized type of oracle today is known as an "input oracle," which fetches data from the real-world (off-chain) and delivers it onto a blockchain network for smart contract consumption. As an example, an input oracle fetches real-time performance data from cloud servers, such as uptime, response times, and resource utilization, and delivers this data onto the blockchain. The SLA smart contract then uses this data to check if the cloud service provider is meeting its performance targets. If not, penalties may be automatically applied to the provider. An input oracle for our case is the **Event Management System** [48].

Output Oracles: The opposite of input oracles are "output oracles," which allow smart contracts to send commands to off-chain systems that trigger them to execute certain actions, for example, informing a banking network to make a payment, telling a storage provider to store the supplied data, or pinging an IoT system to unlock a car door once the on-chain rental payment is made.

Cross-Chain Oracles: Another type of oracles considered the cross-chain oracles that can read and write information between different blockchains. Cross-chain oracles enable interoperability for moving both data and assets between blockchains, such as using data on one blockchain to trigger an action on another or bridging assets cross-chain so they can be used outside the native blockchain they were issued on.

Compute-Enabled Oracles: A new type of oracle becoming more widely used by smart contract applications are "compute-enabled oracles," which use secure off-chain computation to provide decentralized services that are impractical to do on-chain due to technical, legal, or financial constraints. For example, computing zero-knowledge proofs to generate data privacy or running a verifiable randomness function to provide a tamper-proof and provably fair source of randomness to smart contracts.

3.1.2.2 SLA Evaluation & Enforcement

At predetermined intervals, the smart contract triggers its SLA evaluation function to compare the current performance data against the agreed-upon SLA thresholds. The smart contract's function that is responsible for checking SLA compliance is invoked by an external entity (Event Management System or another processes). If the service performance falls below the specified levels, the smart

contract automatically triggers the penalty mechanism. For instance, in a cloud storage SLA with a 99.9% uptime guarantee, the smart contract regularly verifies the service's uptime and enforces penalties, notifies the service provider, and/or reports the breach to anomaly detection system if uptime drops below the agreed threshold.

3.1.2.3 SLA Termination

Either party can initiate SLA termination using the smart contract. This step involves:

- Calling the termination function: Invoking the appropriate function within the smart contract to initiate termination.
- Notifying both parties: Alerting both the computing continuum provider and the consumer about the termination request.
- Confirming termination: Obtaining confirmation from the requesting party to proceed with termination.
- Terminating the smart contract: The smart contract ceases monitoring and enforcing the SLA terms.

3.1.3 Communication between SCE and Other Services

Designing new interfaces and mechanisms for obtaining feedback from the monitoring algorithms and interacting with off-chain services and other third-party services. What is the interface between the Event Management System and Smart Contract Encapsulator modules.

Figure 8 illustrates the interactions of the Smart Contract Encapsulator (SCE) with various services in the system. The SCE interfaces with the blockchain via the Fabric Gateway (FGW), facilitating secure and efficient communication with the Hyperledger Fabric network. The SLA Generator sends newly created SLAs to the AMQP server, from which the SCE retrieves them. The SCE then converts these SLAs into corresponding smart contract and commits them to the blockchain. When the Brokerage Quality Assurance (BQA) component identifies Service Level Objective (SLO) violations or applicable penalties, it transmits this information to the AMQP server. In response, the SCE invokes the appropriate smart contract functions to process the relevant actions associated with the affected SLAs on the blockchain.

Additionally, the SCE publishes SLA-related updates—both during SLA creation and upon receiving SLO violations—back to the AMQP server. These updates are consumed by other modules for further processing and analysis.

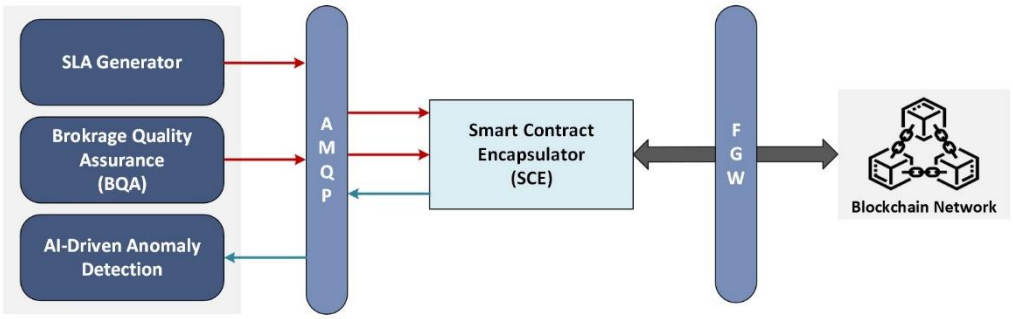

Figure 8. SCE Component Interactions

4 AUTOMATIC DEPLOYMENT OF SECURE NETWORK OVERLAY

Upon selection of the appropriate resources in which the application is deployed, an overlay network is created to securely interconnect the compute elements involved. Building upon the foundations established in the initial D4.1 deliverable, we have extended Overlay Network Manager (ONM) networking capabilities to address connectivity challenges, particularly focusing on Network Address Translation (NAT) traversal. In particular, we target scenarios where nodes such as edge devices, virtual machines, or IoT endpoints reside behind one or more layers of NAT or are deployed in firewall-restricted networks. These conditions hinder traditional VPN setups that require public IP addresses or direct peer-to-peer reachability. To overcome this, we integrate Headscale/Tailscale[5], a lightweight, WireGuard-based mesh VPN stack that automates key management and NAT traversal. On the other hand, this approach enables the secure device on-boarding and management as the communication of the NebulOuS core with the unprovisioned/ unregistered nodes is achieved over an encrypted network.

4.1 OVERVIEW

As part of the NebulOuS architecture, an Overlay Network Manager has developed targeting to:

- Ensure connectivity between NebulOuS compute resources (physical and virtual).
- Secure data in transit through encryption

This functionality is implemented by the Overlay Network Manager (ONM). During the creation of a new cluster, initiated by the Execution Adapter, the ONM connects compute resources into a secure overlay network. The system creates an on-demand VPN network, providing secure node-level connectivity (e.g., VMs or bare-metal devices) even before Kubernetes deployment. This ensures that intra-cluster traffic remains encrypted within each cluster.

4.1.1 Technical Implementation

The ONM utilizes the WireGuard VPN protocol, an open-source solution that encapsulates IP packets over UDP, enabling secure Layer-3 tunneling between physical or virtual resources.

Key components of WireGuard implementation include:

- Public Key Association: Secure communication is established via a public key associated with a tunnel source IP address.
- Network Interfaces: WireGuard creates its own network interfaces, each with a private key and a list of peers (public keys).
- Tunnel Rules: Each public key maps to specific IP addresses allowed in the tunnel, ensuring strict access control.

Once the WireGuard interface is configured with a private key and peers' public keys, data packets are securely transmitted across the network.

[5] https://github.com/juanfont/headscale

4.1.2 Achievements

The ONM implementation resulted in:

- VPN communication across nodes.
- Stable inter-node communication using a WireGuard mesh.
- Reliable pod-to-pod communication in Kubernetes, resolving issues with the Cilium Container Network Interface (CNI).
- Mitigation of networking disruptions caused by misconfigurations.
- Automation of network configurations for Kubernetes clusters (nodes and master).
- Dynamic integration of Cilium with WireGuard mesh communication via automated scripts.

Overall, these achievements enable NebulOuS to securely and automatically interconnect distributed resources across multiple clouds, edge nodes, and network domains, ensuring full interoperability in heterogeneous environments.

4.1.3 Use-case Description performing the main functionalities of ONM in a local deployment.

The following example demonstrates how pod-to-pod connectivity was achieved and tested in the deployed environment:

- Pods communicate seamlessly across nodes through a WireGuard mesh.
- WireGuard interfaces (e.g., wg0) were deployed on multiple nodes (e.g., VM2 and VM3).
- Successful connectivity was validated with pings between pods across nodes.

The image below visually represents the WireGuard-based overlay network architecture. It shows:

1. Multiple compute nodes (e.g., VM2 and VM3) are connected through a WireGuard mesh, with wg0 interfaces on each node.
2. A secure tunnel encapsulating the communication between nodes.
3. Verification of connectivity, as demonstrated by successful pings between pods residing on different nodes.

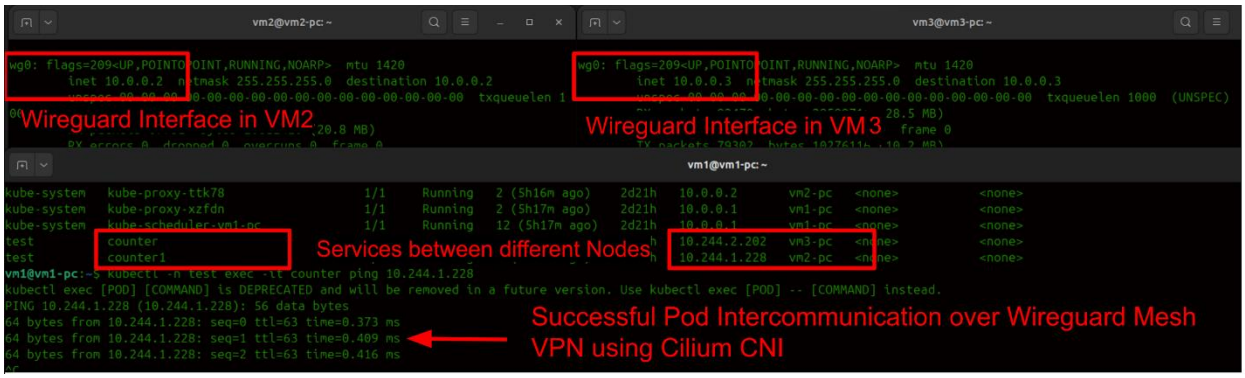

Figure 9. ONM created WireGuard-based overlay network

4.1.4 Headscale/Tailscale Implementation - Support connectivity for devices behind NAT or firewalls

The custom ONM mechanism extends the network dynamically; however, the deployment of network nodes behind NAT introduces significant challenges and high complexity. To overcome these

complexities, we focused on enhancing ONM's flexibility and adaptability for environments using NAT traversal.

After investigating the most reliable solutions for NAT traversal, we identified Headscale/Tailscale as the ideal choice. Headscale/Tailscale is open-source and integrates seamlessly with WireGuard, utilizing its foundational capabilities while extending its functionality to work in NAT-heavy environments.
Features of Headscale/Tailscale

- Simplified NAT Traversal: Effortlessly connects devices located behind NATs or firewalls.
- Automatic Configuration: Minimizes manual setup by dynamically configuring nodes for secure communication.
- WireGuard Integration: Maintains compatibility with the ONM's previous implementation, as it uses WireGuard as its underlying protocol.

4.1.5 Deployment and Testing

To validate the Headscale/Tailscale solution, we conducted a deployment across three geographically distributed nodes:

1. Headscale Server: Located in the Ubitech testbed.
2. VM: Hosted in the Thessaloniki testbed, situated behind a double NAT (host machine and virtual machine).
3. Raspberry Pi: Deployed in the Athens experimentation environment, also positioned behind NAT.

The setup successfully established a VPN network interconnecting the Thessaloniki and Athens nodes. The Headscale server, located in a separate Athens environment, automatically configured the Tailscale clients for secure connectivity.

Kubernetes Cluster Deployment

Following the successful VPN configuration, we deployed a Kubernetes cluster:

- Master Node: Thessaloniki testbed.
- Worker Node: Raspberry Pi in Athens.

We integrated Cilium CNI to ensure compatibility with the broader NebulOuS implementation. Challenges related to Cilium routing over the Tailscale VPN were addressed, and pod-to-pod communication across nodes was successfully established.
The following implementation steps and testbed specifications describe the detailed configuration used for setting up the cluster
Configuring Tailscale on Nodes:

- e.g., tailscale up --login-server https://hs.ubitech.eu --auth-key 2f01a9aba8…..

Testbed Environment Specifications
VM node

- OS: Ubuntu 24.04.1 LTS
- Resources: 2 vCPUs, 4GB RAM, 20GB disk storage.

Raspberry Pi:

- OS: Linux 5.15.0-1070-raspi, Ubuntu 22.04.5 LTS (Jammy Jellyfish).
- Resources: 4 vCPUs, 8GB RAM, 60GB disk storage.

Key Achievements

- Seamless NAT traversal and secure communication between nodes using Headscale/Tailscale.

- Reliable pod-to-pod communication in the Kubernetes cluster, aligning with NebulOuS project goals.
- Enhanced automation for Kubernetes deployment and networking configurations.

4.1.6 Validation of Kubernetes Pod Communication over Headscale/Tailscale Mesh with Cilium CNI

Figure 10 illustrates the successful deployment and operation of a Kubernetes cluster utilizing a WireGuard-based VPN mesh and Cilium CNI based on the deployment described in the previous subsection. The target was the successful pod-to-pod communication between a the VM located in Thessaloniki (k8s Master) and the Raspberry pi (k8s Worker) utilizing Headscale/Tailscale VPN interfaces.

The figure comprises three panels (terminals):

1. Cluster Deployment

The top terminal showcases the output of the *kubectl get pods -A -o wide* command, which lists all running pods across namespaces within the cluster. Key details include:

- Pods in Multiple Namespaces: The default, kube-system, and cilium namespaces are represented.
- Node Assignments: Pods are distributed between the raspberrypi and vm8-pc nodes.

This confirms that the cluster is operational, with pods running across multiple nodes.

2. Pod-to-Pod Communication Test

The middle terminal demonstrates a successful ICMP communication test between two pods across the cluster.

- A pod (nginx-arm-6f7f59658b-n9jjq) initiates a ping to another pod with the IP address 10.244.0.91.
- Results show 0% packet loss, with a round-trip time (RTT) averaging around 24.429 ms, confirming reliable pod-to-pod communication across the WireGuard VPN mesh.

3. Cilium CNI Health Status

The bottom panel displays the health status of the Cilium CNI through the command *kubectl exec -ti cilium-fnmvg cilium-health status.*

- The health check confirms that connectivity tests (ICMP and HTTP) between the cluster nodes (raspberrypi and vm8-pc) are successful.
- The RTT values for ICMP and HTTP are recorded, reflecting low-latency and stable communication.

To summarize, this figure demonstrates the effectiveness of the Headscale/Tailscale integration and Cilium CNI in facilitating seamless and secure communication across a Kubernetes cluster. It highlights:

1. Proper deployment of Kubernetes pods across nodes.
2. Reliable inter-pod communication over a WireGuard VPN mesh.
3. A fully operational Cilium network with stable connectivity between nodes.

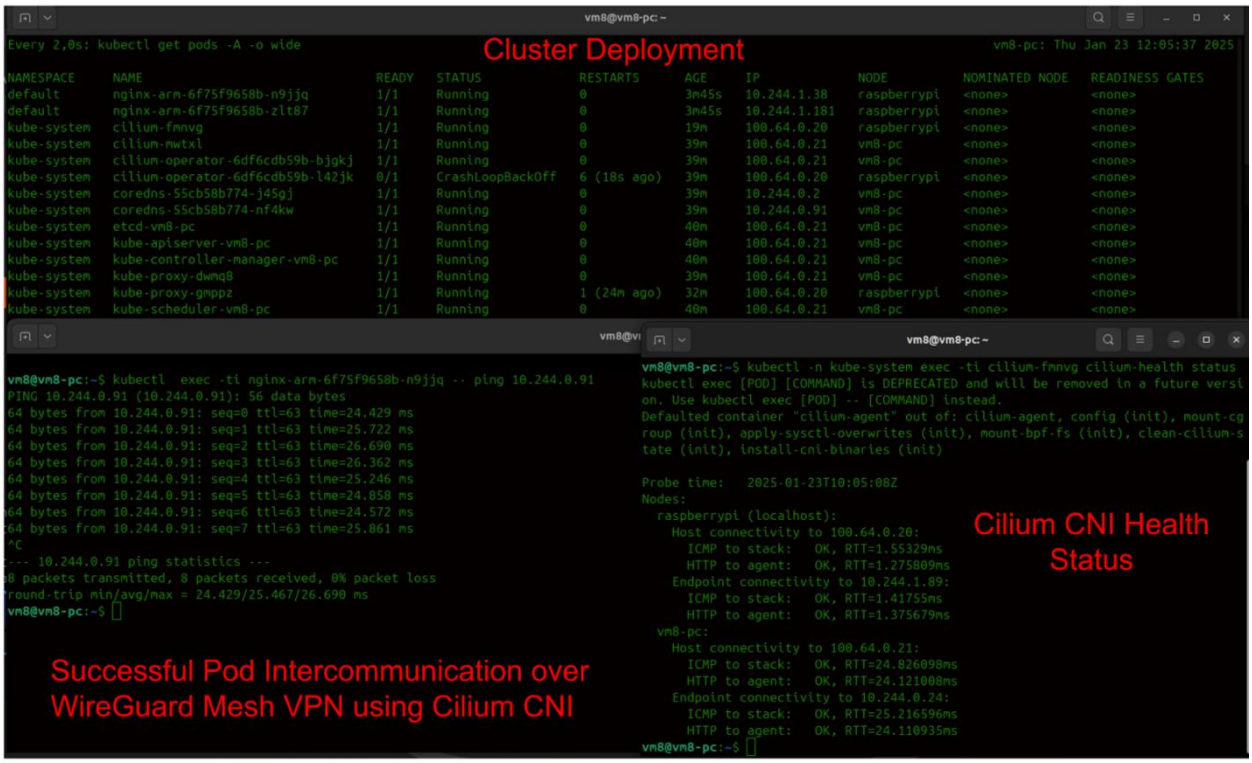

Figure 10. Headscale/Tailscale integration and Cilium CNI

4.2 SECURE DEVICE ONBOARDING AND MANAGEMENT - INTEGRATION OF HEADSCALE/TAILSCALE SOLUTION IN NEBULOUS CORE

In scenarios where multiple geographically distributed nodes seek to collaboratively form Kubernetes clusters, NebulOuS facilitates the automated and secure creation of these clusters while addressing a set of core requirements.

The Overlay Network Manager (ONM) subsystem is responsible for establishing secure communication channels between pods across different nodes in the Kubernetes cluster. This is accomplished through the following mechanisms:

- The ONM utilizes WireGuard to generate cryptographic key pairs (private and public keys) for each node participating in the cluster.
- These keys are then used to automatically configure secure VPN tunnels between the nodes.
- Generates a WireGuard IP for each node
- All node-to-node traffic is encrypted and routed through these secure tunnels, providing end-to-end encryption.

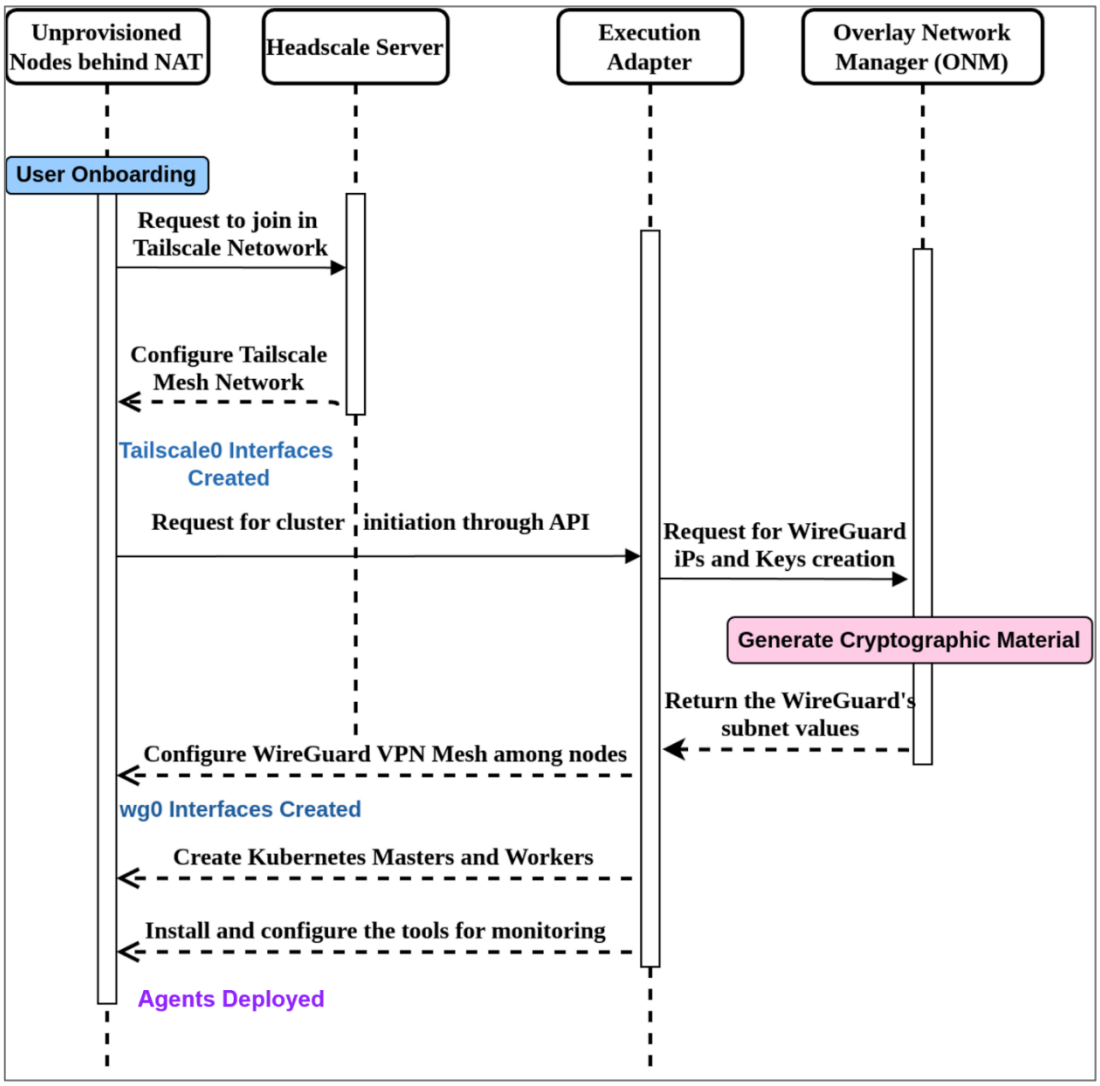

Figure 11. NebulOuS – Cluster Initiation Procedures integrating ONM functionalities

The Execution adapter is the foundation component which enables the secure registration, configuration, and management of distributed Kubernetes clusters. NebulOus components deployed as pods (e.g, the ONM and Resource Manager Pod) with a Tailscale sidecar container ensuring that traffic remains encrypted and authenticated. The system exposes a REST API through which users can initiate cluster creation by providing necessary node information in a structured JSON format. This information includes IP addresses, SSH credentials (username and password/key), and cluster configuration parameters. Upon receiving this information, the Nebulous initiates a multi-stage workflow, as illustrated in Figure 11:

1. User Onboarding: Users are first authenticated and onboarded into the Nebulous secure network using Tailscale. This is accomplished by executing *<tailscale up --login-server <headscale server> --auth-key <auth-key>*"> on their unprovisioned nodes. The Headscale server URL and authentication key are provisioned by Nebulous system administrators, ensuring only authorized users can access the VPN network.
2. Node Registration and VPN Overlay Initialization. After user authentication, the Execution Adapter triggers the ONM to establish the secure VPN mesh between cluster nodes. This

process involves generating cryptographic material and establishing secure tunnels between participating nodes, as detailed in next Subsection.

3. ONM Kubernetes Cluster Bootstrapping. Once secure communication is established, the system deploys the required Kubernetes components on the designated nodes. This is accomplished through SSH-based remote execution, where bootstrap scripts are transferred to the target nodes and executed with appropriate parameters. The bootstrapping process designates master and worker nodes according to the user-defined configuration and initializes the Kubernetes control plane.

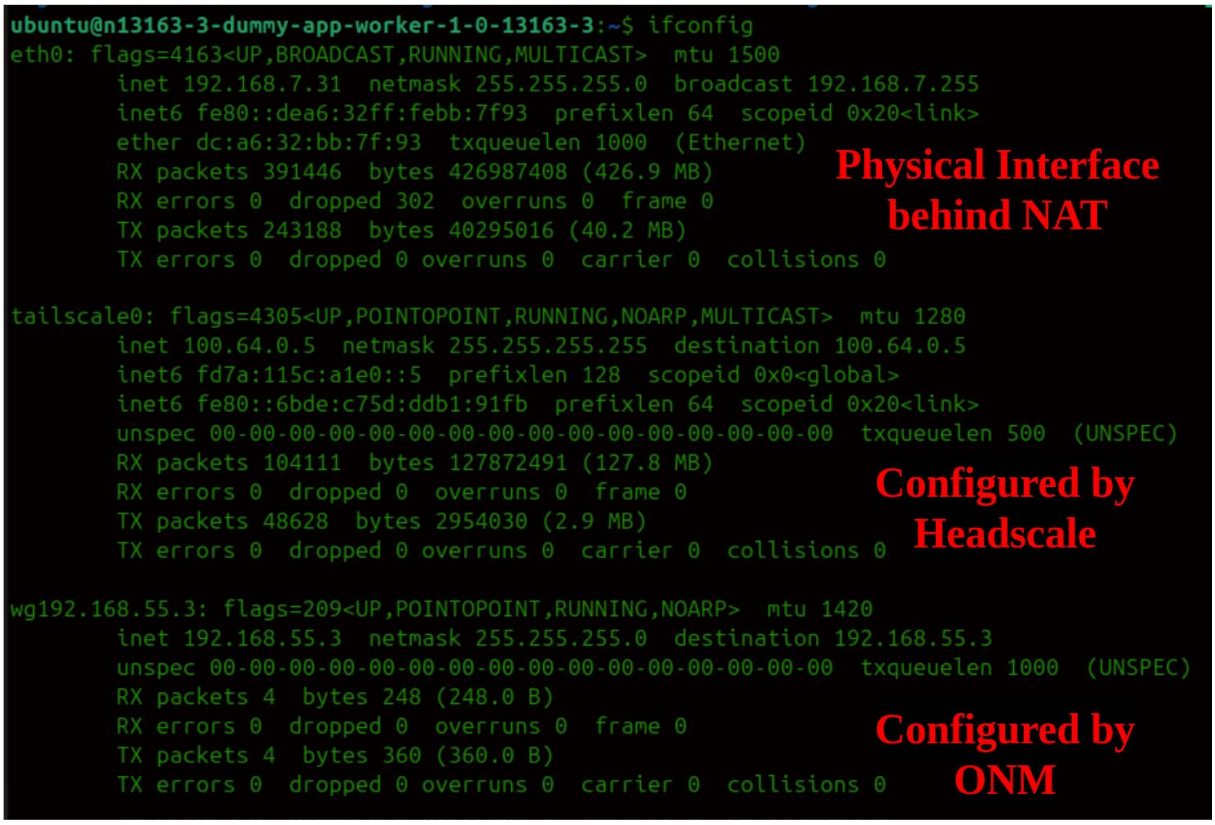

Figure 12. Created Interfaces on edge cloud node (Raspberry Pi)

The core components of NebulOuS, alongside the ONM, and the Kubernetes Cluster creation (bootstrap scripts), and the Tailscale solution, have been tested both locally and across multiple clusters within the broader framework of the NebulOuS project. Specifically, (i) the NebulOuS pods provide communication over Tailscale for exposed ports; while (ii) the ONM with Kubernetes Cluster creation [onmgithub] are main components of the project and have been extensively tested in multiple scenarios with geographically distributed nodes.

Figure 12 illustrates the created network interfaces on a Raspberry Pi device configured through Nebulous. The image shows: (i) the Tailscale IP through which the Raspberry Pi communicates with the Execution Adapter, the ONM, as well as (ii) the ONM-created interface that enables communication among Kubernetes nodes via a WireGuard-based VPN mesh network.

The main components of the Overlay Network Manager (ONM) and its supporting scripts are publicly available:

- **Overlay Network Manager (ONM) – Java service:**
  https://github.com/eu-nebulous/overlay-network-manager
  Orchestrates the overlay network setup and manages the global state in a Postgres DB.
- **Overlay bootstrap script – Bash script:**
  https://github.com/eu-nebulous/sal-scripts/tree/main/installation-scripts-onm
  Executed by SAL/Proactive on infrastructure resources to initiate VPN bootstrapping by contacting ONM.
- **WireGuard configuration scripts – Bash scripts:**
  https://github.com/eu-nebulous/overlay-network-manager/tree/main/network-manager/bootstrap-agent-scripts/wireguard
  Executed by ONM on infrastructure resources to install and configure the WireGuard VPN.

4.3 SUMMARY OF TECHNICAL CONTRIBUTIONS AND CHALLENGES

The presented system combines several protocols and tools to enable fully automated deployment of secure overlay networks across distributed Kubernetes clusters. The integration of WireGuard with Headscale/Tailscale provides robust VPN connectivity, even in complex environments with NAT or firewall restrictions. At the same time, the automation of the entire process through the Overlay Network Manager (ONM) ensures that clusters can be dynamically extended with minimal manual configuration.

The solution addresses multiple challenges such as NAT traversal, secure device onboarding, inter-node encryption, dynamic key management, and integration with Kubernetes networking (e.g., Cilium CNI). By combining lightweight VPN mesh capabilities, automation scripts, and container-native deployment, the system achieves secure cluster formation across heterogeneous and geographically distributed infrastructures. This allows NebulOuS to support cloud-to-edge deployments while maintaining end-to-end encrypted communication between nodes and pods.

5 SECURE AND PRIVACY-BY-DESIGN IN DATA STREAMS PROPAGATION

The second pillar of the NebulOuS security and privacy mechanisms focuses on policy-based access control. This allows users to define and enforce rules that control who can access specific resources. In addition to access control, the system also includes network policies and real-time observability and enforcement to monitor and react to security events. Policies for controlling how data propagates across components are also supported. Since the NebulOuS Meta-OS uses Kubernetes for container orchestration, securing access to Kubernetes clusters is a key focus. Finally, NebulOuS imposes measures to secure the communication between the NebulOuS control plane and the application's cluster master.

5.1 APPROACH OVERVIEW

5.1.1 Access Control In Kubernetes

Access Control in Kubernetes is practically realized by controlling access to the Kubernetes API [49]. Kubernetes (k8s) provides a well-structured way to achieve fine-grained cluster access control, using "**admission controllers**".

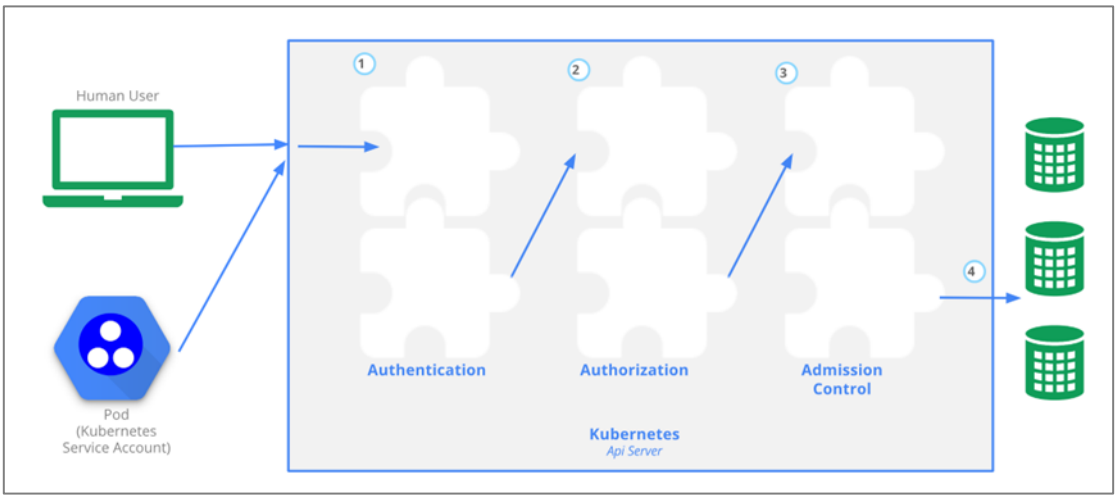

Figure 13. Access Control in Kubernetes

In k8s, every communication goes through the API Server. Changes that come through the API server are persisted into etcd..

An **Admission Controller** is code that runs after API server requests are authenticated and authorized, and before the request results in a change to etcd[6]. They intercept inbound mutation requests. An admission controller can, thus, mutate or reject the requests based on user-defined policies.

Although there are build-in admission controllers in k8s, external admission plugins can also be run as webhooks configured at runtime to achieve Dynamic Admission Control [50]. Dynamic Admission Controllers are made possible by loading the MutatingAdmissionWebhook and ValidatingAdmissionWebhook compiled admission controllers when the API server starts.

Admission webhooks in K8s are HTTP callbacks that receive 'admission requests' and do something with them. There are two types of admission webhooks: Validating Admission Webhook and Mutating Admission Webhook. Mutating admission webhooks are invoked first; they can modify objects sent to the API server to enforce custom defaults.

After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and can reject requests to **enforce custom policies**. This webhook calls out to a configured policy engine service to have the current payload validated by any policies that match. If the validation results in false return, then the request stops and the status is immediately returned back to the calling client, by the API server.

With these two admission controllers running, we can configure extensions to the API server request flow at runtime, using services running on data plane nodes. This means that after the API server is

[6] It should be noted that admission controllers do not respond to Kubernetes read operations, like get, watch and list. To prevent those operations, we will use RBAC.

up and the cluster is running, we can add policy engine services to the data plane at runtime and configure them to be called by API server webhooks.

NebulOuS offers the definition and enforcement of arbitrary, custom security policies by its users leveraging the aforementioned native Kubernetes mechanisms. This way, we allow for fine-grained control on **who is allowed to perform what actions on Kubernetes clusters, under a particular context.**

In the case of Dynamic Admission Control, the exact flow of an API request is depicted below:

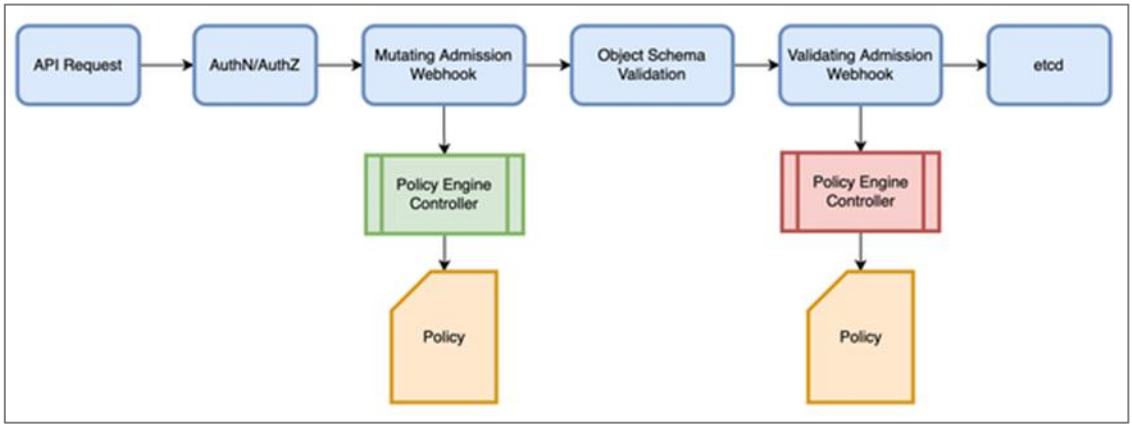

Figure 14. Kubernetes Dynamic Admission Control using external policy engines: flow of an API request

5.1.2 Policy engine

There are several policy engines that can act as admission controllers for Kubernetes (Kyverno [51], Open Policy Agent [52], jsPolicy [53], KubeWarden [54], etc.). In NebulOuS, we are using OPA Gatekeeper [55], which is the official Kubernetes-native integration of the **Open Policy Agent (OPA)**.

OPA Gatekeeper is an open-source policy engine that enforces fine-grained access and security rules across Kubernetes resources. It uses **constraint-based policies** to define what is allowed within the cluster. These policies are written in a declarative way and are enforced automatically when any resource is created or updated.

Gatekeeper supports both **role-based** and **attribute-based** access controls and allows for building custom rules that reflect specific organizational or application needs. Policies can restrict actions based on the resource type, labels, namespaces, user identity, or other Kubernetes metadata.

Gatekeeper works by:

- **Constraints**, which define what rules must be followed.
- **ConstraintTemplates**, which define the logic behind the rules.
- A built-in **controller**, which checks incoming API requests and blocks those that violate defined constraints.

This model allows flexible and consistent enforcement of policies across clusters. Changes to authorization behavior can be made by simply updating or adding constraints, without modifying the application logic or cluster setup.
OPA Gatekeeper also integrates with auditing tools, enabling the detection of policy violations even if enforcement is not active. This helps with gradually adopting policies in a safe, observable way.

5.1.3 OPA Gatekeeper Policies

OPA Gatekeeper is used for policy enforcement in Kubernetes through its integration as a Validating Admission Webhook [56]. This means it checks every API request to the Kubernetes cluster, such as creating or updating resources, and decides whether the action should be allowed based on defined policies. These policies help prevent operations that don't meet organizational or security requirements.
When a user sends a request to perform an action in the cluster, like deploying a pod or creating a service, Gatekeeper evaluates that request against all active constraints. If any policy is violated, the request is rejected before the change takes effect. This allows administrators to define clear rules about what is allowed in the cluster, such as disallowing certain container images, enforcing naming conventions, or requiring labels on resources.
OPA Gatekeeper makes it possible to manage policies declaratively, review violations through audit logs, and ensure consistent enforcement across environments [57].

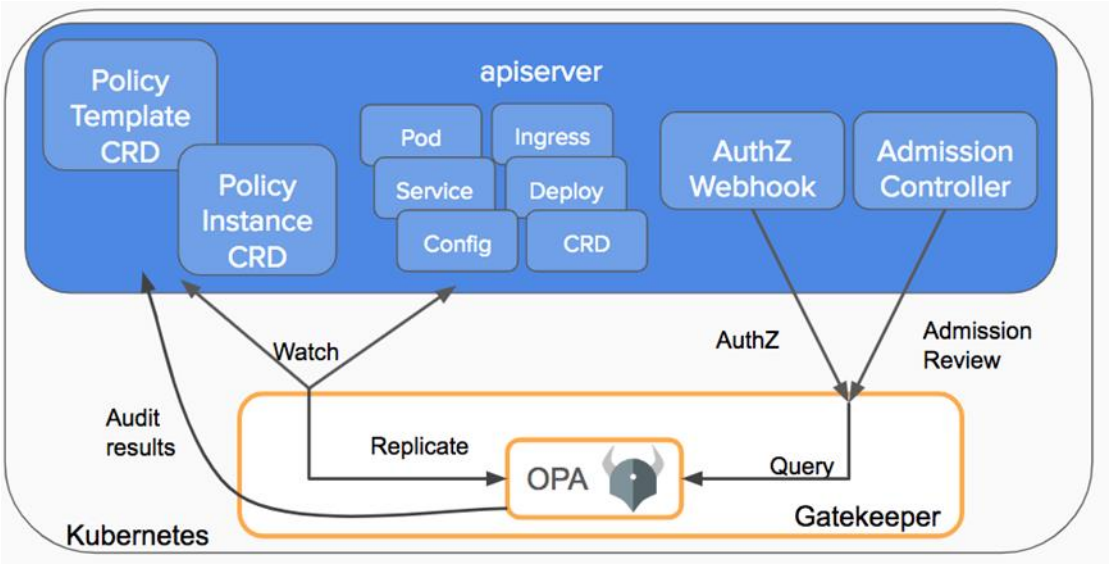

Figure 15. OPA Gatekeeper as a Validating Admission Webhook

5.1.3.1 Examples

For example to restrict container images to trusted sources, NebulOuS uses OPA Gatekeeper to enforce repository-based constraints. The policy below ensures that only images pulled from specific, approved repositories [58] (e.g., `nebulousrepo/`) can be used in Pod definitions.
This enforcement is useful for securing workloads and preventing the use of unverified or unauthorized container images.
First comes the definition of a ConstraintTemplate named "`k8sallowedrepos`":

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8sallowedrepos
  annotations:
    metadata.gatekeeper.sh/title: "Allowed Repositories"
    metadata.gatekeeper.sh/version: 1.0.2
    description: >-
      Requires container images to begin with a string from the specified list.
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRepos
```

```
      validation:
        openAPIV3Schema:
          type: object
          properties:
            repos:
              description: The list of prefixes a container image is allowed to have.
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sallowedrepos

        violation[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          not strings.any_prefix_match(container.image, input.parameters.repos)
          msg := sprintf("container <%v> has an invalid image repo <%v>, allowed repos are %v",
[container.name, container.image, input.parameters.repos])
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.initContainers[_]
          not strings.any_prefix_match(container.image, input.parameters.repos)
          msg := sprintf("initContainer <%v> has an invalid image repo <%v>, allowed repos are
%v", [container.name, container.image, input.parameters.repos])
        }

        violation[{"msg": msg}] {
          container := input.review.object.spec.ephemeralContainers[_]
          not strings.any_prefix_match(container.image, input.parameters.repos)
          msg := sprintf("ephemeralContainer <%v> has an invalid image repo <%v>, allowed repos
are %v", [container.name, container.image, input.parameters.repos])
        }
```

Then the Constraint that follows the set of the above rule is defined:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: repo-is-openpolicyagent
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
  parameters:
    repos:
      - " nebulousrepo/"
```

This constraint will allow only Pods in the default namespace to use container images starting with `nebulousrepo/`.

The following Pod uses an image from an unauthorized source and will be denied:

```
apiVersion: v1
kind: Pod
metadata:
  name: unauthorized
  namespace: default
spec:
  containers:
    - name: nginx
      image: unauthorizedrepo/nginx:1.14.2
```

OPA Gatekeeper will reject this request with a message indicating that the image repository is not allowed.

This Pod complies with the policy and will be admitted:

```
apiVersion: v1
kind: Pod
metadata:
  name: authorized
  namespace: default
spec:
  containers:
    - name: nginx
      image: nebulousrepo/nginx:1.14.2
```

Because the image starts with the allowed prefix, the admission request passes the policy check.

5.1.3.2 Creating Constraints

When OPA Gatekeeper authorizes a request, it evaluates the full Kubernetes AdmissionReview object, which contains details about the resource being created or modified. This request is processed against any active policies defined in Gatekeeper.

As part of its setup, Gatekeeper installs a set of predefined `ConstraintTemplates` from the official OPA Policy Library. These templates provide ready-to-use rules for enforcing common security and operational practices, such as disallowing privileged containers, requiring specific labels, or restricting image registries.

In NebulOuS, these templates are not only applied to standard resources like Pods and Deployments, but more importantly, they are extended to validate **Application** resources. This is especially relevant because NebulOuS adopts the **Open Application Model (OAM)**, where the Application resource is a central abstraction. By enforcing constraints at the Application level, we ensure policies are applied directly to user-defined workloads, regardless of their internal structure.

As with all Gatekeeper policies, these ConstraintTemplates and their corresponding Constraints are defined as YAML or JSON files and can be deployed using standard Kubernetes tooling, such as:

```
kubectl apply -f <filename>
```

5.1.4 Cilium Network Policies

Kubernetes supports network policies that control how pods talk to each other and to the outside world. These policies help define what kind of traffic is allowed into or out of a pod. However, the built-in Kubernetes network policies are limited. They can't easily handle things like DNS-based rules or identity-aware filtering.

To improve this, NebulOuS uses **Cilium** for network security. Cilium is built on eBPF, a Linux feature that lets the system enforce rules directly in the kernel, without adding extra software. This makes policy enforcement fast and efficient [59].

Cilium adds its own type of network policy, called a CiliumNetworkPolicy. These policies let us write more detailed rules than standard Kubernetes policies. For example, Cilium policies can allow or block traffic based on the labels of pods (rather than IP addresses), the port or protocol being used, or even the domain name being accessed. This is useful in dynamic cloud environments, where IP addresses change often.

In NebulOuS, we use Cilium to control how application components talk to each other, especially those defined using the Open Application Model (OAM). Cilium helps enforce strict separation

between applications, limit outgoing traffic, and block unwanted network access. This supports the platform's goal of strong, built-in security.

The example below shows a simple Cilium policy that blocks all incoming traffic to pods with the label `app: my-service` on port `8080`. It stops any pod or external client from connecting to that port.[60]

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: block-port-8080
  namespace: default
spec:
  endpointSelector:
    matchLabels:
      app: my-service
  ingress:
    - toPorts:
        - ports:
            - port: "8080"
              protocol: TCP
      fromCIDRSet:
        - cidr: "0.0.0.0/0"
```

This policy applies to any pod in the `default` namespace with the label `app: my-service`. It blocks TCP traffic on port `8080` from all sources (`0.0.0.0/0`). The policy is defined in YAML, like any other Kubernetes resource, and enforced automatically by Cilium.

5.1.5 Security Observability and Logging

While policy enforcement at admission time (e.g., through OPA Gatekeeper) prevents non-compliant resources from being created, runtime enforcement is critical to detect and respond to security events that occur during execution.

Tetragon[7] represents a significant advancement in Kubernetes runtime security enforcement for several theoretical and practical reasons:
Tetragon is implemented as an eBPF-based security observability and runtime enforcement solution. Unlike traditional monitoring tools that rely on system calls or log analysis, Tetragon leverages eBPF (extended Berkeley Packet Filter) technology to observe system activities at the kernel level with minimal performance impact.
The theoretical advantages of Tetragon include:

- Kernel-Level Visibility: Tetragon operates at the kernel level, providing visibility into all system activities, including process executions, file accesses, and network connections.
- Low Overhead Monitoring: eBPF technology allows for efficient filtering and processing of events directly in the kernel space, significantly reducing the performance impact compared to traditional monitoring approaches.
- Real-Time Enforcement: Tetragon can detect and respond to security violations as they occur during application runtime, enabling immediate threat mitigation.

[7] https://tetragon.io/

- Granular Observability: Tetragon can observe fine-grained activities such as specific system calls, process creations, file operations, and network connections, providing a comprehensive view of application behavior.

Tetragon's architecture is built around its ability to monitor and enforce security at the kernel level through eBPF programs. The workflow follows these key steps:

1. eBPF Programs Attachment: Tetragon attaches eBPF programs to kernel hooks, intercepting system events such as process executions, file operations, and network connections.
2. Event Generation: When observed activities match predefined patterns or policies, Tetragon generates structured events that include detailed contextual information about the activity.
3. Policy Evaluation: These events are evaluated against security policies defined as TracingPolicy or RuntimePolicy resources in Kubernetes.
4. Response Actions: Based on policy evaluation, Tetragon can trigger various response actions, including logging, alerting, or actively blocking the operation through eBPF enforcement mechanisms.
5. Integration with Security Tools: Tetragon forwards security events to a centralized collection system, Elasticsearch serves as the repository for these events, enabling long-term storage, correlation, and analysis.

Implementation of centralized data collection and observability through tetragon could be achieved with a pipeline built on Tetragon, Filebeat[8], Elasticsearch[9], and Kibana[10]:

1. **Tetragon** agents on each node capture kernel-level events through eBPF programs.
2. **Filebeat** collects these security events and forwards them reliably to centralized storage.
3. **Elasticsearch** indexes all events, providing scalable storage and efficient querying capabilities.
4. **Kibana** enables visualization and analysis through customizable dashboards.

5.1.6 Data Stream Propagation Control

While the original NebulOuS architecture, as reported in *D6.1 - 1st Release of the NebulOuS Integrated Platform and Use Case Planning* section *2.3. GROUNDED ARCHITECTURE*, had most components running on the NebulOuS control plane (as depicted in Figure 16), a new iteration of the architecture has been introduced, with several key components moved to the application cluster master (Figure 17).

[8] https://www.elastic.co/beats/filebeat
[9] https://www.elastic.co/elasticsearch
[10] https://www.elastic.co/kibana

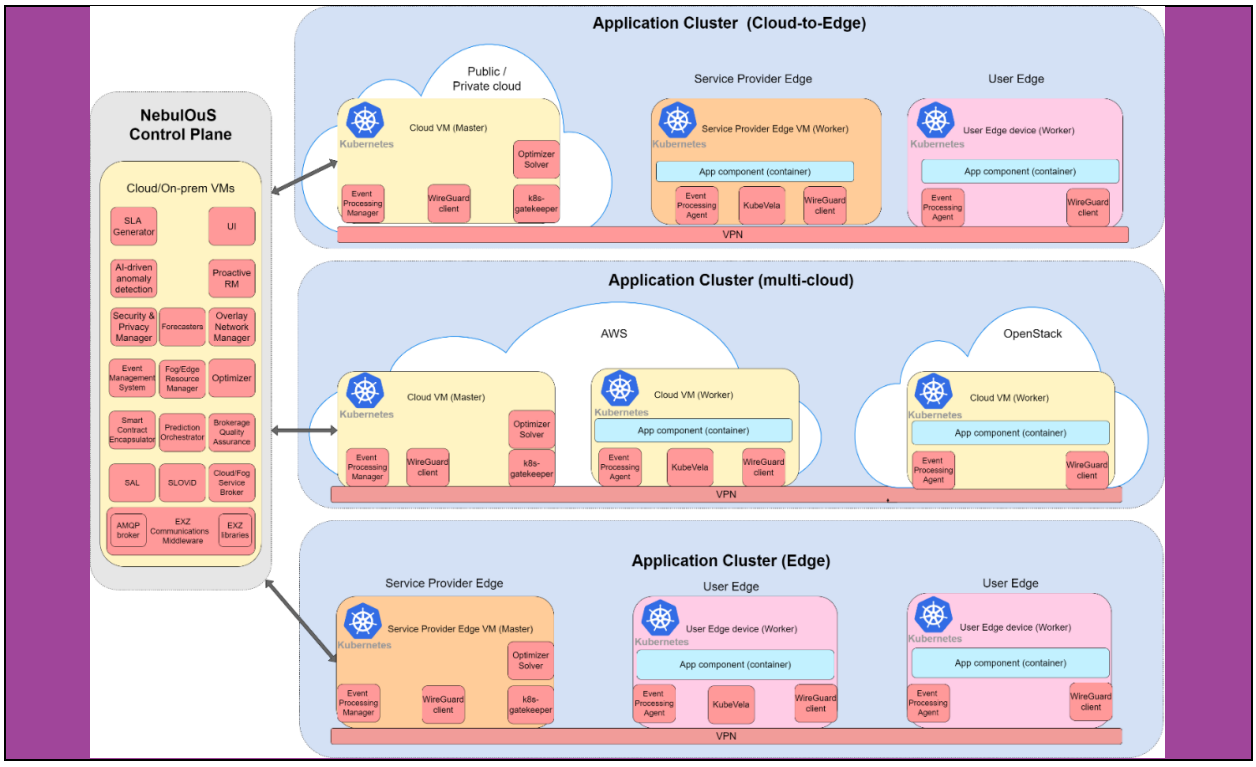

Figure 16. NebulOuS old architecture (deployment view)

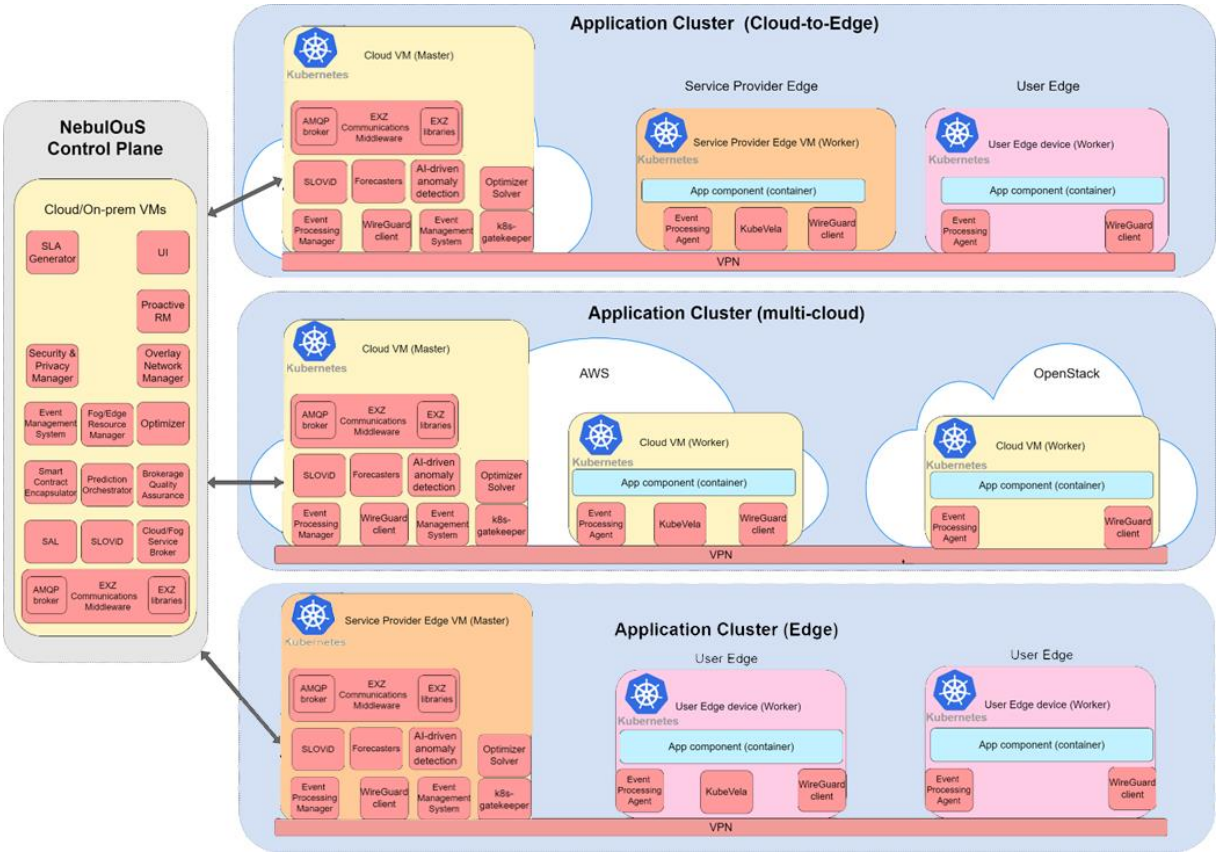

Figure 17. NebulOuS new architecture (deployment view)

The components now deployed on the application cluster master include: Metric prediction systems (forecasters and orchestrator); SLO violation detection (SLOViD); Anomaly detection and Time series database and message broker (AMQP Broker) supporting these operations.

This architectural restructuring significantly reduces message passing between the application cluster master and the NebulOuS control plane, as metrics collected by applications no longer need to be sent to the control plane. However, this change necessitates the deployment of a message broker in the application cluster master, similar to the one in NebulOuS core. This local message broker facilitates communication between NebulOuS components running on the application cluster master, primarily enabling the EMS to send metrics to the prediction mechanism and SLO violation detection components.

While the architectural changes reduce overall communication overhead, essential interactions between the NebulOuS control plane and application cluster master remain necessary. These include control plane operations such as the Optimizer controller sending AMPL files to the solver and metric lists to EMS, as well as reverse communications where the solver sends reconfiguration requests back to the optimizer controller.

To enable this distributed communication model, two specialized plugins for Apache ActiveMQ Artemis (the message broker) have been developed: The **control plane bridge plugin** intercepts cluster definition messages from the optimizer controller to SAL, configuring the control plane message broker to automatically forward specific messages to the broker at the application cluster master. Complementarily, the **application cluster master bridge plugin** establishes connectivity with the control plane message broker upon startup and manages upstream message forwarding.
The system implements a bidirectional bridge between the application cluster and control plane, with robust security measures in place. In one hand all messages received by the control plane broker coming from a bridge users session are validated to confirm that the application ID property (embedded in all messages) matches the user's identity. This verification mechanism ensures message isolation between applications, preventing cross-application interference - for instance, ensuring that only Application A can transmit messages pertaining to its own operations. Furthermore, both brokers utilise a unique set of credentials, meaning that credentials to communicate brokers for Application A are different than these used by Application B.

Source code for the application cluster plugin and the control plane bridge plugin can be found in the project GitHub[11].

5.2 IMPLEMENTATION

5.2.1 Security and Privacy Manager

The **NebulOuS Security and Privacy Manager** is implemented as a backend service in **Java 17**, using the **Quarkus [12]3.6.1** framework. It provides full support for managing admission control and network security policies across multiple Kubernetes clusters from the control plane.

[11] https://github.com/eu-nebulous/iot-dpp-orchestrator/tree/iot-pipelines/iot_dpp/src/main/java/eut/nebulouscloud/bridge
[12] https://quarkus.io/

The service uses the **Fabric8** [13]**Kubernetes Java client** to communicate with the Kubernetes API of each cluster. This client allows the service to create, update, and delete Kubernetes-native resources such as Gatekeeper policies and Cilium network policies. Cluster access credentials (in the form of kubeconfig or k3s configuration files) are mounted as a **Kubernetes ConfigMap** [14]during installation. This removes the need for external storage or databases to manage connectivity. All policy definitions and their state are stored directly within the Kubernetes clusters, using Custom Resource Definitions (CRDs)[15].

When **OPA Gatekeeper** is installed, it registers standard policy enforcement CRDs. The Security and Privacy Manager interacts with these CRDs via Fabric8 to manage **ConstraintTemplates** and **Constraints**. In addition to admission policies, the component now also integrates with **Cilium**, enabling the definition and enforcement of **network-level security policies**.

This unified approach supports policy management across both the application and network layers, using Kubernetes-native tools and interfaces.

For the second release, the Security and Privacy Manager support CRUD operations for cluster admission rules, using some of the core methods:

- **`createOrUpdateConstraint`** `(ConstraintDTO constraintDTO)`
- **`listConstraints`**`(String namespace)`
- **`deleteConstraint`**`(String name, String namespace)`

The aforementioned methods can be found in the *ConstraintService.java* file. The DTO used in this method can be found in *ConstraintDTO.java*, and consists of the following fields:

```
public class ConstraintDTO {
    private String name;
    private String namespace;
    private List<Kind> kinds;
    private List<String> namespaces;
    private List<String> repos;
    private List<Range> ranges;
    private List<String> tags;
    private List<String> exemptImages;
}
```

`createOrUpdateConstraint(ConstraintDTO constraintDTO)` creates or updates a Constraint resource that applies a specific template to a resource type (e.g., Pods, Applications) and namespace, including any required parameters.

`listConstraints(String namespace)` returns all active Constraints in the specified namespace. This provides visibility into which policies are currently enforced on cluster resources.

`deleteConstraint(String name, String namespace)` deletes a specific Constraint instance, stopping the enforcement of its associated policy for the targeted resources.

[13] https://fabric8.io/
[14] https://kubernetes.io/docs/concepts/configuration/configmap/
[15] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

In addition to Gatekeeper policies, the Security and Privacy Manager supports the creation and management of CiliumNetworkPolicy resources. These policies provide identity-aware, fine-grained control over network traffic at the L3/L4 level and are enforced by Cilium using eBPF.

The service uses the Fabric8 client to interact with the Kubernetes API and manage these network policies in a Kubernetes-native way. All operations are performed without requiring direct user access to the cluster or Cilium tooling.

- **createOrUpdateCiliumPolicy** (CiliumPolicyDTO ciliumPolicyDTO)
- **listCiliumPolicies** (String namespace)
- **deleteCiliumPolicy** (String name, String namespace)

The aforementioned methods can be found in the *CiliumPolicyService.java* file. The DTO used in these operations is defined as follows:

```
public class CiliumPolicyDTO {
    private String name;
    private String namespace;
    private String policySpec;
}
```

createOrUpdateCiliumPolicy(CiliumPolicyDTO, ciliumPolicyDTO) creates or updates a CiliumNetworkPolicy resource in the specified namespace, using the provided specification.

listCiliumPolicies(String namespace) retrieves all active Cilium policies in a given namespace.

deleteCiliumPolicy(String name, String namespace) deletes the specified policy from the cluster, removing the associated network rules.

5.2.2 EFK Logging Stack

NebulOuS includes a centralized logging and observability system to collect, store, and analyze logs from all clusters in the platform. Elasticsearch functions as the central search and analytics engine (log store) for collected logs, Kibana provides a web-based interface to query and visualize the log data, and Fluentd[16] serves as the log collector that gathers logs and forwards them to Elasticsearch for indexing [61]. In the NebulOuS platform's logging architecture, Elasticsearch and Kibana are deployed in the core cluster (central logging infrastructure), while Fluentd runs on every node of each application cluster and on NebulOuS Core to capture node-level logs and send them to the centralized Elasticsearch for indexing and analysis. Monitoring these security-relevant logs continuously is crucial for early threat detection [62].

In the NebulOuS platform, the EFK stack is deployed in a centralized yet distributed manner to gather logs from across the cloud continuum. Elasticsearch (with its data nodes) and Kibana are deployed on the NebulOuS core cluster, providing a central repository and user interface for all logs. Meanwhile, Fluentd is deployed on each node of every application cluster, as well as on NebulOuS Core, typically via a container DaemonSet. This means a Fluentd agent runs on every node to capture container and system logs locally and forward them to the Elasticsearch cluster on NebulOuS Core. The centralized Elasticsearch instance indexes and stores these aggregated logs, and Kibana on the enables engineers to search and visualize the logs through a web dashboard. Collecting and monitoring security relevant

[16] https://www.fluentd.org/

logs is crucial for early detection of suspicious activities and threats, serving as a first line of defense in protecting the system.

5.2.3 Tetragon Integration

In this section, we describe the observability within a Kubernetes cluster using Tetragon. As shown in Figure 18, there is a central point where data is collected from the observability tool. For example, in the Nebulous cluster, each node contains Tetragon agents with Filebeat sidecars that send data to Elasticsearch (central point in Nebulous core). Through Kibana, custom rules are implemented in Elasticsearch to detect malicious activities such as process-level threats. These include attempts of privilege escalation within containers, such as unauthorized usage of commands like sudo or su.

Figure 19 illustrates the central data collection in Elasticsearch located in Nebulous. This image shows the Elasticsearch/Kibana interface displaying logs from a Cilium and Tetragon monitoring system. The interface is being accessed through a browser at kibana.test.nebulouscloud.eu. The main panel shows a log viewer with "logs-cilium_tetragon.log-default" selected as the data view. The timeline graph at the top displays log event frequency over a period from May 3 to May 18, 2023, with a 12-hour interval auto-refresh setting. Each log entry contains detailed information about Cilium and Tetragon process events, including:

- agent_ephemeral_id
- agent_id agent_name (showing "tetragon-hk5sg")
- agent_type (showing "filebeat")
- agent_version (showing "8.13.3")
- cilium_tetragon_log_node_name (showing "wsp-pc")
- process information including parent and exit process data process flags (showing values like "proctx aufs rootcwd")
- process init_tree status (showing "false") process execution IDs

Figure 20 shows the Elastic Security interface displaying security alerts. The alert originated from the host "vm2-pc" (showing 100% of alerts coming from this host), which corresponds to the VM2-Worker node shown in the previous architecture diagram. The alert has a risk score of 73, indicating it's considered a significant security threat and is labeled "Bash Shell Execution in Metasploitable2 Container" which indicates a potential security breach where someone has executed a bash shell inside a Metasploitable2 container.

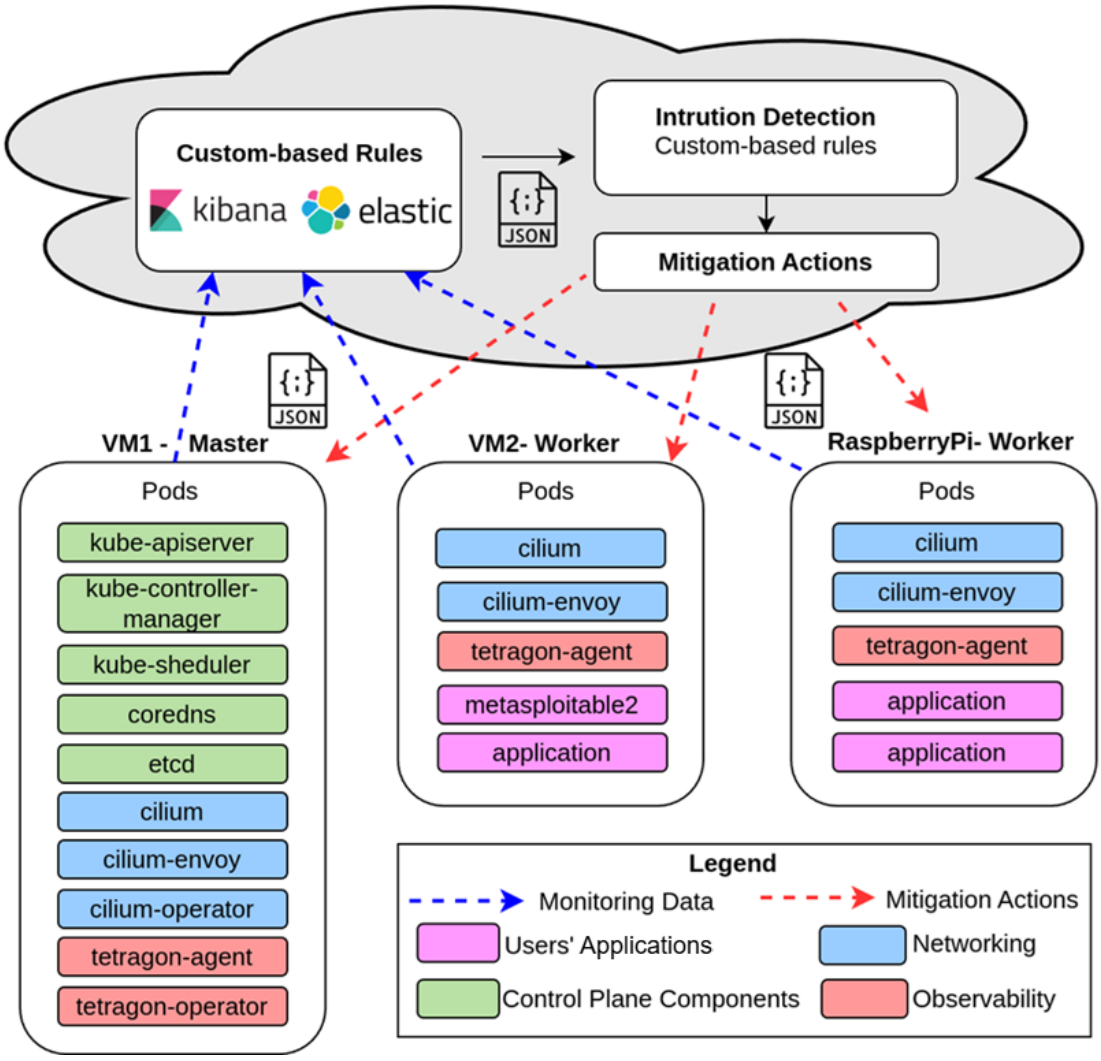

Figure 18. Kubernetes Cluster Security Architecture with Tetragon-based Observability

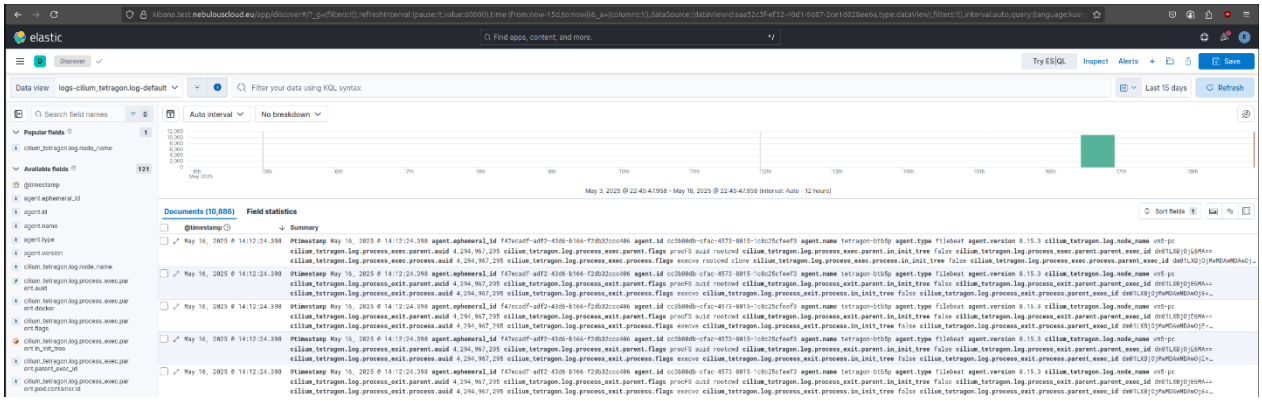

Figure 19. Elasticsearch Kibana Dashboard Displaying Tetragon Security Logs

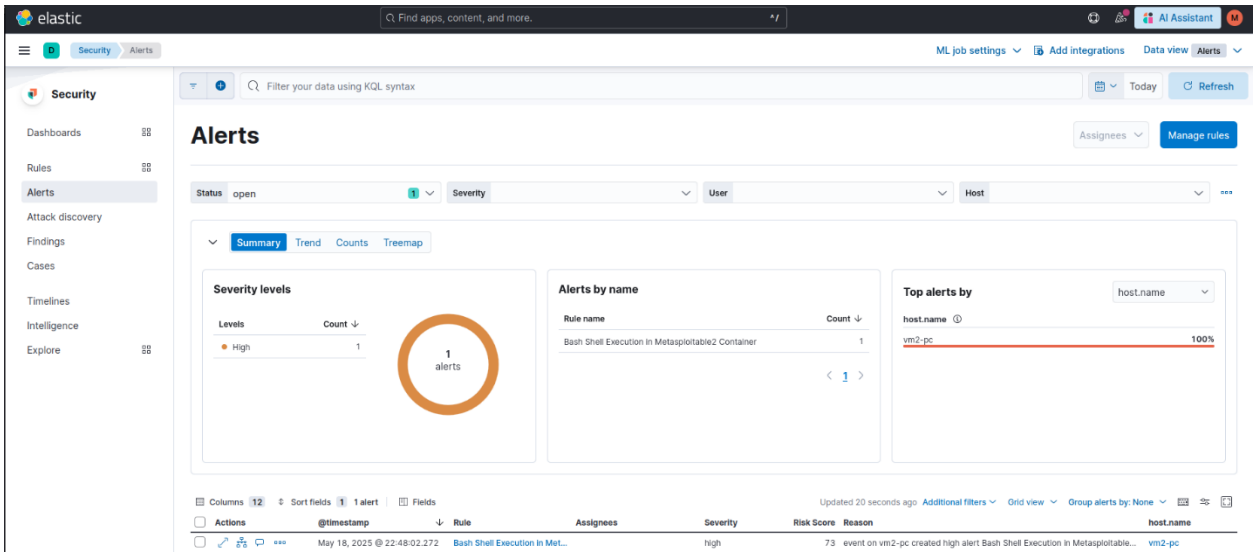

Figure 20. Elastic Security Alert Dashboard Showing Detected Security Threat

6 DIGITAL TWINS ORCHESTRATION IN CLOUD COMPUTING CONTINUUM

Task 4.6 of NebulOuS focuses on advancing the orchestration of digital twins in a way that transcends traditional monolithic models. By developing a framework capable of dynamically aggregating and composing digital twins according to the needs of specific applications and the characteristics of the underlying environment, the project enables a more adaptive and efficient use of cloud and edge resources. This orchestration capability is closely interconnected with the developments of Tasks 4.1, 4.2, and 3.3, forming a foundational part of the NebulOuS platform's vision for seamless, intelligent deployment across heterogeneous infrastructures.

6.1 CHALLENGES OF TRADITIONAL DIGITAL TWINS

Traditional digital twin (DT) concepts originated in industries like manufacturing, automotive, and aerospace, where their primary role is to model and monitor physical assets—such as machines, vehicles, or industrial systems—through real-time data streams capturing physical state, operational parameters, and lifecycle metrics. However, as recent surveys highlight, this paradigm has notable limitations when transposed to digital entities like application deployments. [63][64]

Firstly, traditional DTs typically assume a monolithic structure—a single, well-bounded asset modelled through a static digital replica. Sharma et al. comment that such monolithic reference frameworks "lack a universal reference framework" and demonstrate "domain dependence," making it difficult to repurpose these twins across varied contexts [65]. Modern applications, in contrast, are modular, distributed, and ephemeral: microservices may scale in and out dynamically, be deployed across cloud and edge infrastructure, and update continuously. Static, single-object models therefore fail to accurately represent these shifting topologies.

Secondly, the limited flexibility of classical DTs presents a challenge. As reviewed by Wu et al. (2023), conventional twins are built for known, well-defined scenarios and do not adapt easily to changing

conditions [64]. In dynamic contexts such as cloud-edge, new microservices or replicas may appear unpredictably, requiring an adaptable twin that can grow, shrink, and reconfigure at runtime. Traditional twins lack this agility.

Thirdly, the physical-centric focus of legacy DTs — emphasizing sensor readings like temperature, vibration, or wear — makes them ill-suited for software deployments. Liu et al. identify key DT features as physical mapping and co-evolution over an asset's lifecycle; such features are largely irrelevant when monitoring deployment-level concerns such as latency, resource utilization, message throughput, or service energy usage [66].

Lastly, conventional twins often rely on static or slow-rate data sources (e.g., fixed IoT sensors), whereas digital systems generate highly volatile, dynamic data streams. Knebel et al. (2020) highlight that real-time DTs require careful distribution across cloud-fog hierarchies to cope with latency and scale, pointing to the mismatch between static sensors and dynamic software environments. [67] Service replicas may start and stop rapidly, network conditions shift, and workloads spike unpredictably—static modeling fails to capture these realities.

Taken together, these structural mismatches—a monolithic architecture, limited adaptability, physical-centric data focus, and static sourcing—make traditional digital twins ill-suited for the runtime properties of distributed software deployments. This necessitates a reconceptualization, where twins become modular, data-driven, digitally-native entities capable of reflecting dynamic system architecture, software-specific metrics, and high-frequency events. This transformation is critical for enabling intelligent, context-aware orchestration in platforms like NebulOuS.

6.2 DIGITAL TWINS IN THE NEBULOUS PLATFORM

In the NebulOuS platform, the digital twin concept is reimagined to meet the demands of dynamic, distributed, and context-sensitive digital application deployments. Key adaptations include:

- Modularity and Aggregation: Instead of a single, static twin, the NebulOuS framework supports the aggregation of multiple partial digital twins into a composite representation. Each partial twin can model a specific aspect (e.g., a microservice, a data processing pipeline) of an application, and be dynamically combined based on the deployment context.
- Focus on Digital Metrics: The digital twins are tailored to monitor and model application-specific performance indicators such as resource consumption patterns, response time variability, service dependencies, and workload elasticity, rather than physical properties.
- Dynamic and Context-Aware Evolution: NebulOuS digital twins are designed to evolve alongside their real-world counterparts. They adapt to runtime changes in the deployment environment, such as resource availability, user demand shifts, or failures in the underlying infrastructure.
- Integration with Optimisation and Orchestration: Unlike traditional twins that passively mirror their physical counterparts, NebulOuS digital twins actively feed data into the platform's optimisation mechanisms. They enable continuous learning and decision-making, supporting self-optimising deployments across cloud and edge resources.

By embracing these adaptations, the NebulOuS project ensures that the digital twin paradigm not only remains relevant but becomes a core enabler for the efficient orchestration of next-generation distributed applications.

In the context of NebulOuS, a digital twin is a dynamic, data-driven digital representation of a

deployed application within the cloud-edge continuum. Unlike traditional digital twins that focus on modeling physical assets or static systems, the digital twins in NebulOuS are primarily concerned with capturing and feeding performance data from applications to the platform's optimisation components.

The primary role of the digital twin is to provide real-time and historical performance information about deployed applications to the optimiser developed in Task 3.3. This optimiser uses the data collected by the digital twins to train and refine optimisation models, enabling better resource allocation, deployment adaptation, and performance tuning across heterogeneous cloud and edge environments.

6.2.1 Digital Twin Architecture

The classical structure of a digital twin (see Figure 21) is typically composed of three main layers: System State, System Design and Configuration, and System Behaviour. The System State captures real-time data reflecting the current condition of the physical asset. The System Design and Configuration layer holds the models, parameters, and static configuration data that define the system. Finally, the System Behaviour layer focuses on analysing, simulating, and predicting the asset's behaviour over time. Together, these components support a variety of digital twin services, such as condition monitoring, failure prediction, optimisation analysis, and operational guidance.

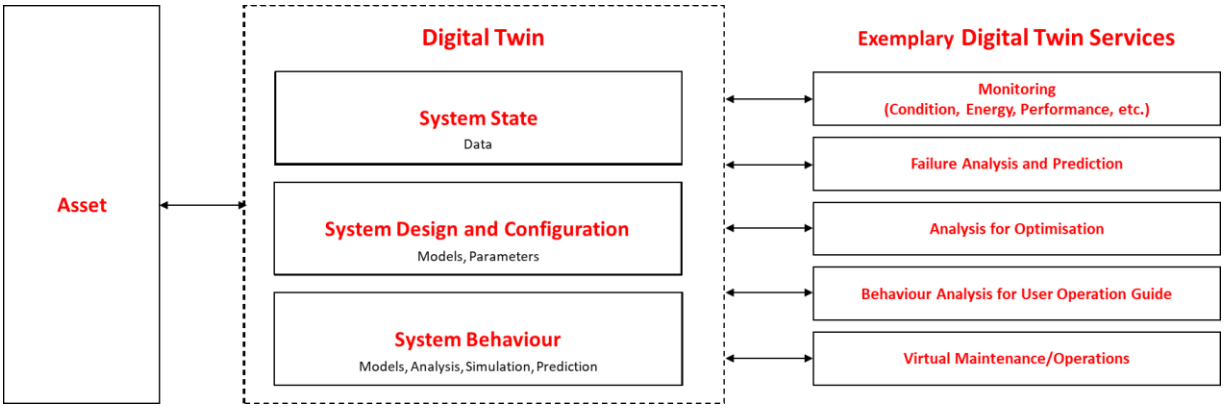

Figure 21. Traditional Digital Twin Architecture [68]

In adapting this general digital twin structure to the needs of dynamic application deployments managed by the NebulOuS platform, the three main parts — System State, System Design and Configuration, and System Behaviour — are still valid as shown in Figure 22.

The **System State** represents the live operational snapshot of the deployed application and the underlying infrastructure. In NebulOuS, this includes real-time infrastructure metrics collected automatically by the platform, such as CPU usage, RAM utilization, disk I/O, and network latency across the cloud-to-edge resources brokered by NebulOuS. In addition, service-level metrics provided by the applications themselves — for example, request processing times and throughput for REST APIs — contribute to describing the application's current performance. These metrics are collected via NebulOuS's integrated monitoring framework, which aggregates both infrastructure metrics and application-specific metrics from the deployed instances and Kubernetes clusters.

The **System Design and Configuration** layer captures the intended architecture and operational setup of the application. Within NebulOuS, this design is specified by the application owner using the Open Application Model (OAM), which describes the service graph, infrastructure requirements, Quality of Service (QoS) targets, and deployment preferences. The OAM definitions include details

such as required CPU and RAM resources, geographical preferences, data locality constraints, and scaling configurations (e.g., minimum and maximum number of service replicas). These specifications are interpreted and processed by NebulOuS modules to determine an optimal deployment plan, considering both the organizational preferences managed by the Organization Admin and the resource offers provided by Resource Providers.

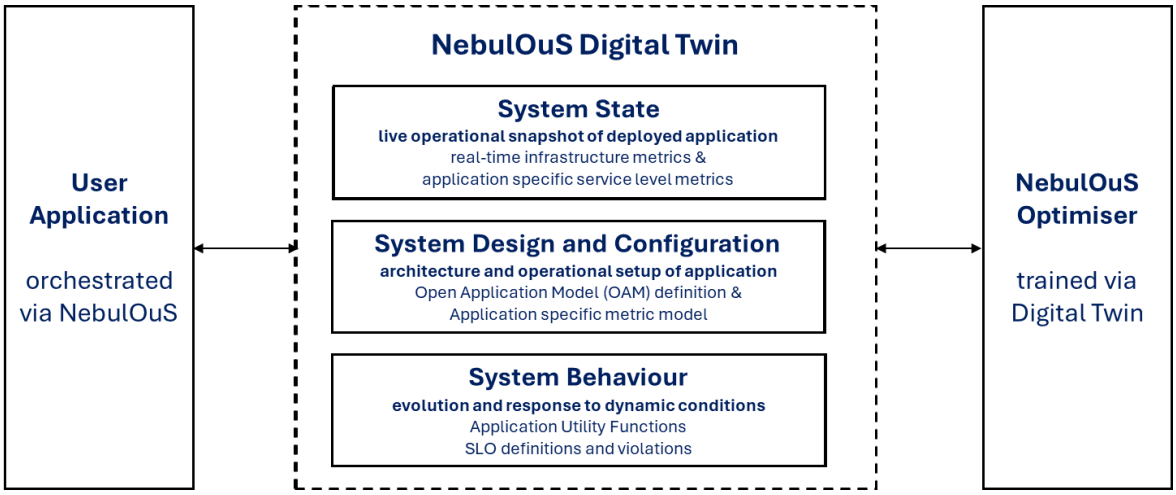

Figure 22: NebulOuS Digital Twin Architecture

The **System Behaviour** describes how the deployed application responds to dynamic conditions and how it evolves over time. In NebulOuS, this behaviour is observed through continuous QoS monitoring performed after deployment. The platform evaluates compliance with SLA objectives, detects QoS violations, and triggers re-optimisation when needed. Behavioural insights include autoscaling decisions (when applications scale out or in), application migrations across nodes (in response to changing conditions), and proactive reconfigurations triggered by predictive QoS degradation models. Furthermore, NebulOuS's optimisation engine (linked to Task 3.3) refines its models based on the monitored data, learning to anticipate resource needs, select better resource placements, and predict future SLA violations. The optimisation-feedback loop ensures that system behaviour is not static but progressively adaptive and aligned with user-defined goals.

6.2.2 Nebulous Digital Twin Application Traces

To enrich the digital twin with fine-grained, execution-level insights, NebulOuS introduces a structured system of application traces. These traces are emitted directly by the deployed application at runtime and are captured via the NebulOuS log collection framework. Designed to provide deep visibility into the dynamic execution of distributed services, these traces complement monitoring and configuration data by adding causal and temporal context to application-level events.

Each trace entry adheres to a standardised schema, capturing key dimensions of an event in the system:

```
{
  "CompName" : "string",          // Kubevela Component Name
  "ReplicaID": "string",          // Unique ID per Component Replica
  "EventType": "in",              // Type of Event: in / out / ack
  "EventTime": 123456789,         // Timestamp in milliseconds
  "PayloadSize": 123,             // Message payload size in bytes
  "ActivityID": "string",         // End-to-end consistent identifier
for flow tracking
```

```
  "RemoteCompName": "string"     // Remote Kubevela Component name
involved in the event
}
```

These traces are produced at various logical points in the application workflow — for example, when a component receives a message (in), sends data (out), or acknowledges receipt (ack). By associating each event with a ReplicaID, the system can distinguish between individual instances of the same component, enabling fine-grained behavioural analysis. The inclusion of a globally consistent ActivityID allows the digital twin to reconstruct end-to-end execution paths, thereby capturing distributed call flows and the causal relationships between events across components.

Traces also capture payload characteristics and communication patterns by logging PayloadSize and RemoteCompName, offering insights into service interactions, data volumes, and potential bottlenecks or latency contributors. Because each event is timestamped (EventTime), the system can analyse temporal properties such as message transit times, inter-component delays, or execution latencies.

By integrating these application-level traces into the digital twin, NebulOuS gains the ability to:

- Reconstruct execution flows across multiple components and replicas.
- Analyse causal dependencies and identify delays or failures in distributed interactions.
- Correlate application logic with infrastructure metrics, closing the loop between abstract execution and physical resource behaviour.
- Support performance debugging, optimisation, and root-cause analysis in real-time or post-mortem evaluation.

Overall, this trace-driven extension enhances the digital twin's ability to not only reflect the state of the deployed application, but also understand how it executes, why it behaves in a certain way, and how its performance and interactions evolve over time.

To enable detailed insight into application execution flows and inter-component interactions, the NebulOuS platform supports a structured mechanism for collecting digital twin application traces. These traces are emitted by the application itself as log entries and are ingested through a centralised logging pipeline for indexing, querying, and analysis.

6.2.3 Logging Interface: Emitting Traces

Applications are expected to emit digital twin trace events to standard output (stdout) using a consistent and identifiable log prefix. Each trace log line must begin with the keyword DTTRACE, followed by a structured, JSON-formatted message containing the trace fields. This approach ensures compatibility with containerised application environments and avoids introducing complex dependencies within the application code.

```
DTTRACE      {"CompName":"payment-service","ReplicaID":"payment-5f678c9b9d-
abc12","EventType":"in","EventTime":1714971615000,"PayloadSize":2048,"Acti
vityID":"txn-12345","RemoteCompName":"frontend"}
```

This log line format satisfies several key criteria:

- Human-readable and machine-parseable, thanks to the JSON structure.
- Easily filterable via the DTTRACE keyword by the logging agent.
- Structured metadata captures the core event properties for downstream indexing and analysis.

To collect and manage these logs, NebulOuS leverages a combination of Fluentd and Elasticsearch:

Fluentd acts as the log aggregation and parsing layer. Running as a daemonset or sidecar in the Kubernetes environment, Fluentd collects stdout logs from application pods, identifies entries containing the DTTRACE prefix, and parses the embedded JSON content into structured fields.

Upon parsing, Fluentd applies a transformation pipeline that:

- Tags and categorizes the log event (e.g. as digitaltwin.trace)
- Validates the structure and required fields
- Optionally enriches the log with metadata such as pod name, namespace, and cluster ID

Elasticsearch serves as the central repository for storing and indexing the parsed traces. This mechanism ensures that application-generated traces are collected efficiently and integrated seamlessly into the broader digital twin analytics pipeline within NebulOuS.

The definitions and examples of the NebulOuS Digital Twin are curated in the GitHub repository reachable at https://github.com/eu-nebulous/digital-twin/, while the NebulOuS digital twin integration into the NebulOuS Optimiser is curated in the following repository: https://github.com/eu-nebulous/optimiser-digital-twin.

7 CONCLUSIONS

This deliverable marks the final iteration of Work Package 4, presenting the complete architecture, implementation, and validation of NebulOuS mechanisms for secure deployment and orchestration of applications across heterogeneous cross-cloud and fog environments. The work detailed in D4.2 reflects the project's commitment to delivering an intelligent, secure, and automation-driven orchestration layer, aligned with the overarching vision of the NebulOuS Meta-Operating System.

The advancements achieved in this phase can be summarised as follows:

- **Robust and Flexible Deployment Architecture**: The Executionware stack—comprising the Deployment Manager (SAL) and the Execution Adapter (ProActive)—has matured into a fully automated, scriptable, and cloud-native orchestration layer. It supports seamless deployment across public clouds, private infrastructures, and edge devices, offering high modularity, traceability, and operational resilience.
- **Smart Contract Integration for SLA Management**: The deliverable introduces a novel mechanism to transform formal SLAs into blockchain-based smart contracts. This enables automated, transparent, and verifiable SLA enforcement across trusted and untrusted infrastructures, paving the way for decentralized trust in application lifecycle management.
- **Secure Overlay Networking and Device Onboarding**: The Overlay Network Manager (ONM), now enhanced with Headscale/Tailscale capabilities, successfully addresses the challenges of secure communication and NAT traversal across distributed nodes. This solution ensures encrypted data exchanges and reliable pod-to-pod communication even in constrained environments.
- **Privacy-by-Design and Runtime Observability**: Through the integration of Kubernetes-native policy enforcement mechanisms (OPA Gatekeeper, Cilium) and eBPF-based runtime observability (Tetragon), the platform enforces security policies at both admission and execution time. This two-tiered model strengthens the platform's trustworthiness and supports compliance with evolving regulatory and operational requirements.

- **Digital Twin-Oriented Orchestration**: Finally, the adoption of a modular, adaptive digital twin framework for application deployment introduces a new layer of contextual intelligence. This dynamic monitoring and feedback mechanism enables real-time optimisation, failure anticipation, and behaviour-aware adaptation across the deployment continuum.

With these developments, WP4 has successfully delivered a production-ready orchestration backbone for the NebulOuS platform. The outcomes of D4.2 will directly feed into the final integration activities in WP6 and the forthcoming validation and demonstration tasks involving project use cases and third-party Open Call adopters.

Future work will focus on integrating these components into the NebulOuS second platform release, enhancing interoperability across project layers, and refining the orchestration logic based on real-world application feedback. These next steps will reinforce NebulOuS's ambition to become a reference architecture for hyper-distributed application deployment in European cloud-edge ecosystems.

8 REFERENCES

[1] D4.1 - Initial Orchestration Layer & Security-enabled Overlay Network Deployment, Christos-Alexandros Sarros, Nikos Papageorgopoulos, Giorgos Kitsos (UBITECH), Michael Benguigui, Ankica Barišić (Activeeon), Radosław Piliszek, Jan Marchel (7Bulls), 20.02.2024

[2] Scheduling Abstraction Layer (SAL), Activeeon, GitHub repository, https://github.com/ow2-proactive/scheduling-abstraction-layer (accessed May 30, 2025)

[3] NebulOuS Deployment Manager (SAL), EU NebulOuS Project, GitHub repository, https://github.com/eu-nebulous/sal (accessed May 30, 2025)

[4] ProActive, Activeeon, GitHub repository, https://github.com/ow2-proactive/ (accessed May 30, 2025)

[5] ProActive documentation, Activeeon, https://doc.activeeon.com/main.html (accessed May 30, 2025)

[6] ProActive, Activeeon, https://proactive.activeeon.com/ (accessed May 30, 2025)

[7] Kubernetes (k8s), Cloud Native Computing Foundation, GitHub repository, https://github.com/kubernetes/kubernetes (accessed May 5, 2025)

[8] K3s, Rancher Labs, GitHub repository, https://github.com/k3s-io/k3s (accessed May 30, 2025)

[9] IAAS connector, Activeeon, GitHub repository, https://github.com/ow2-proactive/connector-iaas (accessed May 30, 2025)

[10] NebulOuS GUI, EU NebulOuS Project, GitHub repository, https://github.com/eu-nebulous/gui (accessed May 30, 2025)

[11] Google Compute Engine (GCE), Google, Official website, https://cloud.google.com/compute (accessed May 30, 2025)

[12] Amazon EC2 (Elastic Compute Cloud), Amazon, Official website, https://aws.amazon.com/ec2/ (accessed May 30, 2025)

[13] OpenStack, Open Infrastructure Foundation, Official website, https://www.openstack.org/ (accessed May 30, 2025)

[14] Apache jclouds, Apache Software Foundation, GitHub repository, https://github.com/apache/jclouds (accessed May 5, 2025)

[15] Microsoft Azure, Microsoft, Official website, https://azure.microsoft.com/ (accessed May 30, 2025)

[16] Azure SDK for Java (Azure API), Microsoft, GitHub repository, https://github.com/Azure/azure-sdk-for-java (accessed May 5, 2025)

[17] AMD64 Architecture, AMD, Official documentation, https://www.amd.com/en/technologies/64-bit-computing (accessed May 30, 2025)

[18] ARMv7 Architecture, Arm Ltd., Official documentation, https://developer.arm.com/documentation/ddi0406/latest (accessed May 30, 2025)

[19] ARMv8 Architecture, Arm Ltd., Official documentation, https://developer.arm.com/documentation/ddi0487/latest (accessed May 30, 2025)

[20] NebulOuS Resource Manager, EU NebulOuS Project, GitHub repository, https://github.com/eu-nebulous/resource-manager (accessed May 30, 2025)

[21] NebulOuS Optimiser Controller, EU NebulOuS Project, GitHub repository, https://github.com/eu-nebulous/optimiser-controller (accessed May 30, 2025)

[22] NebulOuS SAL scripts, EU NebulOuS Project, GitHub repository, https://github.com/eu-nebulous/sal-scripts (accessed May 30, 2025)

[23] KubeVela, Open Application Model (OAM) Project, Official website, https://kubevela.io/ (accessed May 30, 2025)

[24] Postgres-PA Container Image, EU NebulOuS Project, Quay.io repository, https://quay.io/repository/nebulous/postgres-pa (accessed May 30, 2025)

[25] ProActive K8s Node Container Image, EU NebulOuS Project, Quay.io repository, https://quay.io/repository/nebulous/proactive-k8s-node (accessed May 30, 2025)

[26] ProActive Scheduler Container Image, EU NebulOuS Project, Quay.io repository, https://quay.io/repository/nebulous/proactive-scheduler (accessed May 30, 2025)

[27] ProActive K8s Dynamic Node Container Image, EU NebulOuS Project, Quay.io repository, https://quay.io/repository/nebulous/proactive-k8s-dynamic-node (accessed May 30, 2025)

[28] Helm, CNCF – Cloud Native Computing Foundation, GitHub repository, https://github.com/helm/helm (accessed May 30, 2025)

[29] Activeeon SAL Image, Activeeon, DockerHub repository, https://hub.docker.com/r/activeeon/sal (accessed May 30, 2025)

[30] NebulOuS SAL Container Image, EU NebulOuS Project, Quay.io repository, https://quay.io/repository/nebulous/sal (accessed May 30, 2025)

[31] NebulOuS Helm Charts, EU NebulOuS Project, GitHub repository, https://github.com/eu-nebulous/helm-charts (accessed May 30, 2025)

[32] Postman Collection, Postman, https://www.postman.com (accessed May 30, 2025)

[33] Newman, Postman, GitHub repository, https://github.com/postmanlabs/newman (accessed May 30, 2025)

[34] Node.js, OpenJS Foundation, Official website, https://nodejs.org/ (accessed May 30, 2025)

[35] Node Package Manager (npm), npm, Inc., Official website, https://www.npmjs.com/ (accessed May 30, 2025)

[36] Xiao, Yang and Zhang, Ning and Lou, Wenjing and Hou, Y. Thomas, A Survey of Distributed Consensus Protocols for Blockchain Networks, IEEE Communications Surveys & Tutorials, 2020.

[37] Tabatabaei, Mohammad Hossein and Vitenberg, Roman and Veeraragavan, Narasimha Raghavan, Understanding blockchain: definitions, architecture, design, and system comparison, arXiv preprint arXiv:2207.02264, 2022.

[38] Envisioned UAV Communication Using 6G Networks: Open issues, Use Cases, and Future Directions, IEEE Internet of Things Journal, 2020.

[39] ggarwal, Shubhani and Kumar, Neeraj and Tanwar, Sudeep, Blockchain

[40] Wu, Mingli and Wang, Kun and Cai, Xiaoqin and Guo, Song and Guo, Minyi and Rong, Chunming, A comprehensive survey of blockchain: From theory to IoT applications and beyond, IEEE Internet of Things Journal, 2019.

[41] Merlina, Andrea and Vitenberg, Roman and Setty, Vinay, A general and configurable framework for blockchain-based marketplaces, Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, 2022.

[42] Zhang, Kaiwen and Vitenberg, Roman and Jacobsen, Hans-Arno, Deconstructing Blockchains: Concepts, Systems, and Insights, DEBS, 2018.

[43] Belotti, Marianna and Bo{\v{z}}i{\'c}, Nikola and Pujolle, Guy and Secci, Stefano, A vademecum on blockchain technologies: When, which, and how, IEEE Communications Surveys \& Tutorials, 2019.

[44] Xie, Junfeng and Yu, F Richard and Huang, Tao and Xie, Renchao and Liu, Jiang and Liu, Yunjie, A survey on the scalability of blockchain systems,IEEE Network, 2019.

[45] Zhang, Rui and Xue, Rui and Liu, Ling, Security and privacy on blockchain, ACM Computing Surveys (CSUR), 2019.

[46] Szabo, Nick, Formalizing and securing relationships on public networks, First monday, 1997.

[47] Pierro, G. A., Tonelli, R., & Marchesi, M. (2019). Blockchain oracles: A framework for blockchain-based applications. In Business Process Management: Blockchain and Central and Eastern Europe Forum (pp. 19–34). Springer, Cham. https://doi.org/10.1007/978-3-030-30429-4_2

[48] D5.2 “Final Mechanisms for Autonomous Reconfigurations in ad-hoc Cloud Computing Continuums” (ICCS, M33)

[49] Kubernetes Access Control Documentation, Kubernetes Project, Official documentation, https://kubernetes.io/docs/reference/access-authn-authz/ (accessed May 30, 2025)

[50] Extensible Admission Controllers, Kubernetes Project, Official documentation, https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/ (accessed May 30, 2025)

[51] Kyverno, Nirmata, GitHub repository, https://github.com/kyverno/kyverno (accessed May 30, 2025)

[52] Open Policy Agent (OPA), Open Policy Agent Project, Official website, https://www.openpolicyagent.org/ (accessed May 30, 2025)

[53] jsPolicy, jsPolicy Project, GitHub repository, https://github.com/loft-sh/jspolicy (accessed May 30, 2025)

[54] KubeWarden, KubeWarden Project, Official website, https://kubewarden.io/ (accessed May 30, 2025)

[55] Gatekeeper – Policy Controller for Kubernetes, Open Policy Agent Project, GitHub repository, https://github.com/open-policy-agent/gatekeeper (accessed May 30, 2025)

[56] Open Policy Agent (OPA), “Customizing Admission Behavior” (Gatekeeper v3.19 Documentation). OPA Gatekeeper Docs. Available: open-policy-agent.github.io/gatekeeper/website/docs/customize-admission. [Accessed: May 30, 2025].

[57] Open Policy Agent (OPA), “How to use Gatekeeper”. OPA Gatekeeper Docs. [Online]. Available: open-policy-agent.github.io/gatekeeper/website/docs/howto. [Accessed: May 30, 2025].

[58] Open Policy Agent Gatekeeper, "Allowed Repositories (Constraint Template from Policy Library). *OPA Gatekeeper Library Docs*. [Online]. Available: *open-policy-agent.github.io/gatekeeper-library/website/validation/allowedrepos*. [Accessed: May 30, 2025]
[59] Cilium Project, "cilium/cilium – eBPF-based Networking, Security, and Observability". [Online]. Available: *github.com/cilium/cilium*. [Accessed: May 30, 2025]
[60] Cilium Project, "Network Policy (CiliumNetworkPolicy CRD)". [Online]. Available: *docs.cilium.io/en/stable/network/kubernetes/policy/*.
[61] Red Hat, *"Understanding Red Hat OpenShift Logging,"* OpenShift Container Platform 4.8 Documentation. [Online]. Available: https://docs.openshift.com/container-platform/4.8/logging/cluster-logging.html. *[Accessed: May 30, 2025]*.
[62] *"The Importance of Security Logs in Cybersecurity,"* CarPen Rebuild, Jan. 29, 2025. [Online]. Available: https://carpen-rebuild.hr/en/the-importance-of-security-logs-in-cybersecurity/. *[Accessed: May 30, 2025]*.
[63] Fuller, A., Member, S., Fan, Z., Day, C., & Barlow, C. (n.d.). Digital Twin: Enabling Technologies, Challenges and Open Research. https://doi.org/10.1109/ACCESS.2020.2998358
[64] Wu, H., Ji, P., Ma, H., & Xing, L. (2023). A Comprehensive Review of Digital Twin from the Perspective of Total Process: Data, Models, Networks and Applications. Sensors (Basel, Switzerland), 23(19), 8306. https://doi.org/10.3390/S23198306
[65] Sharma, A., Kosasih, E., Zhang, J., Brintrup, A., & Calinescu, A. (2020). Digital Twins: State of the Art Theory and Practice, Challenges, and Open Research Questions. Journal of Industrial Information Integration, 30. https://doi.org/10.1016/j.jii.2022.100383
[66] Yao, J. F., Yang, Y., Wang, X. C., & Zhang, X. P. (2023). Systematic review of digital twin technology and applications. Visual Computing for Industry, Biomedicine, and Art 2023 6:1, 6(1), 1–20. https://doi.org/10.1186/S42492-023-00137-4
[67] Knebel, F. P., Wickboldt, J. A., & de Freitas, E. P. (2020). A Cloud-Fog Computing Architecture for Real-Time Digital Twins. Submitted to Journal of Internet Services and Applications. https://arxiv.org/abs/2012.06118v3
[68] Hribernik, K., Cabri, G., Mandreoli, F., & Mentzas, G. (2021). Autonomous, context-aware, adaptive Digital Twins—State of the art and roadmap. Computers in Industry, 133, 103508. https://doi.org/10.1016/j.compind.2021.103508

CONSORTIUM


UBITECH
ubiquitous solutions


SEERC
SOUTH-EAST
EUROPEAN
RESEARCH
CENTRE


7bulls.com


Activeeon
SCALE BEYOND LIMITS


AUGMENTA
PRECISION AGRICULTURE · REDEFINED


ubiwhere
SUITING THE FUTURE


eurecat


mercabarna


ICCS
ΕΜΙΣΕΥ


existanze
#connectingdots


F6S


TTAnalysis


@fire


Telefónica


BIBA

NebulOuS

A META OPERATING SYSTEM FOR BROKERING HYPER-DISTRIBUTED APPLICATIONS ON CLOUD COMPUTING CONTINUUMS

Funded by the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Directorate-General for Communications Networks, Content and Technology. Neither the European Union nor the granting authority can be held responsible for them.