NebulOuS

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

D4.1 INITIAL ORCHESTRATION LAYER & SECURITY-ENABLED OVERLAY NETWORK DEPLOYMENT

[20/02/2024]

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	INITIAL ORCHESTRATION LAYER & SECURITY-ENABLED OVERLAY NETWORK DEPLOYMENT
Deliverable number	D4.1
Deliverable version	1.0
Contractual date of delivery	31 January 2024
Actual date of delivery	20 February 2024
Nature of deliverable	OTHER
Dissemination level	Public
Work Package	WP4
Deliverable lead	UBI
Author(s)	Christos-Alexandros Sarros, Nikos Papageorgopoulos, Giorgos Kitsos (UBITECH), Michael Benguigui, Ankica Barišić (Activeeon), Radosław Piliszek, Jan Marchel (7Bulls)
Abstract	Early results of task T4.1, and T4.2, on the deployment and management of distributed applications on the cloud-to-edge continuum, particularly focusing on resource pool management. Besides traditional cloud-native workloads, we also present our plan to accommodate serverless applications on the compute continuum. We introduce the details of our mechanism for the automatic establishment of secure overlay networks between the distributed compute resources (T4.4) and detail our approach to provide fine-grained policy-controlled access control mechanisms on the provisioned clusters (T4.5).
Keywords	cloud-edge continuum, fog computing, orchestration, networking, vpn, security, access control, kubernetes.

CONTRIBUTORS

Name	Organization
Christos-Alexandros Sarros	UBITECH
Nikos Papageorgopoulos	UBITECH
Giorgos Kitsos	UBITECH
Michael Benguigui	Activeeon
Ankica Barišić	Activeeon
Radosław Piliszek	7Bulls
Jan Marchel	7Bulls

PEER REVIEWERS

Name	Organization
Paweł Skrzypek	7bulls
Mario Reyes	EUT

REVISION HISTORY

Version	Date	Owner	Author(s)	Changes to previous version
0.1	12/12/23	UBITECH	Christos-Alexandros Sarros	Table of Contents, document outline
0.2	22/12/23	UBITECH	Christos-Alexandros Sarros	Input on Security Policies
0.3	5/1/23	UBITECH	Christos-Alexandros Sarros	Input on Secure Overlay Networks
0.4	12/1/23	Activeeon	Michael Benguigui	Input on Deployment and Orchestration
0.5	17/1/23	7Bulls	Radosław Piliszek, Jan Marchel	Input on Serverless
0.6	19/1/23	UBITECH	Nikos Papageorgopoulos, Giorgos Kitsos	Refined input on Security Policies and Secure Overlay Networks

0.7	26/1/23	UBITECH	Christos-Alexandros Sarros	1st full draft
0.8	6/2/23	7Bulls	Paweł Skrzypek, Radosław Piliszek, Jan Marchel	Reviewers feedback
0.9	15/2/23	UBITECH, Activeeon	Christos-Alexandros Sarros, Michael Benguinui, Ankica Barišić	Revised sections based on reviewers feedback. Extension of the Deployment Manager for cluster deployment/redeployment
1.0	20/2/23	UBITECH	Christos-Alexandros Sarros	Formatting and minor corrections for final version

TABLE OF ABBREVIATIONS AND ACRONYMS

Abbreviation/Acronym	Open form
AE	Activeeon
ABAC	Attribute-Based Access Control
AMD64	AMD 64-bit x86 instruction set architecture
API	Application Programming Interface
ARM	Advanced RISC Machines
AWS	Amazon Web Services
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CPU	Central Processing Unit
CRD	Custom Resource Definitions
CRUD	Create, Read, Update, Delete
DNS	Domain Name Service
EC2	Amazon Elastic Compute Cloud
EU	European Union
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure-as-a-Service
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
ONM	Overlay Network Manager
PA	ProActive
PERM	Policy, Effect, Request, Matchers
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
SAL	Scheduling Abstraction Layer
scp	Secure Copy
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

EXECUTIVE SUMMARY

NebulOuS is a novel Meta-Operating System designed for hyper-distributed applications on the cloud-to-edge continuum. Our vision is to allow for optimal deployment, provisioning and reconfigurations of user applications in resource pools that encompass resources from the far edge to public and private clouds. Leveraging cloud and fog brokerage capabilities, NebulOuS aims to allow for the formation of ad-hoc cloud continuums that are created on-demand and seamlessly exploit edge and fog nodes, in conjunction with multi-cloud resources.

In this context, D4.1 is focused on the initial orchestration layer and security-enabled overlay network deployment of the Meta-OS. Regarding each aspect, we present both the overall approach being followed in NebulOuS and the technical details on its implementation in the context of NebulOuS. In this context, this deliverable reports the first version of the respective software components that implement this functionality (as developed up until M17).

In the rest of the document, we initially dive into the details of deployment and management of distributed applications on the cloud-to-edge continuum, particularly focusing on resource pool management. Besides traditional cloud-native workloads, we also present our plan to accommodate serverless applications on the compute continuum. Moreover, we provide extensive details on networking and security aspects our platform. In particular, we describe the automatic establishment of secure overlay networks between the distributed compute resources and detail our approach to support the creation, deployment and enforcement of user-defined access control policies that secure the provisioned clusters.

1 INTRODUCTION

1.1 DELIVERABLE OBJECTIVES

The main goal of D4.1 is to report on the main aspects related to deployment and orchestration for hyper-distributed applications in the cloud-edge continuum, as realized by the NebulOuS Meta-OS. To this end, we present our prototype system along with its documentation.

In particular, D4.1 details our approach regarding the deployment and management of microservice-based applications and serverless workloads, the formation and bootstrapping of secure overlay networks that enable connectivity between the disparate resources and application components, as well the attribute-based access control mechanisms that we leverage to secure the cloud/edge resources along with the application components.

For each aspect, we delve into the techniques and tools that we use to realize our vision, while also providing details on the implementation of the relevant NebulOuS software components that were developed to provide each functionality.

1.2 RELATION TO OTHER DELIVERABLES AND WPS

This deliverable is part of WP4 "Cross-Cloud and Fog Applications deployment", reporting on the early results of tasks T4.1 "Deployment & Orchestration in heterogeneous environments", T4.2 "Serverless support", T4.4 "Automatic deployment of secure network overlay" and T4.5 "Security and privacy-by-design in data streams propagation". D4.2 "NebulOuS Secure Cross-Cloud and Fog Applications deployment & Orchestration based on smart contracts" - delivered in M30 - will provide the final iteration of the mechanisms described in the present deliverable, after the introduction of smart contracts in the application deployment process.

With respect to the work carried out in other WPs, the present deliverable reports on the implementation of functionalities and software components that are part of the NebulOuS reference architecture, as specified and reported in D2.1 "Requirements and Conceptual Architecture of the NebulOuS Meta-OS". The deliverable is also related to D6.1 "1st Release of the NebulOuS Integrated Platform & Use Case Planning" (to be delivered in M18). While the present report focuses on the internal logic of each component that implements the corresponding functionality, D6.1 will report on the implementation of the integrated platform (placing more focus on the interactions between the various platform components).

1.3 DELIVERABLE STRUCTURE

The present report conforms to the following structure:

- Section 1 introduces the objectives of D4.1, as well as its relation to the project Work Packages, Tasks and other Deliverables.
- Section 2 is concerned with application deployment and orchestration, documenting the resource provisioning layer of NebulOuS. It presents our current approach and provides implementation details regarding the relevant software components, i.e. the 'Deployment Manager' and 'Execution Adapter'. In addition, it outlines our approach to also support the deployment of serverless applications.
- Section 3 focuses on the establishment of secure overlay networks between the heterogeneous resources that form a NebulOuS cluster. In this section, we provide details on the general approach followed and open-source tools being used (e.g. WireGuard), while also reporting on the development

of all relevant NebulOuS software components that implement this functionality, namely the 'Overlay Network Manager' and 'Overlay Network Agent'.

- Section 4 describes the incorporation of user-defined security policies in NebulOuS. We first outline our general approach on Kubernetes cluster access control[1], presenting the open-source tools being used (Casbin, k8s-gatekeeper). Subsequently, we report on the implementation of the respective software component – the NebulOuS 'Security Manager' - that allows for seamless policy management and deployment across all clusters that are provisioned by NebulOuS.

- Section 5 concludes the document.

2 DEPLOYMENT AND ORCHESTRATION IN HETEROGENEOUS ENVIRONMENTS

The NebulOuS Meta-OS allows users to compose, deploy and provision hyper-distributed applications in the cloud-edge continuum. We focus on containerized applications and use Kubernetes (k8s) [1] as our container orchestration technology of choice to deploy the applications on top of the underlying infrastructure. Kubernetes was introduced with application scalability in mind. It emerged as yet another open-source orchestration engine, but it grew to dominate the Cloud market. This can be attributed to its highly flexible framework, capable of deploying, managing, and scaling containerized applications across distributed environments. In other words, it provided a systematic way of deploying the whole ecosystem of a containerized application with attention to portability and interoperability.

We differentiate between two interrelated, but discrete aspects with respect to the NebulOuS application deployment and orchestration functionality: *resource pool management* and *application resource management.* We use the term *'resource pool management'* to refer to the management of the VMs and physical nodes within a compute cluster, and the term '*application resource management'* to refer to the management of the containerized workloads.

In NebulOuS, *application resource management* is handled by Kubernetes which provisions the containerized workloads on top of the underlying compute resources. After a user describes its application graph, the application components are deployed in Kubernetes clusters in the form of containers (k8s Pods). The workloads themselves within a single cluster are, thus, provisioned by Kubernetes which makes sure that the deployments adhere to the desired state at any time. For this reason, this 'local' application orchestration functionality is not depicted by a discrete component in the NebulOuS conceptual architecture since k8s takes over this role.

Resource pool management is the management of cloud and edge compute resources (physical or virtual) that form the resource pool over which NebulOuS applications are deployed. This includes spinning up and managing VMs in cloud providers, onboarding and provisioning edge nodes and private cloud infrastructure to NebulOuS, setting up Kubernetes clusters on top of the available infrastructure offerings and managing them (e.g. scaling the clusters themselves, by adding or removing nodes to a k8s cluster). Since Kubernetes itself has no way to provision the underlying infrastructure, this functionality is implemented by the NebulOuS '*Deployment Manager'* and *'Execution Adapter'* components. The former receives the optimal setup for a deployment by the '*Optimizer'* component while the latter executes the actual plan using the available underlying resources.

The optimization details (e.g. placement) for each user application are decided by the '*Optimizer'* component by considering all the globally available resources and the application requirements and

[1] Specifically, we focus on dynamic admission control, which allows users to enforce custom policies that allow or disallow certain actions on a Kubernetes cluster (https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/)

constraints; after settling on an optimal setup, the deployment itself is enacted by the '*Deployment Manager'* and '*Execution Adapter'* components. Currently, we assume that each user application will be deployed in a different cluster.

At this point, we note that when we refer to a 'single' NebulOuS cluster, this does not equate single-cloud deployments. Instead, NebulOuS can pool resources from different providers to form multi-cloud and cloud-to-edge k8s clusters that span different locations. This is achieved by forming a secure overlay network between those resources, bringing VPC-like functionality to the cloud-to-edge continuum (the process is described in Section 3).

The rest of this Section focuses on presenting the details of the '*Deployment Manager'* and '*Execution Adapter'* components, which have implemented the aforementioned functionality in the context of Task 4.1 "Deployment & Orchestration in heterogeneous environments". We conclude the section by outlining our plan to include support for the deployment of serverless applications in the next release, based on the results of Task 4.2 "Serverless support".

2.1 APPROACH OVERVIEW

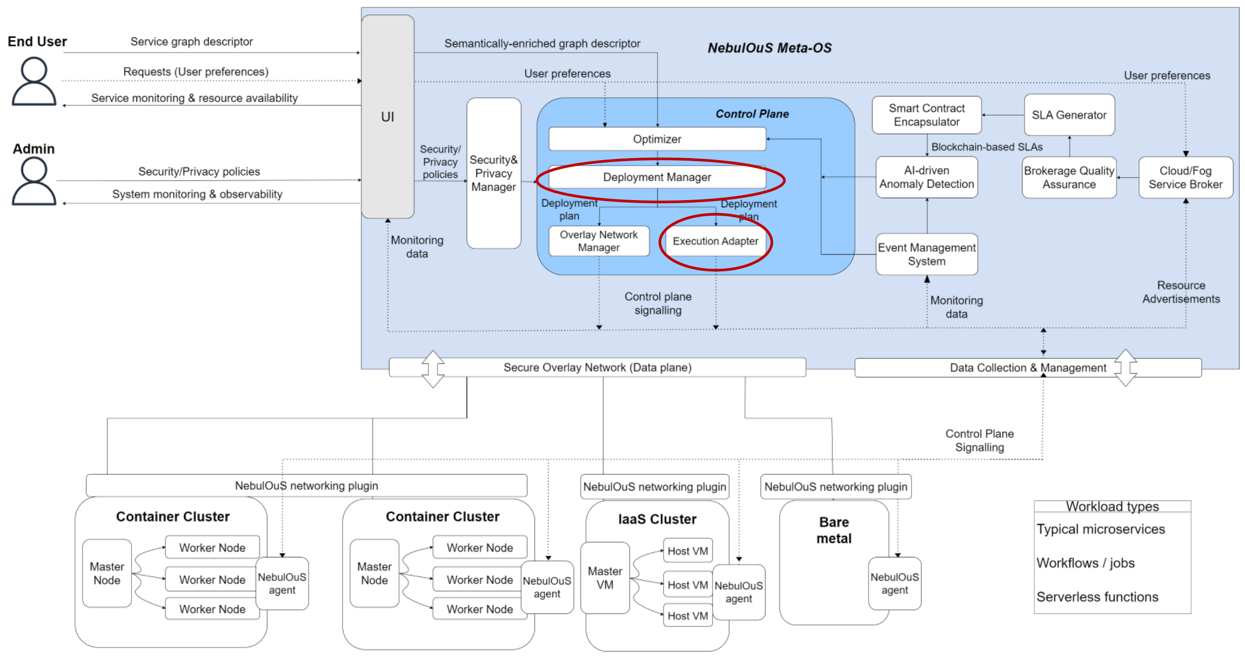


Figure 1: The Deployment Manager and Execution Adapter components, within the NebulOuS architecture

The NebulOuS deployment and orchestration layer builds on Activeeon's ProActive Workflows and Scheduling technology [2]. In particular, the two NebulOuS components that realize this functionality (the '*Execution Adapter' and* '*Deployment Manager'* components are implemented by Activeeon's *ProActive* technology and *Scheduling Abstraction Layer (SAL)* [3].

ProActive, handling the '*Execution Adapter'* role, is the component operating directly with the cloud provider APIs to deploy and manage resources. It allows for the management of heterogeneous resources (Cloud, On-Premise, Edge) thanks to ProActive nodes, offering a homogenization layer between the infrastructure and the component executions. Thanks to that, to orchestrate a multi component application across heterogeneous resources, most of the efforts are required at the ProActive node deployment level instead of at the application submission level. This deployment is achieved thanks to one of the ProActive microservice named IaaS Connector, which includes the main REST endpoints to deploy and manage nodes on any infrastructure.

The '*Deployment Manager'*, also known as SAL, directly interacts with the '*Execution Adapter'* via ProActive REST API to order the resource retrieval according to user constraints (RAM, CPU, cores). It handles the job definition to be submitted, defines the per-task node selection, manages the node candidate caching, and performs additional tasks. This twinning is depicted in Figure 1.

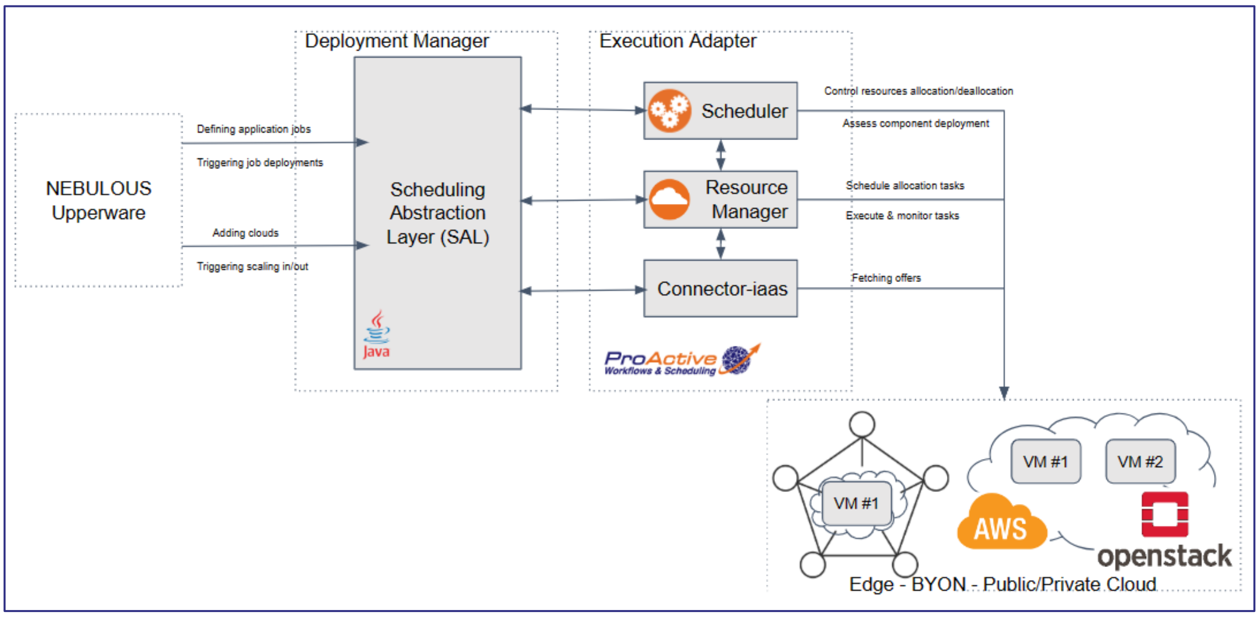

Deployment Manager
Execution Adapter
NEBULOUS Upperware
Defining application jobs
Triggering job deployments
Adding clouds
Triggering scaling in/out
Scheduling Abstraction Layer (SAL)
Java
Scheduler
Resource Manager
Connector-iaas
ProActive Workflows & Scheduling
Control resources allocation/deallocation
Assess component deployment
Schedule allocation tasks
Execute & monitor tasks
Fetching offers
VM #1
VM #1
VM #2
AWS
openstack
Edge - BYON - Public/Private Cloud

Figure 2: NebulOuS resource management

2.2 IMPLEMENTATION

2.2.1 Deployment Manager

The NebulOuS '*Deployment Manager'* component is implemented using Activeeon's SAL. SAL is a Java project exposing the main REST endpoints used to handle the lifecycle of a cluster dedicated to a specific user application. Each endpoint definition is exposed in SAL documentation [3]. For the application developer to run his application on top of NebulOuS, the definition of Cloud providers should be defined via the platform GUI.

The first, preliminary, step when using SAL is to establish a connection to the '*Execution Adapter'* (ProActive) via SAL. This is performed using a *connection* endpoint, to retrieve a *sessionId* according to the user credentials specified. A cloud provider can then be registered using the *add cloud* endpoint. The corresponding sequence diagram is presented in Figure 3:

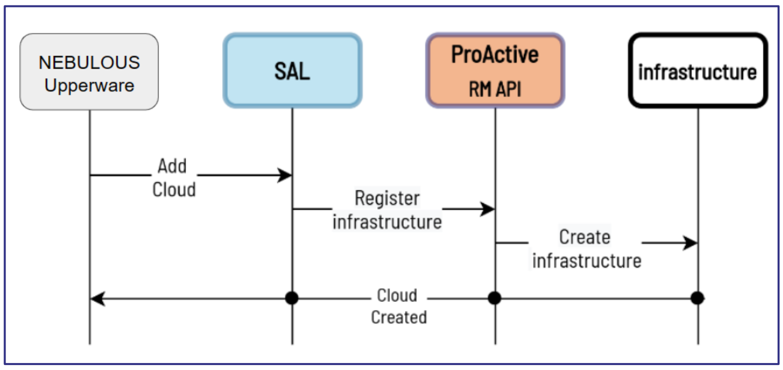


Figure 3: Cloud provider registration sequence diagram

At this stage, the node candidate list can be asynchronously retrieved using the *get node candidates* endpoint which acts according to some user restrictions (e.g. available hardware and their number of cores and memory size, the images, and the regions) and is saved in the local database, as presented in Figure 4.

When calling *filter node candidates* endpoint, feasible combinations will be considered consisting of a combination of hardware, an image, and a region provided by *'Optimizer'* component. All the node candidates are feasible, meaning that the hardware is compatible with the image and can be deployed in the selected region. For instance, EU west 3 can't be used with hardware from EU-west-1, arm64 architecture can't be used with an amd64 image.

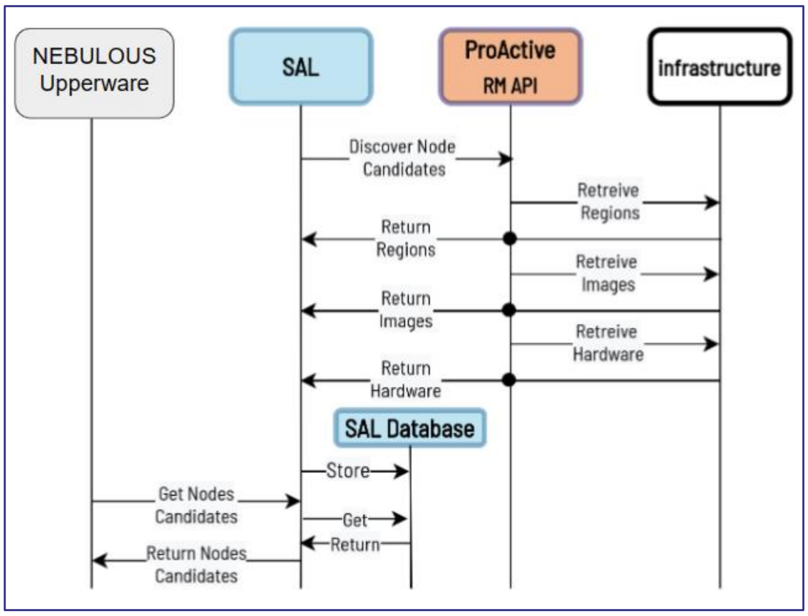


Figure 4 Node candidates sequence diagram

A further idea is to use *define cluster* endpoint, to set how we use node candidates for creating master and worker nodes for a Kubernetes cluster deployment. At a higher representation level, "Cluster.java" is a persisted representation of a cluster in SAL, and includes cluster information: cluster name, the master node name, and a list of *ClusterNodeDefinition*. *ClusterNodeDefinition* is a persisted representation of a node linked to a job to be executed with associated tasks. This representation includes the node name, related node candidate id with its cloud id, related job name and task name.

Each node source centralizes at ProActive level the required information to start VMs and deploy nodes. The REST query body of task flow for deployment is defined following a JSON format [4]. Among other parameters, this JSON describes the provided/required ports, the installation scripts (network configuration, tools installation, etc.) and an application script for each task.

After the cluster is defined, the SAL provides the *deploy cluster* endpoint which creates the cluster and deploys it node by node. At this moment the '*Execution Adapter'* (ProActive) executes the bootstrapping script which sends a REST request to the *'Overlay Network Manager'* component (see Section 3) to install and configure virtual private network (VPN) [5]. Thanks to the generic approach for cluster deployment, the redeployment mechanism is enabled over the *scale-in* and *scale-out* endpoints with support of *'Overlay Network Manager'* component.

Creating Kubernetes clusters manually involves several intricate steps which are being automatized using NebulOuS '*Deployment Manager'* (SAL). Initially, one must provision cloud machines, ensuring they meet Kubernetes' minimum requirements. Subsequently, Kubernetes and its dependencies such as kubeadm, kubelet, kubectl, kubernetes-cni, and a container runtime like Docker engine need to be installed. Network configuration is vital, requiring nodes to communicate either within the same subnet or via configured network routes, security group rules, and firewalls. The control plane is initialized using "kubeadm init" on the master node, followed by connecting worker nodes using "kubeadm join" with appropriate authentication tokens. Kubernetes networking must be set up using solutions like Flannel, Weave, or Calico. Benchmarking ensures proper cluster functionality with commands like "kubectl get nodes" and "kubectl get pods" before finally deploying applications using YAML files. This process demands meticulous attention and a deep understanding of Kubernetes' architecture due to its complexity.

In this solution, we consider the Kubernetes cluster as the application that will be overseen by NebulOuS, which will undertake the provisioning and monitoring of the cluster nodes. To achieve that we define three software components in the deployment type model: "KubeMasterComponent" defining a Kubernetes cluster master provisioned from an OpenStack private cloud, a "KubeWorkerComponent_OS" defining a Kuberenetes worker node provisioned from the same private cloud and finally, "KubeWorkerComponent_aws" defining another Kubernetes worker node provisioned using Amazon Web Services (AWS). The components are defined along with their respective scripts and configurations. We establish the necessary definitions, such as WorkerOSToMaster and WorkerAwsToMaster, to enable communication between workers and the master node.

In addition to the complexity of creating the cluster, Kubernetes is also limited by the scale of this cluster, as it does not provide tools to automatically scale up or down the number of allocated nodes following a surge or a drop in the workload. As a result, sysadmins responsible for managing the cluster are required to manually repeat the previously mentioned steps and then manually add the new node to the cluster. This manual approach creates two significant drawbacks: the first is the operation cost, since the nodes are not removed automatically once the cluster encounters a lower load, this leads to the provision of idle nodes and consequently a financial loss. At the same time, the under-provisioning of the nodes may lead to an overloaded cluster and consequently, performance degradation and service disruptions.

2.2.2 Execution Adapter

The '*Deployment Manager'* (SAL) communicates with the '*Execution Adapter'* (ProActive) thanks to IaaS Connector: a ProActive microservice. This microservice provides on-demand access to computing resources such as servers, storage, and networking, offering more direct control over cloud-based systems. More precisely, IaaS Connector enables to perform CRUD operations on different public or private clouds (AWS EC2 [6], Openstack [7], VMWare [8], Docker [9], etc), and offers REST endpoints to communicate with these infrastructure interfaces, to manage the virtual machines lifecycle. The following schema depicts the global architecture of ProActive with microservices, the most important of which is Connector-IaaS that will be discussed below.

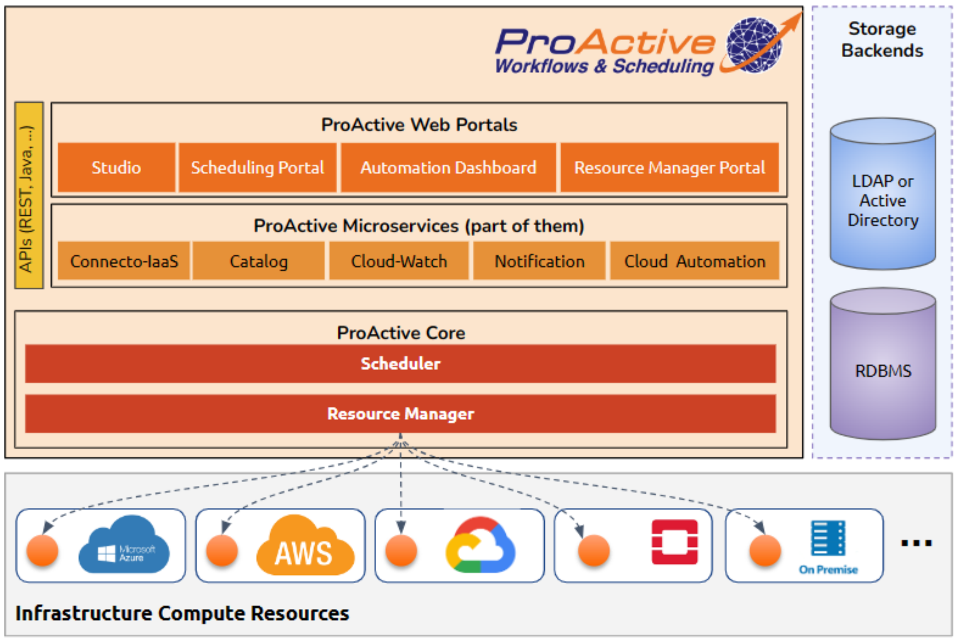


Figure 5: Proactive Architecture

The simplest way to interact with *IaaS Connector* to allocate compute resources is the ProActive *Resource Manager* portal. This latter integrates a Node Source allocation window centralizing all required parameters for each selected infrastructure. The integration with SAL however relies on the REST API.

The IaaS Connector mainly relies on the jclouds toolkit while implementing three main model classes (exposing their own REST endpoints):

- Infrastructure: authentication endpoint, management endpoint...

- Instance: image, hardware, network...

- NodeCandidate: cloud, region, price...

The main cloud provider classes (AWSEC2JCloudsProvider, OpenstackJCloudsProvider, GCEJCloudsProvider) implements the same JCloudsProvider interface including a caching mechanism.

The deployment of ProActive node agents on the compute resources is automatically performed over an SSH connection and a lightweight node.jar.

SAL Code repository	https://github.com/ow2-proactive/scheduling-abstraction-layer
ProActive Project URL	https://github.com/ow2-proactive/scheduling

2.3. SERVERLESS COMPUTING SUPPORT

The scope of serverless computing support in NebulOuS takes into account the project's requirements and goals. Based on this analysis, we have decided to focus on supporting a cloud-agnostic, edge-friendly approach that is well integrated with Kubernetes as NebulOuS's orchestration platform of choice. Additionally, our approach needs to be embracing NebulOuS's values, such as the stance on open source and its community aspects. Following these guidelines, we have identified a single base solution that caters to our needs – Knative [10].

Knative has been accepted to CNCF (Cloud Native Computing Foundation) on March 2, 2022 at the Incubating maturity level. This is the very same foundation that hosts Kubernetes and a wide landscape of compatible tools. This already gives Knative a very strong mandate. We have also analysed the licensing and ecosystem to confirm that indeed it's a tool that aligns with our goals and does not have any downsides with respect to potential adoption.

To better understand the role of Knative in NebulOuS, it is good to quote its description page from CNCF: "*Knative is a developer-focused serverless application layer which is a great complement to the existing Kubernetes application constructs. Knative consists of three components: an HTTP-triggered autoscaling container runtime called "Knative Serving", a CloudEvents-over-HTTP asynchronous routing layer called "Knative Eventing", and a developer-focused function framework which leverages the Serving and Eventing components, called "Knative Functions".".* What follows is that it has three main building blocks: Serving, Eventing, and Functions. Functions integrate the first two as a somewhat higher-level solution. The good integration with Kubernetes is manifested with the management and use of all Knative boiling down to the application of relevant resource manifests which declaratively configure the necessary behind-the-scenes machinery in the Kubernetes-typical control loop fashion. Indeed, we have not found another solution being this "Kubernetes-native" as far as serverless computing solutions go[2].

Due to the time schedule, the first release of NebulOuS does not include full support for serverless computing. Thus, this subsection describes our vision for the adoption and integration of the chosen solution – Knative. Because of the planned character of this, the actual details may change and will be reported in a later deliverable.

The integration of Knative will span both design-time and runtime aspects of NebulOuS. Firstly, in the design-time scope, the existing application modelling approach based on KubeVela will be enhanced to include the possibility to model serverless components of the application – the scale-to-zero functions. This modelling will then be realised at runtime by the extensions to the KubeVela platform to instantiate proper Knative resources. Additionally, care will be taken to properly register these instantiations to the rest of the NebulOuS ecosystem, e.g., monitoring, anomaly detection and security policies.

[2] Notable alternatives that we have checked include Apache OpenWhisk, OpenFaaS, Fission, Fn and Kyma.

3 SECURE NETWORK OVERLAY

Upon selection of the appropriate resources in which the application is deployed, an overlay network is created to securely interconnect compute resources. Our secure overlay allows for the formation of compute clusters that are cross-cloud and can encompass resources from the cloud to the edge, supporting our multi-cloud and cloud-to-edge vision. The secure overlay network that underpins every deployment constitutes an integral part of the NebulOuS security toolbox, leveraging recent advances in VPN technology to offer security and privacy for in-transit application data, regardless of where the application components are deployed. The specific details regarding our approach are outlined below.

3.1 APPROACH OVERVIEW

The NebulOuS Meta-OS interconnects compute resources with one another irrespective of their location, providing effective support, at the connectivity level, for: i) the creation of distributed compute clusters that span from the datacenter to the edge, realizing the vision of a cloud-to-edge continuum and ii) multi-cloud scenarios that can incorporate geo-distributed setups, enabling the seamless management of compute resources from different providers in a single resource pool.

To achieve this, NebulOuS creates a secure network overlay between those resources. This overlay takes the form of a (VPN), which assumes two main functionalities: i) it provides connectivity between the NebulOuS compute resources (physical and/or virtual) and ii) it secures the data in transit by encrypting them. Moreover, network isolation, segmentation, security and privacy are provided by creating separate encrypted virtual networks for the individual compute clusters. Combined, those aspects allow NebulOuS to offer functionality similar to the Virtual Private Clouds (VPC) [11] offered by public cloud providers [12] [13], but on ad-hoc cloud continuums that are formed by the heterogeneous multi-cloud and cloud-to-edge compute resources managed by NebulOuS.

The main component that implements this functionality is the Overlay Network Manager which, during the creation of a new compute cluster by the Execution Adapter, is responsible for bootstrapping the compute resources into a secure overlay network. An on-demand VPN network is created that provides secure node-level connectivity (i.e. VMs or bare metal devices) before the deployment of Kubernetes, ensuring that all intra-cluster traffic will be encrypted for each cluster.

This is especially important, since all communication between the deployed application components within each cluster is secured and remains private, regardless of where the components are deployed. Since a single NebulOuS cluster can span across edge locations, public cloud providers and on-premises private clouds, this arises as a significant consideration. Moreover, the creation of a separate encrypted VPN per-cluster provides isolation between users' applications (especially important since different NebulOuS users can leverage resources from the same providers), offering security and privacy in a multi-tenant, multi-provider heterogeneous environment.

The Overlay Network Manager is the component that automates the setup and management of the different overlays, supplemented by a set of setup scripts. The role of the Overlay Network Manager in the overall NebulOuS architecture is depicted in the Figure below:

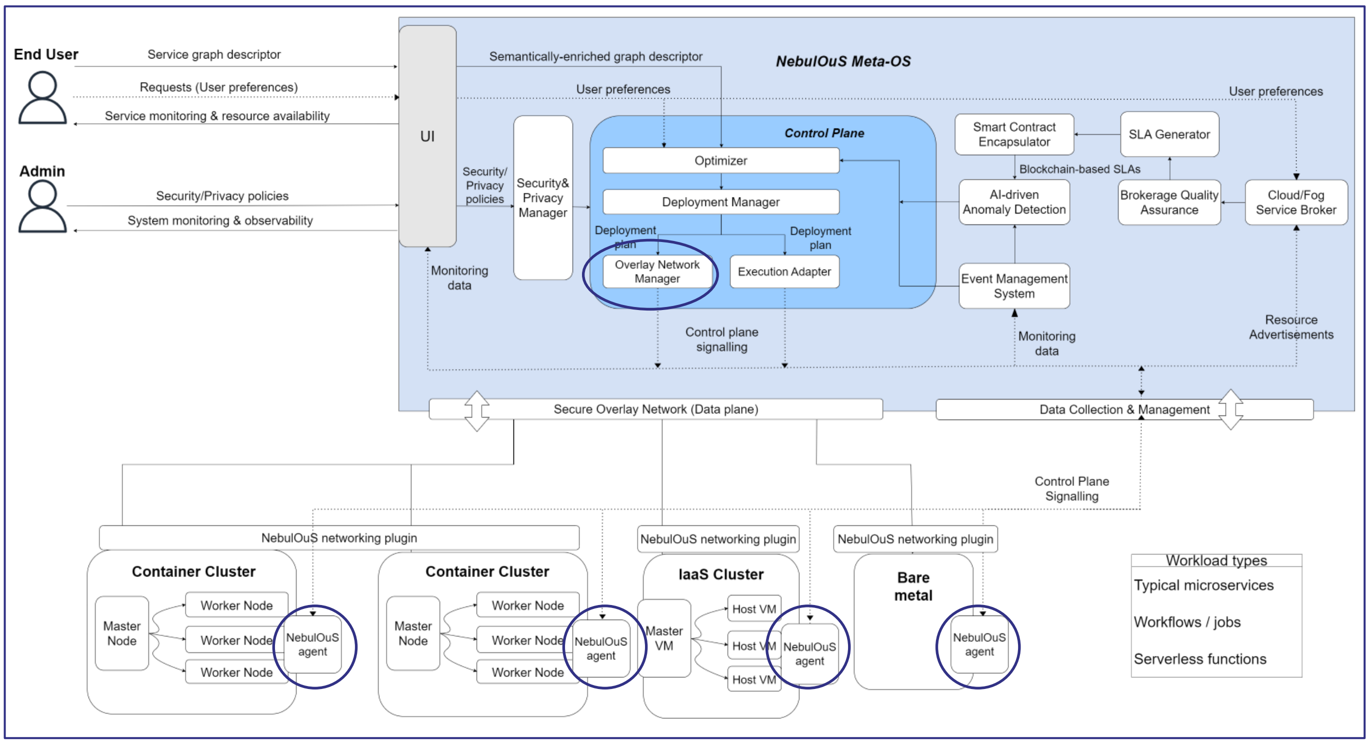


Figure 6: NebulOuS architecture, secure overlay networking components highlighted.

3.2 IMPLEMENTATION

3.2.1 WireGuard VPN

To achieve the on-demand secure overlay creation, we leverage the WireGuard VPN protocol [14] and its open-source implementation. WireGuard encapsulates IP packets over UDP to create a secure Layer-3 tunnel between the connected resources, physical or virtual. It provides a simple yet powerful solution that can act as an alternative to both IPsec and TLS-based solutions (e.g. OpenVPN) [15]. This is achieved through the association of a public key and a tunnel source IP address. Combined with a single-round-trip key exchange mechanism (based on Noise [16] [17]) and mutual authentication of peers (using pre-shared keys), WireGuard allows for transparent session creation and an easy, SSH-like setup.

WireGuard creates its own network interfaces. Each interface has a private key and a list of peers, while each peer has a public key. As a result, each public key is associated with a list of IP addresses that are allowed inside the tunnel. Once a WireGuard interface is added and configured with the private key and the peers' public keys, packets can be securely sent across that interface. The keys are included in configuration files; key distribution is not embedded in the protocol and can be achieved using any out-of-band method – and this is the case also in this project where our Overlay Network Manager distributes these keys.

3.2.1.1 Configuration example

WireGuard calls this approach "Cryptokey routing"[3]. An example of a generic server-client configuration, with respect to the configuration files that are used by the server and client, respectively, is depicted below:

[3] https://www.wireguard.com/#cryptokey-routing

Server configuration file:

```
[Interface]
PrivateKey = yAnz5TF+lXXJte14tji3zlMNq+hd2rYUIgJBgB3fBmk=
ListenPort = 51820

[Peer]
PublicKey = xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
AllowedIPs = 10.192.122.3/32, 10.192.124.1/24

[Peer]
PublicKey = TrMvSoP4jYQlY6RIzBgbssQqY3vxI2Pi+y71lOWWXX0=
AllowedIPs = 10.192.122.4/32, 192.168.0.0/16

[Peer]
PublicKey = gN65BkIKy1eCE9pP1wdc8ROUtkHLF2PfAqYdyYBz6EA=
AllowedIPs = 10.10.10.230/32
```

Client configuration file:

```
[Interface]
PrivateKey = gI6EdUSYvn8ugXOt8QQD6Yc+JyiZxIhp3GInSWRfWGE=
ListenPort = 21841

[Peer]
PublicKey = HIgo9xNzJMWLKASShiTqIybxZ0U3wGLiUeJ1PKf8ykw=
Endpoint = 192.95.5.69:51820
AllowedIPs = 0.0.0.0/0
```

In the server configuration, each peer (a client) will be able to send packets to the network interface with a source IP matching his corresponding list of allowed IPs. For example, when a packet is received by the server from peer gN65BkIK..., after being decrypted and authenticated, if its source IP is 10.10.10.230, then it's allowed onto the interface; otherwise, it's dropped.

When the network interface wants to send a packet to a peer (a client), it looks at that packet's destination IP and compares it to each peer's list of allowed IPs to see which peer to send it to. For example, if the network interface is asked to send a packet with a destination IP of 10.10.10.230, it will encrypt it using the public key of peer gN65BkIK..., and then send it to that peer's most recent Internet endpoint.

In the client configuration, its single peer (the server) will be able to send packets to the network interface with any source IP (since 0.0.0.0/0 is a wildcard). For example, when a packet is received from peer HIgo9xNz..., if it decrypts and authenticates correctly, with any source IP, then it's allowed onto the interface; otherwise, it's dropped.

In other words, when sending packets, the list of allowed IPs behaves as a sort of routing table, and when receiving packets, the list of allowed IPs behaves as a sort of access control list.

3.2.1.2 NebulOuS use

Those features of WireGuard are leveraged in NebulOuS to setup the secure overlay network. For the first release, we have made the following decisions: First, we currently employ split tunnelling[4]; instead of

[4] https://csrc.nist.gov/glossary/term/split_tunneling

forwarding all the traffic of a compute node through the network overlay, we only tunnel Kubernetes traffic. This ensures the isolation of cloud-native applications and encryption of all their traffic, while also allowing any other traffic to/from the compute nodes that is not specific to NebulOuS services to go directly through other interfaces (e.g. the public Internet), instead of routing all traffic via the overlay. This allows i) each compute node to be able to directly access external resources (e.g. downloading images from public repos) without forcing the traffic to go through the VPN first, which adds additional latency and ii) in the Bring Your Own Node scenario, the traffic of non-NebulOuS services running in an edge device are not forced to go through the NebulOuS cluster VPN. This means that a compute resource with a public IP (e.g. cloud provider VMs) will still be accessible through the public Internet with the right credentials; however, we do not expose the node any more than it is already exposed in such a setting. As a result, no additional security measures are needed other than the ones already applied to secure the host (e.g. access credentials, public/private key pairs, cloud provider security groups, firewall rules, etc.).

Second, the VPN is currently set up in a hub-and-spoke topology (following the server-client approach presented above). This means that in a Kubernetes cluster, the node hosting the k8s Master is configured as a WireGuard Server, while the node hosting the k8s Worker is configured to act as a WireGuard client. Using this approach, key management and tunnel establishment are greatly simplified, with the trade-off being that the Worker nodes' pod-to-pod k8s traffic always goes through the Master node that acts as a gateway. Typical pod-to-pod communication within k8s is not altered and the operates as usual, using Cilium[18] and kube-dns[5]. While for most cases this approach is sufficient, for the next release we plan to also support a VPN mesh topology which will allow for the establishment of direct tunnels between Worker nodes (aligning better with the Cilium network topology and investigating potential performance improvements).

The current setup is depicted in Figure 7:

[5] https:// kubernetes.io/docs/concepts/services-networking/dns-pod-service/

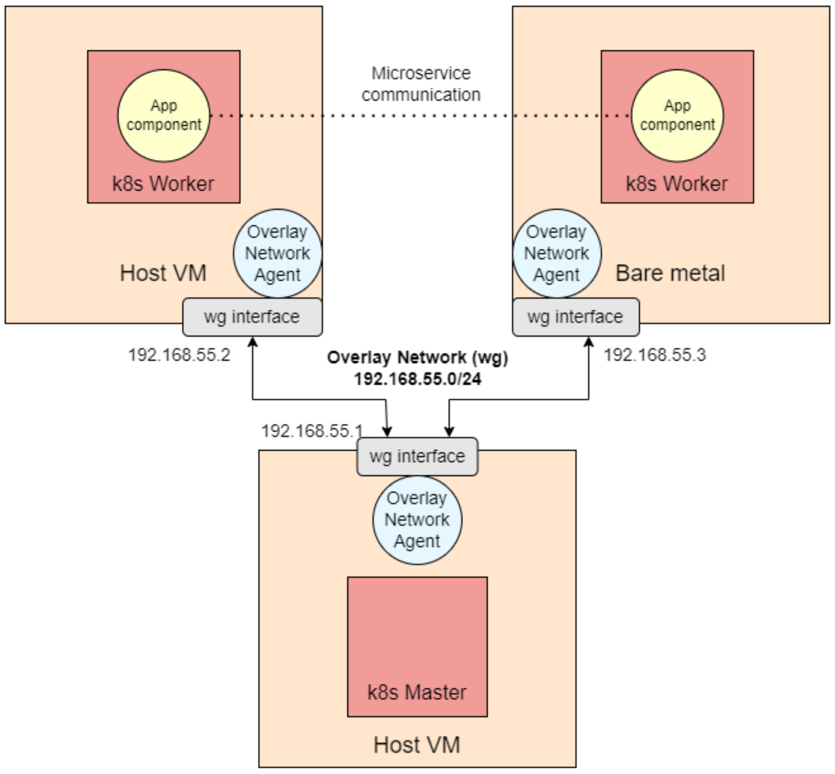


Figure 7: Overlay network in a NebulOuS cluster

From a high-level view, the NebulOuS overlay network management for the various clusters formed by the compute resources onboarded in NebulOuS (private clouds, public clouds, edge devices) is depicted in Figure 8 below:

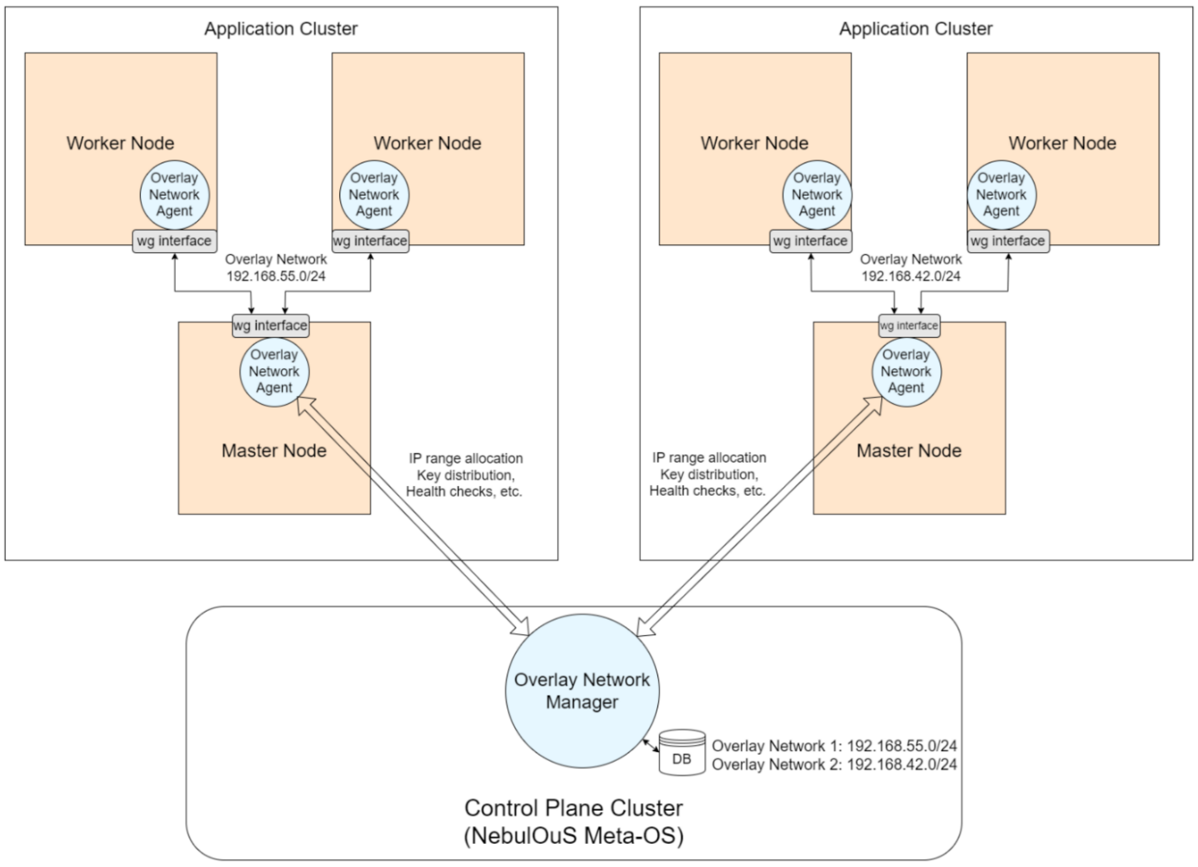

Application Cluster
Worker Node
Overlay Network Agent
wg interface
Overlay Network 192.168.55.0/24
Master Node
IP range allocation Key distribution, Health checks, etc.
Application Cluster
Worker Node
Overlay Network Agent
wg interface
Overlay Network 192.168.42.0/24
Master Node
IP range allocation Key distribution, Health checks, etc.
Overlay Network Manager
DB
Overlay Network 1: 192.168.55.0/24
Overlay Network 2: 192.168.42.0/24
Control Plane Cluster (NebulOuS Meta-OS)

Figure 8: High-level view of NebulOuS overlay network management

3.2.2 Software components

There are two main software components involved in the automatic overlay network setup:

1. The **Overlay Network Manager (ONM)** component is part of the NebulOuS backend service and installed in the control-plane cluster that hosts the NebulOuS platform. The role of ONM is to initiate the creation of a new overlay network in the compute clusters managed by the Meta-OS, centrally managing the addition and deletion of new nodes in the existing overlay networks, the generation and distribution of the respective cryptographic keys, as well as maintaining the overall status of all VPNs that are provisioned at each time by NebulOuS (e.g. health monitoring of the overlay networks).
2. The **overlay network setup scripts** are executed during the initialization process of a compute cluster. They can be thought of as 'setup agents'. There is one main bootstrapping script, which is executed in each NebulOuS-managed resource during its onboarding in NebulOuS. This script initiates the VPN bootstrapping procedure by sending an API call to the ONM component. The ONM then handles the bootstrapping process of each new node in the overlay network. This is done by natively handling issues such as key generation, storage and distribution, also establishing remote SSH connections to execute secondary scripts that install and configure the WireGuard clients in the VM/bare metal devices.

Those components are analysed in the following subsections.

3.2.2.1 Overlay Network Manager.

The Overlay Network Manager is NebulOuS control-plane component that orchestrates the creation, update and deletion of the different overlays created for the NebulOuS-managed compute clusters, while also keeping track of their state at any moment. This also includes key generation and distribution functionality for the WireGuard-based tunnel creation between the peers participating in a VPN. The component itself consists of a backend service and a database that stores the global overlay state information (number of nodes that are part of each VPN, their internal and external IPs, VPN health status, their public and private keys, etc.). The backend service is developed using Java 20 and the Quarkus framework [19] (version 3.2.9.Final), along with a PostgreSQL [20] database. The WireGuard keys are generated using the BouncyCastle Java library[6].

The respective code (both for the scripts and the backend ONM service) is available in the OpenDev project repo [21].

3.2.2.2 Overlay network setup scripts.

The setup scripts are executed in each resource onboarded in NebulOuS. Their purpose is to bootstrap the node (physical device/VM) into an on-demand VPN that is created between the nodes that form a specific compute cluster. A single bootstrapping script is first run by the Execution Adapter component on each machine that is onboarded to NebulOuS. This script initiates the bootstrapping process and sends a call to the Overlay Network Manager, which assumes responsibility for the rest of the process. The ONM establishes remote SSH connections to the specific machines and executes a set of VPN setup scripts. Different scripts are executed based on the node type (Master or Worker nodes).

The rationale behind our choices is that 1) the ONM is the control plane component that centrally manages this process, centrally storing the overlay network state, 2) The WireGuard VPN established in each cluster follows a server/client paradigm with a hub-and-spoke topology, meaning that each VPN consists of a WireGuard server (installed in the cluster Master Node) and several WireGuard clients (installed in the cluster Worker Nodes).

What follows this design, is that the WireGuard server is where the clients' configuration files are generated, while the WG server is also the only node that has all information about all the other nodes (clients) in the VPN. This has two consequences: 1) WG clients need to retrieve their own configuration files from the wg server, and 2) since WireGuard uses mutual authentication to establish communication between two nodes, creating a new WG client also means that the client needs to be separately registered in the WG server.

To orchestrate this set of actions, we have mainly used **SSH** tunnels from the ONM to each machine that acts as a WG server or client to run the necessary scripts in each machine. In a specific case, the script establishes an **scp** connection between the WG client and server, so that the client can retrieve its configuration file.

The scripts executed in each NebulOuS compute node are the following:

- All nodes (script executed by Proactive)
    - **nm-bootstrap-script.sh**
- Master node (scripts executed by the ONM):

a) Creation of a new overlay network
    - **wg-server-create.sh**

[6] https://www.bouncycastle.org/

- **wg-client-create_server.sh**

b) Deletion of an overlay network

- **wg-client-delete_server.sh**

- Worker nodes (scripts executed by the ONM):

a) Creation of a new overlay network

- **wg-client-create_client.sh**

b) Deletion of an overlay network

- **wg-client-delete_client.sh**

3.2.3 Overlay setup.

3.2.3.1 Overlay creation steps.

Bootstrapping new NebulOuS nodes in the VPN of the cluster they are part of, is the first step taken during the node creation workflow of the Execution Adapter component (Proactive).

The overlay network creation flow is depicted in the following Figure:

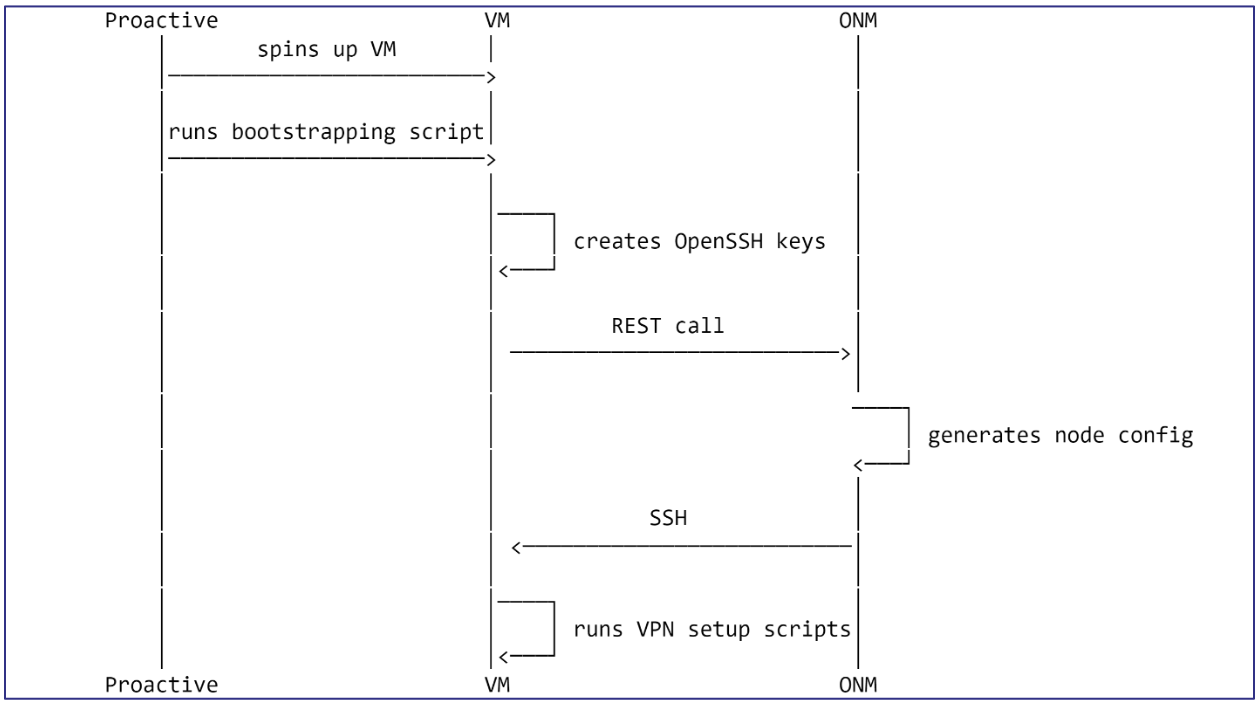


Figure 9: Overlay network creation - sequence diagram

First, Proactive spins up a new VM. In this VM, it executes the overlay network bootstrapping script. This script takes several actions, including the creation of an OpenSSH key pair (so that the ONM can establish an SSH tunnel to the machine to execute the vpn setup scripts). This script then sends a REST call to the ONM which performs all subsequent setup actions.

After receiving this REST call for the creation or deletion of an overlay network node, the ONM implements the logic depicted in the following flowchart:

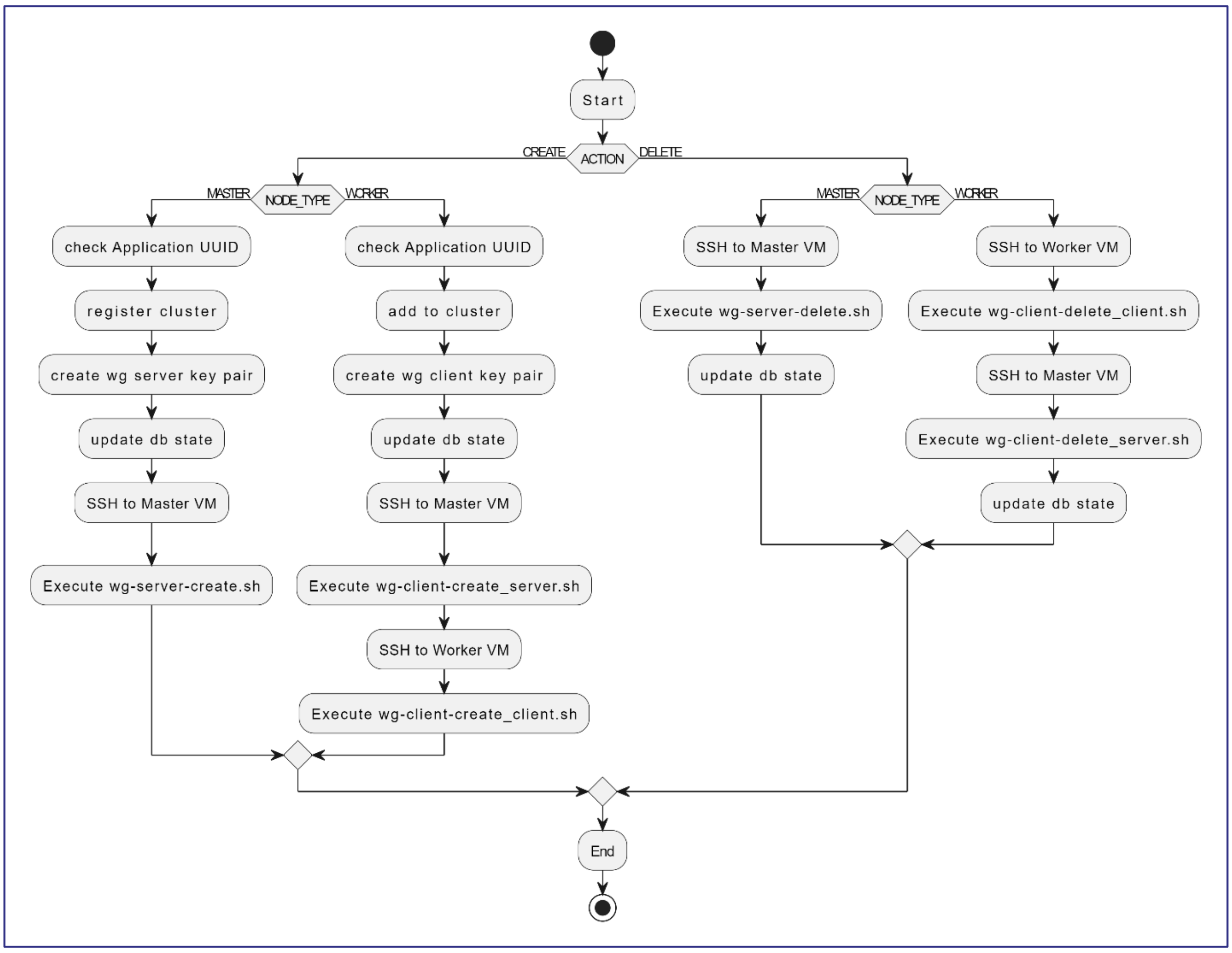


Figure 10: Flowchart depicting the Overlay Network Manager component's internal logic.

Each step in the entire process is analyzed below:

1. When the Execution Adapter creates a new Node, it first executes the overlay network bootstrapping script (**nm-bootstrap-script.sh**) in the Node. This script:
    a. Sends a REST call to the Overlay Network Manager, notifying it regarding the new node.
    b. Creates an OpenSSH public/private key pair.
    c. In the case of a master node, it creates a WireGuard Master Node and in the case of a worker node, it creates a WireGuard Worker Node.
    d. It then adds the node to the correct WireGuard VPN, based on the cluster id (for the 1st release we deploy 1 application per cluster, so we filter the application UUID).
    e. The script uses the following parameters as input, which are provided by Proactive in the form of environmental variables:

i.	**ACTION:**	CREATE/DELETE
ii.	**NODE_TYPE:**	MASTER/WORKER
iii.	**APPLICATION_UUID:**	e.g. 19393adfkjl-sdfkjkj4234-sdlfjk
iv.	**ONM_IP:**	e.g. 1.2.3.4

    f. Example:
    ./nm-bootstrap-script.sh CREATE MASTER 19393adfkjl-sdfkjkj4234-sdlfjk 1.2.3.4
2. After the script is run, the ONM receives a REST call. Depending on the node type (master/worker), the actions taken by the ONM are the following:
    a. In the Master Node case:

The ONM performs an SSH tunnel to the VM and executes the **wg-server-create.sh** script. The script parameters are the following:

i. **WG_SERVER_PRIVATE_KEY**
   This key is created by the ONM and distributed to the node. It is part of the WireGuard configuration file.
ii. **WG_SERVER_PUBLIC_KEY**
   This key is created by the ONM and distributed to the node. It is part of the WireGuard configuration file.
iii. **WG_SERVER_IP**
   This is the internal IP of the node that is onboarded in the VPN. It is created by the ONM and distributed to the node. It is part of the WireGuard configuration file.
   By default, the aforementioned script produces the following configuration:

▪ WireGuard Interface Name:	**wg0**
▪ Listen Port:	**51820**
▪ WireGuard Server IP:	**192.168.55.1**
▪ WireGuard Server Keys Directory:	**/etc/wireguard/server_keys**
▪ WireGuard Server Configuration File:	**/etc/wireguard/wg0.conf**

b. In the Worker Node case:
   The ONM checks in what cluster this node belongs to, using the application UUID. It then:
   i. Creates the WireGuard Node public/private key pairs
   ii. Updates its database state
   iii. Performs two SSH tunnels
      1. The first SSH tunnel is established to the WireGuard Server, to create the configuration file for the WireGuard Client (i.e., the new worker node) and update its own configuration (adding information about the new wg client). This can be done by executing the **wg-client-create_server.sh** script on the server node, with the following arguments:
         a. **WG_CLIENT_NAME**
            Configuration Parameter for the WireGuard Client Configuration file.
         b. **WG_CLIENT_PRIVATE_KEY**
            Configuration Parameter for the WireGuard Client Configuration file.
         c. **WG_CLIENT_PUBLIC_KEY**
            Configuration Parameter for the WireGuard Client Configuration file.
         d. **SSH_USERNAME**
            Username to perform the SSH Tunnel.
         e. **WG_SERVER_PUBLICKEY**
            WG Public key of the Server, needed to create the communication link between the server and client WireGuard nodes.
         f. **SERVER_IP:SERVER_PORT**
            Configuration Parameter for the WireGuard Client Configuration file.
         g. **CLIENT_VPN_IP**
            Configuration Parameter for the WireGuard Client Configuration file.
         h. **ALLOWED_IPS**
            Configuration Parameter for the WireGuard Client Configuration file.

2. The second SSH tunnel is established to the WireGuard Client, to run the **wg-client-create_client.sh** script. This script performs an ***scp*** command to the WireGuard Server Node to get the client node's configuration file which was generated by the server. The conf file is needed to initiate WireGuard communication inside the overlay. The script needs the following arguments:
    a. **WORKER_SSH_USERNAME**
    Username to perform the SSH Tunnel
    b. **OPENSSH_PRIVATE_KEY**
    Master Node OpenSSH Private Key, to perform the SCP command
    c. **SERVER_IP**
    Server IP of the master node, necessary since it provides in which server to perform the SCP command
    d. **CLIENT_NAME**
    WireGuard Client Name. It is the name of the configuration file located on the Master Node.
    e. **MASTER_SSH_USERNAME**
    Username of the Master Node. Needed for the SCP Command

3.2.3.2 Overlay deletion steps.

3. When the Execution Adapter deletes a Node, it executes the overlay network bootstrapping script (**nm-bootstrap-script.sh**) in the Node, setting value of the ACTION argument to 'DELETE'.This triggers the following actions:
    a. The script sends a REST call to the ONM
    b. The ONM checks whether the Node in question is a Master or Worker Node.
    c. Based on the node type, it triggers one of the two flows:
        i. In the Master Node case, it performs one SSH tunnel to the node:
            1. This SSH tunnel is established to execute the **wg-server-delete.sh** script in the machine. This script stops the WireGuard systemd service, removes the WireGuard utilities and configuration files from the machine. After those steps, the machine stops being a part of the cluster's WireGuard overlay network.
            - The script takes the following parameters:
                a. **SSH_USERNAME**
                Used to perform the SSH Tunnel
                b. **WG_INTERFACE_NAME**
                This is the name of the systemd service running the WG interface, which
                needs to be stopped and disabled.
        ii. In the Worker Node case, it performs two SSH tunnels:
            1. The first one is for the WireGuard Client, to execute the **wg-client-delete_client.sh** script in the machine. This script stops the WireGuard systemd service, removes the WireGuard utilities and configuration files from the machine. After those steps, the machine stops being a part of the cluster's WireGuard overlay network.
            - The script takes the following parameters:
                a. **SSH_USERNAME**
                Used to perform the SSH Tunnel
                b. **WG_INTERFACE_NAME**
                This is the name of the systemd service running the WG interface, which
                needs to be stopped and disabled.

The second one is for the WireGuard Server, to execute the **wg-client-delete_server.sh** script in the machine. This script updates the WireGuard Server configuration regarding which clients it is communicating with. It also removes the configuration file of the client that is being deleted. Finally, it restarts its systemd service to apply the aforementioned reconfiguration.

4 SECURITY POLICIES

The second pillar of the NebulOuS security and privacy mechanisms focuses on policy-based access control. This approach allows users to define arbitrary security policies which can then be enforced to control access to resources. Since the NebulOuS Meta-OS uses Kubernetes for container orchestration purposes, our focus is placed on securing access to the Kubernetes clusters.

4.1 APPROACH OVERVIEW

4.1.1 Access Control in Kubernetes

Access Control in Kubernetes is practically realized by controlling access to the Kubernetes API[7]. Kubernetes (k8s) provides a well-structured way to achieve fine-grained cluster access control, using "**admission controllers**".

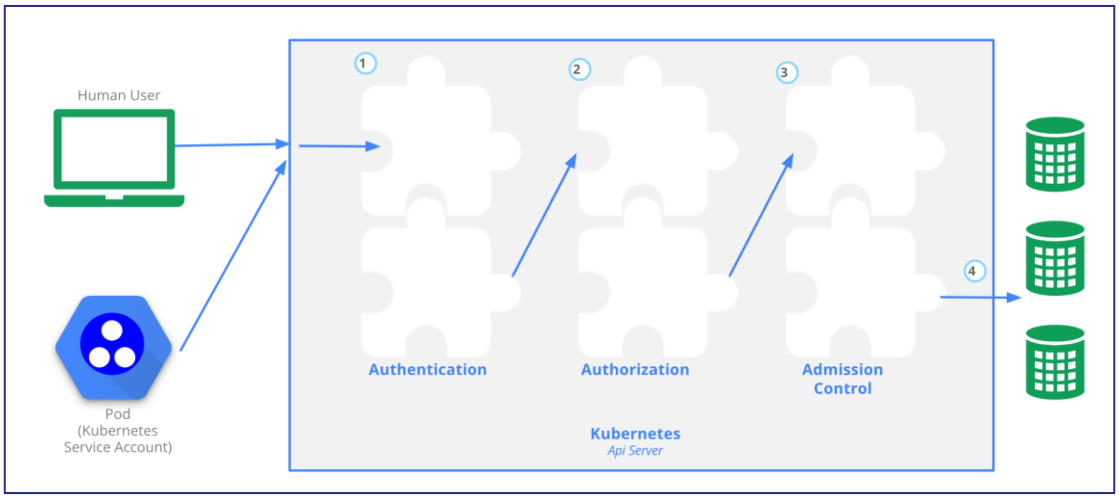


Figure 11: Access Control in Kubernetes

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[8]. They intercept inbound mutation requests. An admission controller can, thus, mutate or reject the requests based on user-defined policies.

[7] https://kubernetes.io/docs/reference/access-authn-authz/
[8] 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 AuthZ.

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[9]. 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 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:

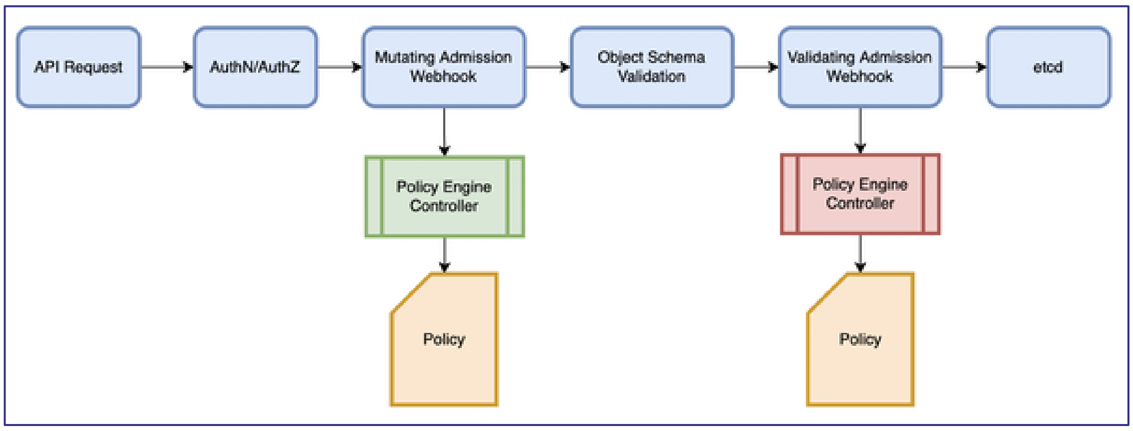


Figure 12: Kubernetes Dynamic Admission Control using external policy engines: flow of an API request

4.1.2 Policy Engine

There are several policy engines that can act as admission controllers for Kubernetes (Kyverno [22], Open Policy Agent [23], jsPolicy [24], KubeWarden [25], etc.). In NebulOuS, we are using the **Casbin** policy engine, specifically its official implementation for Kubernetes (k8s-gatekeeper [26]).

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

Casbin [27] is an open-source access control policy engine that provides support for enforcing authorization based on various access control models. The main reason for our choice is its flexible support of various models (including RBAC and ABAC), even allowing for the definition of custom user-defined models. It can also support the use of XACML-compliant policies via an official translator[10].

Casbin can be used in flows where we want a certain ***object*** or entity to be accessed by a specific user or ***subject***. The type of access, i.e. action, can be read, write, delete, or any other action. This is called the "standard" or classic { **subject, object, action** } flow. Casbin is also capable of handling many complex authorization scenarios other than the standard flow. There can be addition of roles (**RBAC**), attributes (**ABAC**), etc.

To define the access control model, Casbin uses configuration files. The **model** file stores the access model, while the **policy** file stores the specific user permission configuration. An **enforcer** also needs to be created, which loads the **model** and **policy** conf files. Following this model, enforcing a set of rules in Casbin is achieved via:

A **policy** file, which lists subjects, objects, and the desired allowed action (or any other format based on the user needs).

A **model** file, in which the user sets the layout, execution, and conditions for authorization.

An **Enforcer**, which is provided by Casbin for validating an incoming request based on the policy and model files given to the Enforcer.

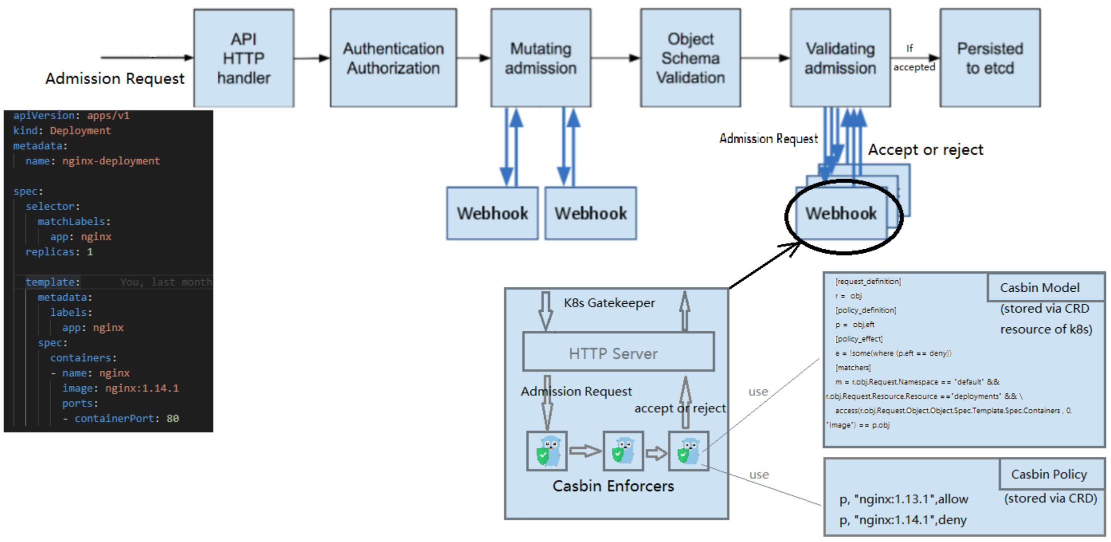
Following this approach, an access control model is abstracted into a CONF file based on the **PERM metamodel** (Policy, Effect, Request, Matchers). These foundations describe the relationship between resources and users.

Switching or upgrading the authorization mechanism is performed simply by modifying a configuration. Custom access control models can be created by users by combining the available models. For example, one can combine RBAC roles and ABAC attributes together inside one model and share one set of policy rules.

4.1.3 Casbin k8s-gatekeeper policies

Casbin can also be used for policy enforcement in Kubernetes, using the *k8s-gatekeeper* implementation. *K8s-gatekeeper* is an admission webhook for k8s which Casbin to apply arbitrary user-defined access control rules to prevent any operation on k8s which the administrator does not desire. It is a Validating Admission Webhook, which means that it decides whether to accept or reject an admission request (an HTTP request describing an operation on specified resources of k8s, such as creating or deleting a deployment).

[10] https://github.com/casbin/xacml-to-casbin-translator/tree/master



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment

spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1

  template:                You, last month
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.1
        ports:
        - containerPort: 80
```

```
[request_definition]
r =  obj
[policy_definition]
p =  obj,eft
[policy_effect]
e = !some(where (p.eft == deny))
[matchers]
m = r.obj.Request.Namespace == "default" &&
r.obj.Request.Resource.Resource =="deployments" && \
access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0,
"Image") == p.obj
```

```
p, "nginx:1.13.1",allow
p, "nginx:1.14.1",deny
```

Figure 13: K8s-gatekeeper as a Validating Admission Webhook

4.1.3.1 Examples

For example, when somebody wants to create a deployment containing a pod running nginx, K8s will generate an admission request, e.g.:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.1
        ports:
        - containerPort: 80
```

This request will go through all the middleware shown in the picture, including K8s-gatekeeper. K8s-gatekeeper can detect all the Casbin enforcers stored in k8s's etcd, created and maintained by the user (via kubectl or a client). Each enforcer contains a Casbin model and a Casbin policy. **The admission request will be processed by every enforcer**, one by one, and **only by passing all enforcers** can a **request** be **accepted** by this K8s-gatekeeper.

For example, if for some reason the administrator wants to forbid the appearance of image *'nginx:1.14.1'* while allowing *'nginx:1.3.1'*, an enforcer containing the following rule and policy can be created:

Rule:

```
[request_definition]
r =  obj

[policy_definition]
p =  obj,eft

[policy_effect]
e = !some(where (p.eft == deny))

[matchers]
m = r.obj.Request.Namespace == "default" && r.obj.Request.Resource.Resource =="deployments" &&
\
access(r.obj.Request.Object.Object.Spec.Template.Spec.Containers , 0, "Image") == p.obj
```

Policy:

```
p, "nginx:1.13.1",allow
p, "nginx:1.14.1",deny
```

By creating an enforcer containing the model and policy above, the previous admission request will be rejected by this enforcer. This means that k8s won't create this deployment.

4.1.3.2 Creating models and policies

When K8s-gatekeeper is authorizing a request, the input is the *go* object of the Admission Request. This AdmissionReview object is defined by k8s' official *go* API[11]. Therefore, for any model used by k8s-gatekeeper, the [request definition] part of a model will always be in the following form:

```
[request_definition]
r =  obj
```

Any name other than *obj* can be used, as long as it is consistent with the name used in the [*matchers*] part.

The ABAC feature of Casbin is used to write down rules. To overcome some limitations present in the expression evaluator integrated in Casbin[12], k8s-gatekeeper provides various extensions[13] in the form of "Casbin functions"[14] that implement these features.

Using Casbin's basic grammar, policies and models can be created. In K8s-gatekeeper, a Casbin model is stored in a Kubernetes CRD called 'CasbinModel'. Similarlyοπ, a Casbin Policy is stored in a Kubernetes CRD, called 'CasbinPolicy'. The definitions of the respective model and policy CRDs are provided in the following files: config/auth.casbin.org_casbinmodels.yaml[15], config/auth.casbin.org_casbinpolicies.yaml[16].

Once a *CasbinModel* and *CasbinPolicy* file is created, they can be deployed and put into effect using *kubectl apply -f <filename>*.

Thus, the enforcement of access control rules in a Kubernetes cluster comes down to creating two relevant objects: a CasbinModel and CasbinPolicy, which define the access model and policy. Several enforcers can be created, meaning that several rules (in the form of model/policy file pairs) can co-exist at the same time.

[11] https://pkg.go.dev/k8s.io/api/admission/v1#AdmissionReview
[12] Specifically, there is no support for indexing in maps, arrays (slices) or array expansion.
[13] https://github.com/casbin/k8s-gatekeeper#421-externsion-functions
[14] https://casbin.org/docs/function/
[15] https://github.com/casbin/k8s-gatekeeper/blob/master/config/auth.casbin.org_casbinmodels.yaml
[16] https://github.com/casbin/k8s-gatekeeper/blob/master/config/auth.casbin.org_casbinpolicies.yaml

A request that causes a change in the cluster must satisfy all of the rules encoded in the different enforcers, before it is persisted in etcd and applied to the cluster.

4.2 IMPLEMENTATION

4.2.1 Security and Privacy Manager

The respective models and policies deployed in NebulOuS clusters are provided by admin users via the NebulOuS GUI. The GUI provides a text editor to write the respective models and policies, which are then serialized in JSON form and sent to the Security and Privacy Manager – the latter is ultimately responsible for deploying them to the cluster.

The NebulOuS *Security and Privacy Manager* component is implemented by a backend service written in Java. This service consists of a client that communicates with the k8s-gatekeper policy engines deployed in each k8s cluster, to offer the necessary functionality for Create/Read/Update/Delete policy operations. It also consists of a PostgreSQL database that stores some pre-defined policies, along with additional information necessary to deploy and manage policies in the NebulOuS-managed clusters (e.g. keep track of which policies are deployed in which cluster, store kubeconfig files necessary for cluster access, etc.).

In general, there are two options to create/update Casbin models and policies using the k8s-gatekeeper engine: either via the CLI (using kubectl) or using a k8s-gatekeeper client implementation. Since the admission policies are managed by the NebulOuS control plane component, we wrote a Java client using Java 17 and Quarkus 3.6.1.

For the first release, the *Security and Privacy Manager* supports CRUD operations for cluster admission rules (Casbin models and policies), using the following methods:

- **createOrUpdateCasbinModel** (CasbinModelDTO casbinModelDTO)
- **listCasbinModels** (String namespace)
- **deleteCasbinModel** (String casbinModelName, String namespace)
- **createOrUpdateCasbinPolicy** (CasbinPolicyDTO casbinPolicyDTO)
- **listCasbinPolicies** (String namespace)
- **deleteCasbinPolicy** (String casbinPolicyName, String namespace)

The aforementioned methods can be found in the *CasbinPolicyResource.java* and *CasbinModelResource.java* files. The DTO used in certain methods can be found in *CasbinModelDTO.java*, and consists of the following fields:

```
public class CasbinModelDTO {
    private String name;
    private boolean enabled;
    private String modelText;
    private String namespace;
}
```

The `createOrUpdateCasbinModel` method is used to create a Casbin model to be applied by k8s-gatekeeper in a specific k8s cluster, by specifying i) the model name, ii) whether the model is enabled, iii) the actual Casbin model text (using Casbin's PERM meta-model format) and iv) the k8s namespace in which the model will be deployed. The same approach is followed by the *createOrUpdateCasbinPolicy* method.

The `listCasbinModels` and `listCasbinPolicies` methods are used to list all the Casbin models and policies in a k8s cluster, under a particular namespace (by specifying this namespace).

The `deleteCasbinModel` and `deleteCasbinPolicy` methods are used to delete a Casbin mode or policy, respectively, by specifying its name and the k8s namespace in which it is applied.

The way our implementation works is the following: when the k8s-gatekeeper policy engine is installed in a k8s cluster, it installs Custom Resource Definitions (CRDs) for the Casbin policies and models. These CRDs extend the default k8s API by enabling Casbin models and policies to be registered and viewed by k8s as Custom Resources, allowing k8s to create respective Objects that can be accessed and manipulated using its API. The *Security and Privacy Manager* component implements a client that communicates with the k8s API (and, indirectly, with k8s-gatekeeper), allowing it to create such resources on a k8s cluster. After their creation, the models and policies are consumed and enforced by the k8s-gatekeeper policy engine that is installed in each cluster. The code for the Security and Privacy Manager component is available in the project's OpenDev repo.

Moreover, to resolve some issues with the original implementation of k8s-gatekeeper, we forked the code and made some minor modifications in the configuration files[17], since the default configurations resulted in installation errors. In particular, our fork includes an installation script (createGatekeeper.sh) that automatically sets up k8s-gatekeeper in each cluster as an internal webhook. In this script, we make a copy of the managed cluster's *kubeconfig* file and create a configmap from this kubeconfig, The *webhook_deployment.yaml* file was modified; we added the created configmap as a volume, which is then mounted by the k8s-gatekeeper Deployment. Last, we modified the *webhook_internal.yaml* file which defines a V*alidatingWebhookConfiguration* object, changing its default name from "*webhook.domain.local*" to "*casbin-webhook-svc.default.svc*" so that it matches the default Certificate Authority (CA). bundled with the default webhook configuration provided by the original repo. We have also updated the README file to keep it up to date.

For the next release, we plan to extend our *Security and Privacy Manager* by adding support for: i) managing policies in multiple k8s clusters and ii) deploying and managing network security policies, using Kubernetes CNI plugins (such as Cilium).

The first extension aims to support NebulOuS' vision of a Meta-OS for the cloud-edge continuum which can deploy and manage applications over different resource pools. In this context, the Security and Privacy Manager control-plane component needs to be able to support multiple applications by multiple users, which are deployed in multiple clusters. With this extension, this single control plane component will be able to keep track of the state of all the different policies deployed in each one of the managed clusters and provide fine-grained security management by communicating with the admission controllers deployed in each cluster.

The second extension aims at extending the range of security policies that can be deployed, leveraging the Cilium CNI plugin. By integrating network policy management into our Security and Privacy Manager, a user will be able to not only manage cluster access control (controlling what is being admitted to a cluster), but also govern network security specifics. This greatly enriches the NebulOuS security capabilities by providing fine-grained control over pod communication, e.g. isolating specific workloads from outside reachability, directing traffic to specific endpoints, etc.

[17] The refined code of the k8s-gatekeeper engine that is deployed in the NebulOuS clusters is available in the following k8s-gatekeeper repository.

5 CONCLUSIONS

In this deliverable, we presented the initial orchestration and deployment layer of the NebulOuS Meta-OS, along with the relevant networking and security aspects. We outlined the general approach and delved into the technical details for each aspect, providing the first version of the respective software components. The components that are documented in this report are included in the first release of the platform.

We first focused on the deployment and management of distributed applications across the continuum from cloud to edge. A particular emphasis was placed on resource pool management, detailing our support for creating and provisioning the available resources in the cloud-edge continuum. We described our approach for deploying microservice-based applications and presented our plans to accommodate serverless workloads. We dived into the specifics of automatically establishing secure overlay networks over those resources and concluded the report by unveiling our approach to define and enforce fine-grained security policies in the provisioned clusters.

Our next steps include enhancements and improvements to the components described in the present report. We are going to refine the Deployment Manager endpoints and Execution Adapter implementation as a result of the integration testing with Overlay Network Manager and Optimizer components, with a focus on better supporting Kubernetes cluster scaling actions and serverless support. The Overlay Network Manager and overlay setup scripts will be refined to include support for peer-to-peer VPN topologies, making the overlay more aligned with certain intra-cluster pod-to-pod communication patterns. Last, the Security and Privacy Manager will be enhanced to also allow for the deployment of Kubernetes Network Policies that can control pod traffic to secure the workloads from external actors, as well as offer support for the deployment of pre-defined ABAC and RBAC policies that can be used as baked-in solutions to improve the clusters' overall security posture.

6 REFERENCES

[1] 'Kubernetes'. [Online]. Available: https://kubernetes.io/
[2] 'ProActive from Activeeon'. Accessed: Feb. 20, 2024. [Online]. Available: https://proactive.activeeon.com/
[3] 'ow2-proactive/scheduling-abstraction-layer'. ProActive, Feb. 21, 2023. Accessed: Feb. 20, 2024. [Online]. Available: https://github.com/ow2-proactive/scheduling-abstraction-layer
[4] 'JSON'. Accessed: Feb. 20, 2024. [Online]. Available: https://www.json.org/json-en.html
[5] H. Abbas *et al.*, 'Security Assessment and Evaluation of VPNs: A Comprehensive Survey', *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–47, 2023.
[6] 'Cloud Compute Capacity - Amazon EC2 - AWS', Amazon Web Services, Inc. Accessed: Feb. 20, 2024. [Online]. Available: https://aws.amazon.com/ec2/
[7] 'Open Source Cloud Computing Infrastructure', OpenStack. Accessed: Feb. 20, 2024. [Online]. Available: https://www.openstack.org/
[8] 'Explore VMware Cloud Solutions', VMware. Accessed: Feb. 20, 2024. [Online]. Available: https://www.vmware.com/cloud-solutions.html
[9] 'Home', Docker Documentation. Accessed: Feb. 20, 2024. [Online]. Available: https://docs.docker.com/
[10] 'Home - Knative'. Accessed: Jan. 26, 2024. [Online]. Available: https://knative.dev/docs/
[11] 'Virtual Private Cloud - an overview | ScienceDirect Topics'. Accessed: Jan. 26, 2024. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/virtual-private-cloud
[12] 'Private Cloud - Amazon Virtual Private Cloud (VPC) - AWS', Amazon Web Services, Inc. Accessed: Jan. 26, 2024. [Online]. Available: https://aws.amazon.com/vpc/
[13] 'Google Virtual Private Cloud (VPC)', Google Cloud. Accessed: Jan. 26, 2024. [Online]. Available: https://cloud.google.com/vpc
[14] J. A. Donenfeld, 'Wireguard: next generation kernel network tunnel.', presented at the NDSS, 2017, pp. 1–12.
[15] S. Mackey, I. Mihov, A. Nosenko, F. Vega, and Y. Cheng, 'A performance comparison of WireGuard and OpenVPN', presented at the Proceedings of the Tenth ACM Conference on data and application security and privacy, 2020, pp. 162–164.
[16] T. Perrin, 'The noise protocol framework', *PowerPoint Presentation*, 2018.
[17] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan, 'Noise: A Library of Verified High-Performance Secure Channel Protocol Implementations', in *2022 IEEE Symposium on Security and Privacy (SP)*, Feb. 2022, pp. 107–124. doi: 10.1109/SP46214.2022.9833621.
[18] 'Cilium - Cloud Native, eBPF-based Networking, Observability, and Security'. Accessed: Jan. 26, 2024. [Online]. Available: https://cilium.io
[19] 'Quarkus - Supersonic Subatomic Java'. Accessed: Jan. 26, 2024. [Online]. Available: https://quarkus.io/
[20] P. G. D. Group, 'PostgreSQL', PostgreSQL. Accessed: Jan. 26, 2024. [Online]. Available: https://www.postgresql.org/
[21] nebulous, 'overlay-network-manager', OpenDev: Free Software Needs Free Tools. Accessed: Jan. 26, 2024. [Online]. Available: https://opendev.org/nebulous/overlay-network-manager
[22] 'Kyverno'. Accessed: Jan. 26, 2024. [Online]. Available: https://kyverno.io/
[23] 'open-policy-agent/gatekeeper'. Open Policy Agent, Jan. 26, 2024. Accessed: Jan. 26, 2024. [Online]. Available: https://github.com/open-policy-agent/gatekeeper
[24] 'Easier & Faster Kubernetes Policies | jsPolicy'. Accessed: Jan. 26, 2024. [Online]. Available: https://www.jspolicy.com/
[25] 'Kubewarden: Kubernetes Dynamic Admission at your fingertips'. Accessed: Jan. 26, 2024. [Online]. Available: https://www.kubewarden.io/
[26] 'casbin/k8s-gatekeeper'. Casbin, Oct. 26, 2023. Accessed: Jan. 26, 2024. [Online]. Available: https://github.com/casbin/k8s-gatekeeper
[27] 'Casbin · An authorization library that supports access control models like ACL, RBAC, ABAC | Casbin'. Accessed: Jan. 26, 2024. [Online]. Available: https://casbin.org/

CONSORTIUM

NebulOuS

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