NebulOuS

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

D5.1 PROTOTYPE OF MONITORING & FOG/IOT DATA STREAMS MANAGEMENT IN CLOUD COMPUTING CONTINUUM

[20/03/2024]

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	Prototype of Monitoring & Fog/IoT Data Streams Management in Cloud Computing Continuum
Deliverable number	D5.1
Deliverable version	Final
Contractual date of delivery	31 January 2024
Actual date of delivery	20 March 2024
Nature of deliverable	Other
Dissemination level	Public
Work Package	WP5
Deliverable lead	ICCS
Author(s)	Yiannis Verginadis (Editor, ICCS), Ioannis Patiniotakis (ICCS), Andreas Tsagkaropoulos (ICCS), Dimitris Apostolou (ICCS), Gregoris Mentzas (ICCS), Dimitra Tzormpaki (ICCS), Fotis Paraskevopoulos (EXZ), Ankica Barišić (AE), Rudi Schlatte (UiO), Geir Horn (UiO), Marta Rozanska (UiO), Paula Cecilia Fritzsche (EUT), Robert Sanfeliu Prat (EUT), Aleix Vila Cano (EUT)
Abstract	This deliverable reports on the 1st iteration of all WP5 related NebulOuS technology. First, it reports on an efficient monitoring mechanism for processing Fog-enabled applications monitoring data. Second, it reports on a distributed IoT architecture for time series and data storage and management, with secure stream data processing capabilities able to run the anomaly detection models, both at the edge, and the cloud. Third, it reports on NebulOuS technology for autonomously driving reconfigurations in transient cloud computing continuum. Last, this deliverable reports on an asynchronous message-based API that drives the communications among NebulOuS components.
Keywords	Cloud Continuum Monitoring, IoT/Fog Data Management, Self-Adaptive Reconfiguration

Funded by the European Union

DISCLAIMER

COPYRIGHT

CONTRIBUTORS

Name	Organization
Yiannis Verginadis	ICCS
Ioannis Patiniotakis	ICCS
Andreas Tsagkaropoulos	ICCS
Dimitris Apostolou	ICCS
Gregoris Mentzas	ICCS
Dimitra Tzormpaki	ICCS
Fotis Paraskevopoulos	EXZ
Ankica Barišić	AE
Rudi Schlatte	UiO
Geir Horn	UiO
Marta Rozanska	UiO
Paula Cecilia Fritzsche	EUT
Robert Sanfeliu Prat	EUT
Aleix Vila Cano	EUT

PEER REVIEWERS

Name	Organization
Rita Santiago	Ubiwhere
Radosław Piliszek	7Bulls

REVISION HISTORY

Version	Date	Owner	Author(s)	Comments
0.1	01/12/2023	ICCS	Yiannis Verginadis	Table of Contents
0.2	15/12/2023	ICCS	Yiannis Verginadis	Preliminary Draft on Chapter 2
0.3	11/01/2024	EUT	Paula Cecilia Fritzsche	Preliminary Draft on Chapter 3
0.4	19/01/2024	EUT	Aleix Vila Cano	Update on Chapter 3
1.0	25/01/2024	ICCS	Andreas Tsagkaropoulos	Draft on Chapter 5
1.1	10/02/2024	UiO	Geir Horn	Draft on Chapter 5
1.2	12/02/2024	EXZ	Fotis Paraskevopoulos	Draft on Chapter 6
1.3	26/02/2024	AE	Ankica Barišić	Draft on Chapter 3
1.4	05/03/2024	All	All	Final version on all Chapters
Final	20/03/2024	All	All	Final version after the internal review

TABLE OF ABBREVIATIONS AND ACRONYMS

Abbreviation/Acronym	Open form
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CEP	Complex Event Processing
CFSB	Cloud Fog Service Broker
DAG	Directed Acyclic Graph
DCA	Dendritic Cell Algorithm
DCEP	Distributed Complex Event Processing
DSL	Domain Specific Language
EIP	Enterprise Integration Patterns
EMS	Event Management System
EPA	Event Processing Agent
EPL	Event Processing Language
EPM	Event Processing Manager
EPN	Event Processing Network
IoT	Internet of Things
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
OAM	Open Application Model
K8S	Kubernetes
NSA	Negative Selection Algorithms

QoS	Quality of Service
SAL	Scheduling Abstraction Layer
SLA	Service Level Agreement
SLO	Service Level Objective
SLOViD	SLO Violation Detector
SSL	Secure Shell
UUID	Universally Unique Identifier
VM	Virtual Machine
VPN	Virtual Private Network
YAML	Yet Another Markup Language

TABLE OF CONTENTS

LIST OF FIGURES

Funded by
the European Union

LIST OF TABLES

EXECUTIVE SUMMARY

This document presents a comprehensive overview of the all research and development activities, under Work Package 5, related to the initial iteration of NebulOuS technology. This technology aims at delivering dedicated functionality that enables autonomous application reconfiguration support in ad-hoc cloud computing continuums (i.e., dynamically changing hosting environments that involve multi-cloud, fog and edge resources).

First, it outlines the creation of an advanced monitoring system for Fog-enabled applications, based on distributed complex event processing, which is crucial for maintaining optimal application performance and resource allocation according to the defined Service Level Objectives (SLOs). In addition, this deliverable reports on the development of an AI-driven anomaly detection for edge computing that combines immunological algorithms with machine learning for focusing on network security aspects. It also covers an interoperable IoT/Fog data management mechanism and an orchestration tool for efficient data stream handling across the cloud continuum. Furthermore, it introduces dedicated NebulOuS components for autonomous application reconfigurations in dynamic cloud environments. Last, we report on an asynchronous message-based API for enhanced communication and interoperability within the NebulOuS Meta-OS system.

1 INTRODUCTION

The document provides a detailed overview of the research and development activities associated with Work Package 5. It focuses on the NebulOuS technology that aims to enhance autonomous application reconfiguration support in ad-hoc cloud computing continuums.

To begin with, it delves into the development of an advanced monitoring system designed to efficiently handle the oversight of Fog-enabled applications, ensuring robust monitoring data collection and processing. NebulOuS introduces a highly effective, and resilient monitoring system designed to offer insightful feedback on the quality of service (QoS) metrics associated with the execution of applications within cloud continuums. This is essential for enabling the automated application reconfiguration support, ensuring optimal performance and resource utilization, according to defined Service Level Objectives (SLOs). Therefore, NebulOuS shall offer a monitoring mechanism based on distributed complex event processing (DCEP) that relies on an unlimited number of distributed monitoring agents that formulate a dynamic and hierarchical network of complex event processing agents.

Following this, the document details the research and development of an AI-driven anomaly detection mechanism at the edge to ensure the application QoS according to the agreed service level agreement (SLA). This framework is specialized for handling time series data and other data types, incorporating advanced machine learning techniques to focus and tackle complex security challenges in diverse environments. Therefore, we explore the potential of combining the immunological algorithm with other machine learning techniques towards developing a hybrid approach that harnesses the strengths of both bio-inspired and traditional methods.

Moreover, as part of WP5, we report on the work on an interoperable IoT/Fog data management mechanism that is able to efficiently manage and propagate applications data streams over a dispersed network of cloud computing continuum resources. This approach also involves an IoT data processing pipelines orchestration tool in order to identify and manage data sources, transformation operations and data consumers in a unified way. This may lead to the capability of defining SLOs that are more data streams management oriented and therefore drive the potential application reconfigurations that may be required.

Furthermore, this deliverable explores innovative solutions that facilitate autonomous application adjustments in the dynamic environment of cloud computing continuum, by leveraging the previously mentioned mechanisms, and allowing for seamless application reconfigurations and optimizations across the computing continuum. Systems like the introduced SLO Violation Detector will exploit monitoring data to consider the potential severity of imminent or potential SLO violations in order to enact application reconfigurations.

Last, the document highlights the implementation of an asynchronous message-based Application Programming Interface (API). This API is instrumental in orchestrating effective communication and interoperability among the various components of the NebulOuS meta-OS system, ensuring a cohesive and efficient technological infrastructure.

2 EFFICIENT MONITORING OF FOG-ENABLED APPLICATIONS

In the fast-evolving world of cloud computing, the tasks of monitoring and efficient metrics analysis for triggering application reconfigurations are undeniably intricate. These tasks demand advanced tools and methodologies such as Complex Event Processing (CEP) systems which excel in digesting and processing a multitude of heterogeneous event streams [1] below. Traditional centralized CEP approaches often necessitate substantial bandwidth and computational power usage in order to aggregate monitoring metrics, while they tend to suffer from a lack of resilience and scalability due to a single point of failure. This is true, especially when dealing with dispersed health status monitoring data. To mitigate such issues, distributed CEP architectures are gaining momentum, particularly in dynamic settings like (ad-hoc) cloud computing continuums [2],[3]. These architectures offer a more resilient and scalable solution by decentralizing the processing of monitoring workloads. However, there's a caveat: such systems are typically reliant on simpler and static or predefined infrastructures. This reliance inherently restricts their ability to uncover potential reconfiguration opportunities across diverse and dynamic environments, including multi-cloud, fog and edge computing resources. This limitation poses a significant challenge in fully leveraging the distributed nature of cloud continuums to optimize application performance and resource utilization.

As part of Task 5.1- Efficient, Secure and Fault-tolerant Monitoring of Fog-enabled applications, NebulOuS provides an efficient, lightweight, fault-tolerant monitoring mechanism for catering feedback on QoS aspects of application execution in cloud continuums. This feedback is required for automatically reconfiguring application placement and for determining SLO and SLA violations. As already mentioned, due to the geographical dispersion and heterogeneity of the resources used for application deployment in modern cloud continuum applications, a centralised monitoring mechanism is insufficient. Therefore, NebulOuS shall offer a monitoring mechanism based on distributed complex event processing that relies on an unlimited number of distributed monitoring agents that formulate a dynamic and federated network of complex event processing agents. These agents are appropriate for the lightweight and advanced processing of real-time monitoring streams, by automatically following the decided topology of application deployment in the cloud continuum. To reduce the amount of monitoring data entering the NebulOuS platform, the monitoring mechanism assesses the health status of resources (e.g., to detect failed ones) and actively seeks reconfiguration opportunities. This monitoring mechanism will be presented in the following subsections of chapter 2, building on top of our previous work on the Event Management System (EMS) [4].

2.1 APPROACH

In NebulOuS, as part of Task 5.1, we significantly extended the EMS system [4] to enable event processing capabilities that among others may reach the edge of the network. Specifically, the NebulOuS fault-tolerant monitoring mechanism introduces a multi-layered distributed complex event processing architecture that automatically follows the cloud and edge infrastructural resources (according to the decisions of the Optimiser) in a fault-tolerant and secure way.

EMS was first introduced as a specialized monitoring framework for multi-cloud applications. EMS's decentralized approach presented significant advantages for multi-cloud environments, through a hierarchical filtering approach that effectively overcomes potential bottlenecks and reduces excessive network bandwidth usage. This means that no single monitoring server should be used to aggregate and process infrastructure and application monitoring health data from dispersed VMs. With a network of agents responsible for collecting and processing data from different cloud sources and vendors, EMS provides valuable insights for informed decision-making, when it comes to reconfiguring multi-cloud applications.

Funded by the European Union

EMS was enhanced, within the Horizon Morphemic[1] project, imbuing the system with self-healing capabilities. This was a significant improvement for bolstering the resilience of this monitoring service across a dispersed number of cloud resources that may span multiple cloud service vendors. These self-healing capabilities are anchored in a decentralized EMS topology orchestration, named as the federated EMS. The approach introduced an innovative monitoring agent clustering methodology.

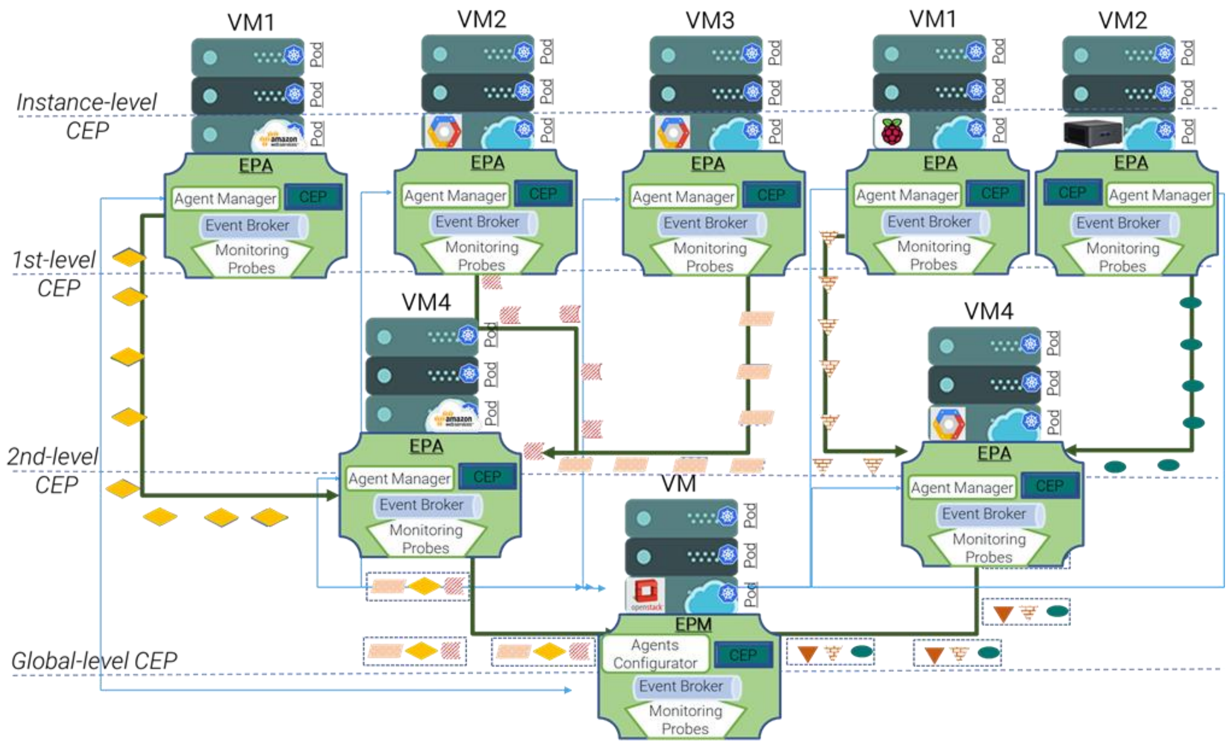


Figure 1: Federated Event Management System (EMS)

In NebulOuS we follow and extend accordingly this monitoring approach, where an Event Processing Manager (EPM), and several Event Processing Agents (EPAs) autonomously manage their monitoring roles, i.e., the most resource capable node autonomously undertakes the role of the "local" aggregator of monitoring data. Furthermore, this monitoring approach can be dedicated to Kubernetes-based deployments over cloud computing continuums. As presented in Fig. 1 the EMS monitoring roles correspond to:

- Instance-level CEP: referring to collecting monitoring metrics from VMs or Kubernetes pods involving mainly the necessary infrastructure, and application-level monitoring probes, along with the metrics propagation to the next level processing node;
- 1st-level CEP: referring to host level monitoring capabilities that "follow" the deployment of application component on cloud, fog or edge resources' virtual machines and/or Kubernetes pods to aggregate monitoring metrics only from the hosting resource;
- 2nd, 3rd ...- level CEP: referring to different levels of monitoring metrics aggregation and processing that make sense per application deployment. These levels may abide to cloud

[1] https://www.morphemic.cloud/

topologies (i.e., cloud availability zone, region or cloud provider) or even be extended to follow meaningful geographical (for the application) areas (i.e. fog and edge locations).

- Global-level CEP: referring to NebulOuS hosting resource where SLOs should be detected to trigger application reconfiguration cycles.

Also, Fig. 1 depicts conceptually the internal structure of an EPA that comprises the following capabilities:

- Monitoring Probes: referring to a subcomponent that is able to measure and publish important for the application QoS, telemetry data;
- Event Broker: referring to an asynchronous message bus that follows the publish/subscribe paradigm to efficiently propagate metrics captured as events throughout the monitoring topology as required;
- CEP: referring to a powerful complex event processing engine that can identify event patterns over signalled time or events windows. The event patterns detected, consolidate composite events, captured as intermediate findings between raw events that may lead to the detection of SLO violations;
- Agent Manager: referring to the necessary functionality for configuring each EPA (i.e., which metrics to monitor and with what sample rates, what event patterns to detect, to which event aggregator should the composite events be propagated to, etc.). This manager also introduces the necessary self-healing functionality, allowing several EPAs to be self-organised in clusters of monitoring nodes, equipped with the capability to replace any failing node or cope with any intermittent connectivity issues.
- Agents Configurator (available only in the EPM): referring to the capability of securely connecting to cloud continuum resources that are to be used for deploying application components. This secure connection allows NebulOuS to install all the necessary subcomponents to EPAs (as mentioned above).

Therefore, EMS is a significant part of the NebulOuS platform as it is able to autonomously deploy and federate an unbounded set of agents, capable of aggregating valuable infrastructure and application level monitoring metrics and gradually propagating them among the different layers of the topology to detect and publish SLOs with the least network bandwidth used, possible. Furthermore, these agents can autonomously orchestrate responses to node failures or any other intermittent connectivity issues, thereby enhancing the overall resilience of the NebulOuS monitoring capabilities.

2.2 EMS CONCEPTUAL ARCHITECTURE

EMS is responsible for establishing a network of agents that is called Event Processing Network (EPN) that collect telemetry data from monitoring probes. Using advanced complex event processing techniques, these monitoring data are filtered and analysed in order to detect SLOs while minimizing the bandwidth used. To properly form this network of monitoring agents, EMS needs to be aware of the application requirements with respect to the desired QoS. As mentioned in D2.2 [5] NebulOuS uses three declarative models for describing cloud continuum applications placement and reconfiguration requirement. Specifically, it uses:

- the Open Application Model as the de-facto standard for describing hyper-distributed applications;
- a model based on AMPL for describing the application components placement constraints and optimisation goals;
- a custom model (based on the metric model from CAMEL [6]) for capturing the QoS requirements associated with hyper-distributed applications and for addressing their monitoring aspects.

The third model, also known as NebulOuS Metric Model, is digested by EMS in order to specify the necessary monitoring data and their processing required per application component, based on the predefined Service Level Objectives (SLOs). EMS is a sophisticated system designed and extended for monitoring cloud continuum applications. It functions as a distributed network, with a central component called EPM (part of the NebulOuS platform) and multiple agent components known as EPAs. Therefore, the core EMS functionalities include the:

- analysis of the NebulOuS Metric Model, given by the application DevOps, in order to extract the required monitoring information per component along with the processing needed.
- secure connection and deployment of EPAs to each application node (i.e., cloud, fog, edge) that hosts an application component (to be monitored).
- secure configuration of each EPA regarding the monitoring sensors that should be used and the complex event processing rules that should be deployed, according to the application component(s) that will be hosted on the same node (according to the decisions of the NebulOuS Optimiser).

Next, the fine-grained core functionalities of EMS, initially provided in [7], are described along with the necessary extensions required to cope with the heterogeneity and dynamicity of cloud continuums. We provide the high-level architectural views of both EPM and EPAs through UML component diagrams depicted in Figures 2 and 3. We begin with the architectural details of EPM that include several sub-components:

- **Metrics Model Translator:** provides a two-step process involving the analysis of the NebulOuS Metric model to produce a multi-root Directed Acyclic Graph (DAG) and also the Generation of EPL[2] rules and other related information. The extensions of this sub-component involve besides the change in the supported domain specific language (DSL) used as input, the underlying object store used, and the ability to deploy multiple complex event processing rules to analyse monitoring metrics coming from multiple components hosted on the same node (e.g., same VM but different Kubernetes pods). In addition, two subcomponents are included:
    - **Validator:** a dedicated service that verifies all the aspects of the received metric model syntax (i.e., not missing core model constructs, all defined raw metrics correspond to available/defined metric sensors etc.)
    - **Persistence:** an appropriate service that persists the incoming metric model and all the exported configuration data (i.e., multi-root DAG, EPL rules) into the underlying file system.
- **Client Installer:** provides the necessary instructions on how to install an EPA to the application nodes, defined in the application deployment model. It comprises three sub-components, the Configurator and two Installers:
    - **Configurator:** prepares the EPAs configuration that includes metrics to collect, metric processing rules (as CEP rules), propagation rules, and also neighbouring EPAs (for clustering), cluster credentials etc.
    - **VM Installer:** is used to deploy EPAs in application VMs. It securely connects to the new application VMs using SSH protocol, and executes a number of predefined jobs, captured as sets of OS-specific instructions. These instructions coincide to the

[2] EPL corresponds to the Event Processing Language used by the ESPER CEP engine (https://www.espertech.com/esper/) that is deployed along with each EPA.

  installation and configuration of the EPA, and the Monitoring Probes that correspond to a Netdata[3] agent.
  - **Kubernetes (K8S) Installer:** is used to deploy EPAs in an application Kubernetes cluster. In this case, it contacts Kubernetes API server and deploys EPAs and monitoring probes as a Kubernetes daemonsets[4]. The settings required for bootstrapping EPAs, are stored as configmaps[5] or secrets.
- **Baguette Server:** undertakes the deployment of the Event Processing Network. Specifically, it designates EPAs installed in each cloud continuum resource, to the appropriate grouping, by sending the corresponding configuration. It also collects VM's or pod's identification information sent from EPAs. The Baguette server encapsulates:
  - a **Node Registry:** that is a SSH server used to accept incoming connections from EPAs. These connections are used to send configurations or other commands to EPAs but also receive cluster related information from the EPAs (e.g., which EPA was elected as the aggregator).
  - **Topology Coordinator:** that undertakes the assignment of configurations to the respective EPAs according to the translated metric model.
- **Control Service**: coordinates and oversees all the EPM operations. It also interacts with the NebulOuS Control Plane and furthermore offers a few EPM management and debugging functions.
- **Broker-CEP Service:** encapsulates an **event message broker** instance and a **CEP engine** instance, hence Broker-CEP provides event propagation, based on the publish subscribe paradigm, and complex event processing capabilities. The **Consumer** sub-component depicted inside Broker-CEP is used to forward the event broker messages into the CEP engine in order to check if they match one or more of the deployed CEP rules.
- **Local Topology Manager:** introduces an EPM in the local cluster of EPMs, while it maintains a view of the available EPMs in the cluster. Only one of them can be considered as the active server (i.e., the Global-level Aggregator) that EPAs are aware of and can interact with. The rest of them are considered stand-by servers. If the active EPM becomes unavailable, Local Topology Managers of stand-by servers will select a new active EPM, using the Aggregator selection protocol described in Section 4.2.4.1 [7]. The new active EPM will reconfigure the EPAs with its address and credentials. Active EPM replicates its current state (including configuration and information of EPAs) to stand-by servers or persists it to a common storage so that if any stand-by server becomes the active one it will be able to recover the work state of the previous active server.
- **Plugin Framework:** corresponds to a programmatic API that enables the registration and interaction of plugins with the EPM. Plugins extend the EPM by introducing new capabilities. For instance, a **Health Checker** plugin is bundled with an EPM, which can check the availability of other (stand-by) EPMs. Health Checker plugins inform the Local Topology Manager when other EPMs malfunction or become inaccessible. Moreover, through this plugin framework a **NebulOuS Middleware Interface** have been developed that allows the communication of EPM with the middleware AMQP broker. This communication permits for example EPM and specifically the Metrics Model Translator to receive a new metric model

[3] https://www.netdata.cloud/

[4] https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[5] https://kubernetes.io/docs/concepts/configuration/configmap/

once the NebulOuS user provides a new application description or propagate to the rest of NebulOuS components an alert concerning SLOs.

- **Subscriptions Advisor:** propagates to the appropriate NebulOuS component a list of event topics that they should be subscribed to according to the applications metric model and utility function. This list may involve raw, composite and/or predicted metrics. For instance, the Severity-based SLO violation detector (presented in section 5) needs to know which composite and predicted metrics should be subscribed to in order to calculate the potential severity of an imminent SLO violation.
- **K8S Pods & Nodes Watcher**: connects to the Kubernetes API server to frequently query for changes in the cluster pods and pinpointing pods across the different cluster nodes to maintain a real-time view of the cluster that should be known by the Baguette Server.
- **Configuration Manager:** baguette server and broker-CEP caches configuration details regarding EPAs that should be provided to kubernetes installer.
- **Web Console: p**rovides a dashboard for monitoring the functioning of the local event broker.

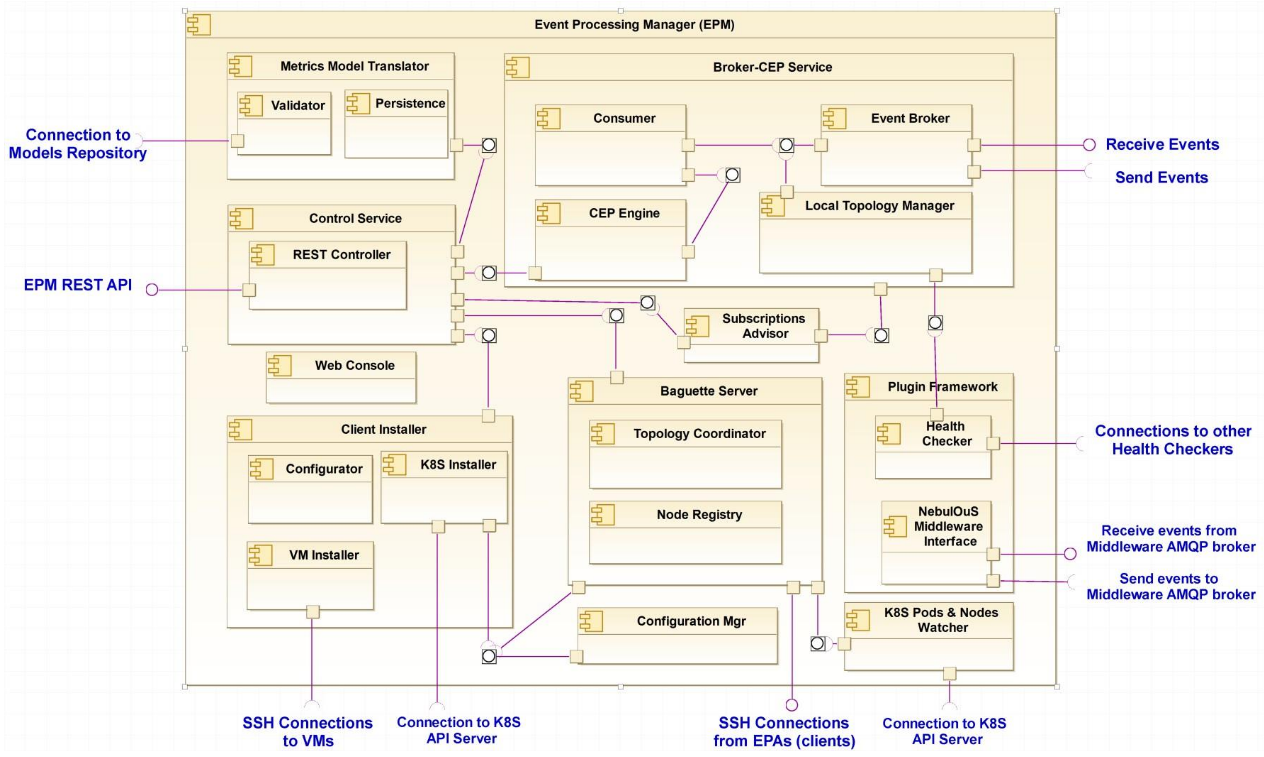


Figure 2: Revised Event Processing Manager (EPM) Architecture

As mentioned above EPM operates as a global server in a federated event processing network comprising EPAs distributed across cloud continuum resources. In NebulOuS, EPAs are deployed using their docker image that binds to EMS-client-configmap. In Figure 3, we present a high-level overview of the EPA architecture, in a UML component diagram, that comprises the following sub-components:

- **Baguette Client:** a component responsible for connecting to EPM and receiving configuration instructions. It comprises two subcomponents:
  - **SSH Client:** provides secure communication with the Baguette server using an EPM signaling protocol.
  - **Command Executor:** configures and starts the Broker-CEP service based on the instructions received from the Baguette Server of EPM.

- **Broker-CEP service:** provides the local Complex Event Processing engine (i.e., Esper[6]), along with the local Event Broker (i.e., ActiveMQ[7]) that collects and propagates metrics to EPAs in other cloud continuum nodes, where components of the same application have been deployed.
- **Local Topology Manager:** introduces the EPA to the local cluster of EPAs, while it maintains a view of the available EPAs (nodes) in the local cluster. It is also responsible for executing the Aggregator selection protocol if the cluster aggregator is lost, i.e. autonomously select the next appropriate nearby node that can undertake the role of aggregator. This process leverages the Atomix[8] framework to establish local clusters of EPAs, ensuring a robust and dynamic system capable of maintaining the EMS's operational integrity by dynamically adjusting to changes within the cloud continuum environment, such as node failures or the integration of new nodes. Furthermore, it updates the Broker-CEP component configuration when a new aggregator publishes its address and credentials. Eventually, it is notified about any changes regarding the active EPMs and updates Broker-CEP configuration accordingly.

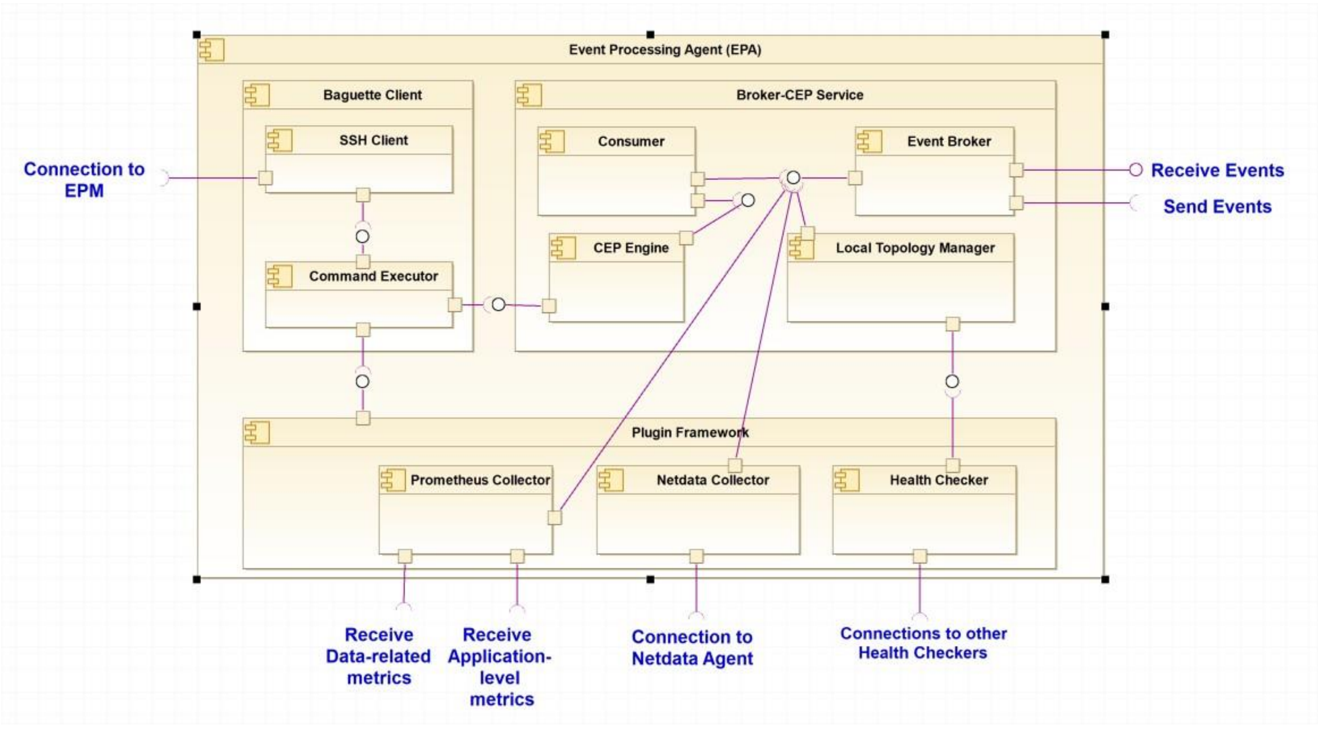


Figure 3: Revised Event Processing Agent (EPA) Architecture

- **Plugin Framework:** corresponds to a programmatic API that enables the registration and interaction of plugins with EPAs. Plugins extend the EPA by introducing new capabilities, such as the desired monitoring probes tool. In NebulOuS the Netdata Collector plugin is bundled in EPA installation package to retrieve the configured telemetry data. It subsequently

[6] https://github.com/espertechinc/esper

[7] https://activemq.apache.org/

[8] Atomix web site: https://atomix.io/

publishes the retrieved measurements to local EPA event broker as events. Additionally, a Health Checker plugin is also included, which periodically connects to event brokers of adjacent EPAs to verify they are operating properly.

In the following section we highlight the extensions to EMS system, both with respect to EPM and EPAs, based on the NebulOuS Task 5.1 work.

2.3 NEW FEATURES AND EXTENSIONS

A number of modifications and extensions have been introduced to EMS as inherited from Morphemic, in order to meet the particular requirements of cloud computing continuums. Additionally, improvements, bug fixes, and library version updates have been made. In the following section the most important of them are detailed.

2.3.1 New EMS Translator

EMS encompasses a Metrics Model Translator component that is responsible, upon request, to retrieve an application metric model (i.e. its monitoring specification), and generate all required structures and information needed for deploying and initializing EMS system and start monitoring the application. The EMS version inherited from Morphemic project has a hardwired Translator component suitable for retrieving an application's CAMEL model (which encompasses metric model) from a CDO server, in the form of a CDO object graph. It subsequently traverses the graph to extract the needed information. CDO and CAMEL domain specific language were architectural choices in Morphemic project hence their use.

In the context of Nebulous, we enhanced EMS to use pluggable Translator components, and developed a plugin that is capable to consume and process Nebulous metric models. Apart from generating the structures and information needed for initializing EMS, it furthermore generates messages sent to other components reporting the metrics they need to use in order to fulfil their respective purposes. In the remaining document, when mentioning Translator or Translator component we refer to the new Metrics Model Translator plugin, unless otherwise indicated.

A new metric model is extracted from the payload of a special event sent by Nebulous UI (once an application is submitted to the system). It is in JSON or YAML format and must abide to the Nebulous Metric Model Specification. This specification has been represented as a meta-model, using JSONSchema which is available here[9]. For this reason, the Translator plugin subscribes to a preconfigured topic (*eu.nebulouscloud.ui.dsl.metric_model*) where such events are published. Upon reception, the Translator plugin, carries out the following tasks:

- Invokes Metric Model Validator to check the model's syntax. More details are given in the next section.
- Parses JSON or YAML payload into a hierarchy of objects (i.e., dictionaries).
- Extracts information like application components and scopes (i.e., level of required metrics aggregation), SLOs, metrics, constants, and function definitions.
- Runs several (additional) checks in order to ensure the model's completeness and integrity.

[9] https://opendev.org/nebulous/monitoring/src/branch/master/nebulous/metric-model

- Builds a Directional Acyclic Graph (DAG) that describes the metrics dependencies in a syntax-agnostic way.
- Generates *complex event processing* rules (or CEP rules) that implement the monitoring functionality described in the metric model. CEP rules are generated using preconfigured templates, combined with the extracted metric model information. Currently, CEP rules are in Esper Processing Language (EPL), since Esper CEP engine is used in EMS.
- Extracted information and CEP rules are stored in a Translation Context, which can be persisted, or exported as an image, SVG or graph in DOT format.

It is worth noting that Nebulous EMS Translator is not a mere adaptation of its ancestor, i.e. the Morphemic Translator, for Nebulous needs. Contrary, it has been developed from scratch, using well-known technologies (for YAML parsing and handling). Its internal functioning differs from its ancestor's (despite their outputs are similar); the former uses a multiple-pass approach for extracting information, while the latter used a single-pass model graph traversal.

2.3.2 Metric Model Validator

This is a new EMS Translator sub-component. Its purpose is to parse a given payload in order to check if it represents a valid Nebulous metric model and report any errors it finds. EMS Translator invokes validator to check metric model validity, before it starts processing it and extracting the needed information.

Metric Model Validator uses the Nebulous Metric Model Specification in order to validate metric models. This specification is captured using a JSONSchema (meta-)model, which is itself captured as a JSON file as mentioned before. This approach allows the easy change/update of the specification, when a new version or a bug fix is available.

When a new metric model is passed to Metric Model Validator, it is first converted from YAML to JSON. Next, a dedicated validation library is used to check validity and get any errors. Any errors found are reported back to Translator in order to stop translation and notify NebulOuS platform. This approach increases the overall system reliability since the defined model is first validated and then implemented into autonomous Event Processing Network.

2.3.3 EMS refactoring – Pluggable Architecture

The EMS architecture has been reworked to allow the use of plugins in certain processing points, instead of using hardwired components like in the previous EMS versions. Plugins can be developed separately from the core EMS and included as libraries with it.

EMS offers default plugin implementations for all functions that require a plugin. Some of them are just placeholders, not performing any important operation (No-Op plugins). When EMS is configured to use a certain plugin, this will replace the default one. The most notable such plugin is EMS Translator. Other plugins are monitoring data collection plugins used in EPAs (e.g., Netdata and Prometheus collectors).

2.3.4 Context-Aware metric event propagation

When EMS is configured to deploy three-tier (or three-level) monitoring topologies, it will autonomously assign the Aggregator role to certain EPAs, typically one per group of nearby nodes (named local cluster). Specifically, EPM will inform all EPAs about all the other EPAs deployed in the same cluster and it will provide them with config details that will let them communicate and be self-organised under a selected aggregator. The aggregator is responsible for (a) receiving monitoring data from other (group) EPAs, (b) collecting the monitoring data from any nearby resource-constraint application nodes (that don't have an EPA installed), and (c) run the metric computations (as CEP rules) pertaining to the second monitoring level, using data from all EPAs and nodes of the group.

The aforementioned tasks are additional to the normal EPA tasks, hence resulting in higher resource consumption. For this reason, Aggregator EPAs periodically assess if the resource consumption gets too high (thus depriving the collocated application component from needed resources). In such a case it will query other EPAs about their status, in order to hand over the Aggregator role. This is achieved by running a distributed leader election protocol, where the state of all agents is estimate and the EPA that has enough resources available will become the new Aggregator. The EPA states are estimated with a relative score that takes into account both the maximum and the available level of resources like CPU usage, memory and storage. The exact scoring function can be set in EMS configuration. Obviously, resource-constraint nodes cannot participate in leader elections. When the Aggregator role passes from one EPA to another, all peers are automatically notified in order to reconfigure themselves and start propagating events to the new Aggregator.

2.3.5 Prometheus / OpenMetrics endpoints leveraging

A new monitoring data collection (Collector) plugin has been added to EPAs. This plugin can collect measurements from Prometheus[10] and OpenMetrics[11] endpoints that follow the latest exposition format. The collected data are mapped onto metric model raw metrics (and in fact to the corresponding event topics), using the configuration provided in the metric model. These raw metrics will be aggregated and processed, on the fly, based on several CEP rules that are constructed according to the metric model.

2.3.6 Application-metric values acquisition

EMS will provide three methods of metric values acquisition. These methods will involve the use of different monitoring probes as well as dedicated EPA data collector plugins (where applicable). It is up to application owner or DevOps to choose which of them are suitable for his/her application. Furthermore, it is possible to mix these methods, meaning that one method can be used for collecting the values of some metrics, while another method for other metrics.

The metric values acquisition methods are:

- Push of metric values to local EPA
- Expose metric values as OpenMetrics endpoint
- Push or Expose metric values to local Netdata agent

As of NebulOuS first release, the first method is available, while the next two are already in progress and included in a subsequent release.

2.3.6.1 Push to Local EPA

This method requires that applications include application-provided and controlled sensors that measure the attributes/metrics of interest, and subsequently send them to the local EPA. Since EPAs are deployed as a Kubernetes daemonset (i.e. one EPA instance exists at every Kubernetes node) there

[10] https://prometheus.io/

[11] OpenMetrics, the de-facto standard for transmitting cloud-native metrics at scale, https://openmetrics.io/

will always be one EPA running at the same node as any Application component. Therefore, the application component responsible for sending the measured values to EPA, will need to be aware of the host IP address (which is also the local EPA's IP address), as well as a few additional connection information provided as a configmap by EPM.

In detail, EPM, just before deploying EPA daemonset, will create and store a Kubernetes configmap that will be accessible by the application. This configmap will contain the port and credentials required in order to connect to EPA instances for sending metrics. The IP addresses are not included since they vary between Kubernetes nodes and because they can be easily retrieved from application components. Afterwards, EPA and Application deployments occur. On start up, EPAs and Application components will read their respective configurations (from the corresponding configmaps) and initialize accordingly. Eventually, Application will start emitting measurements to local EPAs using the configuration loaded at startup.

One important remark is that Applications using this method are required to be able to send measurements using ActiveMQ STOMP[12] or openwire[13] protocol. More options are also considered for the next release of EMS. To assist Application developers to easily fulfil this requirement, two sample implementations of measurements sending to EPA are provided; one in Java and one in Python. Moreover, a sample Deployment manifest is also provided demonstrating one way of reading configmap and getting host IP address.

Sample Kubernetes Deployment manifest, for reading configuration

A sample Kubernetes deployment manifest is given next. A more complete example including a Helm chart, can be found at NebulOuS repository at OpenDev.org, at https://opendev.org/nebulous/monitoring/src/branch/master/nebulous/examples/helm-charts/simple-app.

Table 1. Sample Kubernetes deployment manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app-name
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample-app-name
  template:
    metadata:
      labels:
        app: sample-app-name
    spec:
      containers:
        - name: sample-app-name
          image: "sample-app.image.repository:sample-app.image.tag"
```

[12] https://activemq.apache.org/components/classic/documentation/stomp

[13] https://activemq.apache.org/components/classic/documentation/openwire

```
          imagePullPolicy: IfNotPresent
          env:
            #
            #  EMS ActiveMQ connection info
            #
            - name: 'BROKER_SERVER'
              valueFrom:
                fieldRef:
                  fieldPath: status.hostIP
            - name: 'BROKER_PORT'
              value: '61610'
            - name: 'BROKER_USERNAME'
              valueFrom:
                configMapKeyRef:
                  name: monitoring-configmap
                  key: BROKER_USERNAME
            - name: 'BROKER_PASSWORD'
              valueFrom:
                configMapKeyRef:
                  name: monitoring-configmap
                  key: BROKER_PASSWORD
```

Java metric publisher to EPA

This example requires the use of ActiveMQ client library, which must be included during the code building. The relevant Maven dependency is given next.

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-client</artifactId>
    <version>6.0.1</version>
</dependency>
```

A complete example of a Java application sending random values to the local EPA, can be found at NebulOuS repository at OpenDev.org, at https://opendev.org/nebulous/monitoring/src/branch/master/nebulous/examples/simple-app-java.

Python metric publisher to EPA

This example requires the use of STOMP Python library. It can be installed using the following command:

```
pip install -r requirements.txt
```

The contents of "requirements.txt" file are:

```
docopt==0.6.2

stomp.py==8.1.0

websocket-client==1.7.0
```

A complete example of a Python application for sending random values to the local EPA, can be found at NebulOuS repository at OpenDev.org, at https://opendev.org/nebulous/monitoring/src/branch/master/nebulous/examples/simple-app-python.

2.3.6.2 Expose as OpenMetrics endpoint

This method of acquiring metric values requires the application to expose one or more Prometheus or OpenMetrics endpoints, using the OpenMeric exposition format. EPA will then periodically contact these endpoints and scrape the exposed metrics. To enable EPA contact the endpoints, it is additionally required that the application metric model provides the needed configuration. For each raw metric that will have its values using this method, it is necessary to define a sensor of "prometheus" type and provide the corresponding configuration (including the scraping period).

2.3.6.3 Push or Expose to local Netdata agent

This method of acquiring metric values requires the deployment of Netdata as a daemonset. EPA will periodically contact its collocated Netdata agent and scrape the metrics of interest. It is required that the application metric model provides the needed configuration. For each raw metric that will have its values using this method, it is necessary to define a sensor of "netdata" type and provide the corresponding configuration (including the scraping period, and netdata group, chart, and dimension).

2.4 IMPLEMENTATION DETAILS

The deployment process of all the necessary EPAs for monitoring the application and therefore constructing a dynamic event processing network is presented in Figure 4. The process is essentially triggered once the application Metric Model has been submitted and the NebulOuS Deployment Manager has deployed a new Kubernetes cluster. The deployment of the cluster also includes the automatic deployment of EPM on the master node. Once EPM receives the metric model, it validates and translates it to devise the proper configuration of EPAs (i.e., deployment Helm Charts and Configmaps). Then EPM connects to the Kubernetes API server to instruct the deployment of 1 EPA per cluster node. Each EPA once spawned inside the cluster as deamonset it uses the configmap to connect to EPM (i.e., Baguette server) and announce itself. According to EPMs instructions every EPA configures the necessary monitoring probes (and activates any other available collector e.g., Prometheus) to generate telemetry data, deploys EPL rules to be able to aggregate and process complex events and joins an EPN cluster. As described in the previous section EPAs of a certain EPN cluster automatically decide which EPA will undertake the role of local aggregator, tuning EPL rules and notifying EPM about the decision. From that point on the monitoring network is ready to monitor infrastructure, application and data related metrics, process them and detect potential SLO violations that should trigger reconfigurations.

All EMS components have been implemented using the Java™ programming language, version 21, and the Spring-boot framework, thus making their maintenance quite predictable. Third-party libraries used include, ActiveMQ classic, Esper CE library, Fabric8 Kubernetes client, Apache MINA SSH, Atomix, JGraphT, Project Lombok, various Apache Commons libraries (lang3 and text). EMS is delivered as a set of software packages; one for the EPM node, one for EPAs, and one for the Broker client tool.

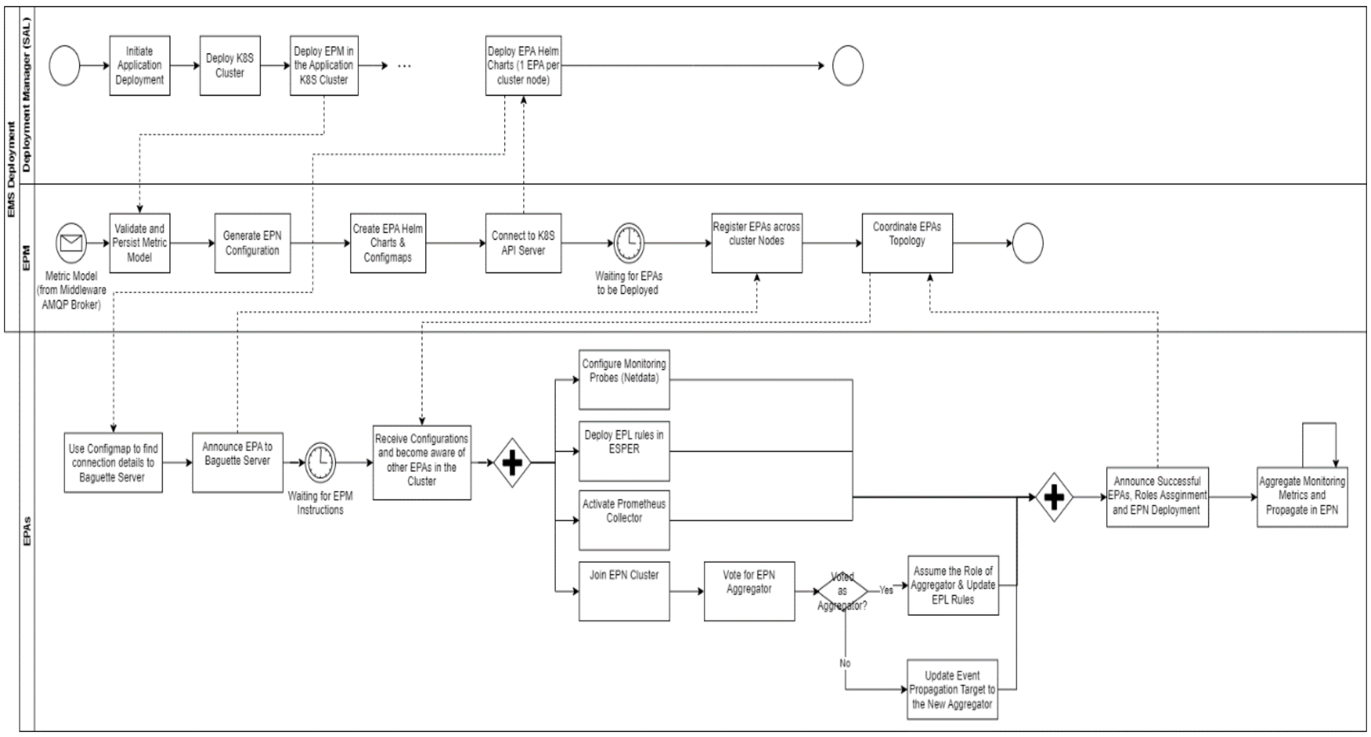

EMS Deployment
Deployment Manager (SAL)
Initiate Application Deployment
Deploy K8S Cluster
Deploy EPM in the Application K8S Cluster
Deploy EPA Helm Charts (1 EPA per cluster node)
EPM
Metric Model (from Middleware AMQP Broker)
Validate and Persist Metric Model
Generate EPN Configuration
Create EPA Helm Charts & Configmaps
Connect to K8S API Server
Waiting for EPAs to be Deployed
Register EPAs across cluster Nodes
Coordinate EPAs Topology
EPAs
Use Configmap to find connection details to Baguette Server
Announce EPA to Baguette Server
Waiting for EPM Instructions
Receive Configurations and become aware of other EPAs in the Cluster
Configure Monitoring Probes (Netdata)
Deploy EPL rules in ESPER
Activate Prometheus Collector
Join EPN Cluster
Vote for EPN Aggregator
Voted as Aggregator?
Yes
No
Assume the Role of Aggregator & Update EPL Rules
Update Event Propagation Target to the New Aggregator
Announce Successful EPAs, Roles Assignment and EPN Deployment
Aggregate Monitoring Metrics and Propagate in EPN

Figure 4: Event Processing Network Deployment Workflow

All EMS parts (EPM, EPA and Broker client) are built and bundled using the well-known Maven system. During the building process EPM and EPA Docker images are created. EMS code is available under MPL v2.0 license and it is hosted in OpenDev.org, at https://opendev.org/nebulous/monitoring/src/branch/master/nebulous

2.4.1 EMS deployment on Kubernetes cluster

Currently, EMS is deployed per application, hence it must be deployed in application Kubernetes cluster, after this has been created and configured but before application components get deployed. EMS deployment completes in a number of steps, occurring at certain points of the overall application deployment process.

First, the EPM needs to be deployed using a suitable Helm chart or an installation script that will invoke Helm. The NebulOuS Deployment Manager will execute the installation script or the Helm utility. EPM configuration can be passed as environment variables or as a host bound configuration file. After deployment, EPM will boot and connect to Nebulous message broker, and subscribe for messages regarding a new application metric model that denote monitoring details of the application to be deployed and maintained by NebulOuS.

Upon receiving a message with the application metric model, EPM will use Translator to process it and derive needed configurations. It will use its K8S Client Installer module to contact Kubernetes API server in order to:

- store a new configmap with the EPA bootstrap configuration (i.e., EPM SSH server address and credentials, collectors to enable etc),
- store a new configmap that can be used by the application components. It contains settings for connecting to the EPAs,
- deploy EPAs as a daemonset on the application cluster.

Next, Kubernetes will deploy EPA daemonset, creating exactly one EPA instance per cluster node (i.e. physical/virtual machine). When an EPA boots, it will read bootstrap settings from EPA configmap and automatically connect to EPM, in order to receive its monitoring-related configuration, as well as clustering information (i.e. nearby EPAs, and clustering key).

2.5 FUTURE WORK

In the upcoming releases of NebulOuS platform, EMS will be further extended with a number of planned features. These include Context-Aware metric security and reworking of EPM to support multiple applications.

2.5.1 Context-Aware metrics security

The first planned EMS extension, is the addition of a new feature in EMS that regards a flexible security mechanism, applied on monitoring data exchange between EMS parts (EPM and EPAs), as well as the collection of monitoring data from monitoring probes.

When exchanging monitoring data, the default EMS configuration requires that all interactions are encrypted. In fact, all EMS parts automatically generate credentials and cryptographic keys, and securely announce them to EPM and other EPAs. This behaviour can be totally disabled (although not advised), in which case all monitoring data exchanges will become unencrypted. However, there are valid cases where data encryption is not critical or comes to a very high cost that does not justify the encryption. For instance, in cases where small IoT devices are part of the application, data encryption could consume significant amount of computing resources, depriving them from the application components. Furthermore, it can also drain the available power source (battery) faster, and probably reduce the expected lifetime of the battery or even of the device. In such situations, it would be very desirable to be able distinguishing between critical monitoring data that must be encrypted before sent over the network, from non-critical data that can go unencrypted. The decision on the data criticality could be either directly stated in the metric model or inferred based on where the organisational boundaries, the use or not of third party devices, the use of trusted hosting providers or not, etc. Additionally, it is desirable this distinction is flexible and adaptable, based on the current operational status of the device; for example, critical measurements are sent encrypted to EMS, unless the battery level is extremely low in which case, they can be sent unencrypted. If the power level increases to a safe level (e.g. device gets plugged to power network) then critical measurements are again sent encrypted.

Another case where monitoring data encryption is not required would be an application deployed in private network or in a network where communication links are encrypted (e.g. using VPN). In this case the monitoring data encryption offers no additional security. If, however, an application node is connected both to secure and insecure networks, then it is desirable to use encryption when transmitting over the insecure links at least.

2.5.2 Support for Multiple Applications

The second major EMS extension we plan, regards reworking EMS in order to be able handle more than one application simultaneously. EMS was originally designed and implemented to monitor a single application (that was a project requirement). However, in the context of a meta-OS like NebulOuS is, this requirement is no longer valid. In fact, the opposite is required, since the same meta-OS services must be able to cope with several different applications and application instances (as it happens in normal OS'es). Handling multiple applications on the side of the EMS means that resources which are inside the EMS, common to all applications need not be committed for each new application which is introduced.

In order to meet this requirement inversion, we currently deploy one EPM instance and a separate EPA network for each application deployment. For this reason, EPM is deployed in the application cluster along with application components and other platform agents.

In the next EMS release, we plan to revise EMS workflows, and data structures, and rework EPM and EPA implementations in order to be able to handle multiple applications. We do not expect significant architectural changes to be required, however the implementation changes are expected to be quite extensive. Next, we detail the most significant of them.

- Introduction of an Application Monitoring Context, where all monitoring-related information pertaining to one application is stored. This information can be either extracted from the metric model, or provided by EPAs, or acquired from any other source. EPM will create and maintain Application Monitoring Context instances to track the state of the monitoring network of each application. Snapshots of Application Monitoring Context instances will be stored, in order to enable load balancing and EPM node replacement. We note that Translator already generates Translation Context instances with the outcomes of metric model analysis. Additionally, EPA data are store in EPM Node Registry, per IP address. Node Registry entries will need to be distinguished per application too.
- Identifiers used for naming event topics, CEP streams or any other structure, will be suitably modified in order to include an application reference or allow deriving it. Moreover, Application identification keys will be introduced in EMS interactions to help distinguishing between applications. These identifiers will be stored in Application Monitoring Context instances.

Apart from the aforementioned extensions, additional changes might be introduced to EMS during the project progress, either in response to new or evolving use case requirements, or due to other reasons.

3 AI-DRIVEN ANOMALY DETECTION AT THE EDGE

As described in deliverable D2.1 Requirements and Conceptual Architecture of the NebulOuS Meta-OS, the AI-driven Anomaly Detection engine ensures Quality of Service (QoS), by detecting scenarios that perform differently from what they are expected to in the resource utilization of k8s-based applications (k8s). Leveraging Netdata for malicious/adversarial/intrusion behavior provides a real-time and comprehensive approach to monitor and respond to deviations from normal patterns.

Our approach aims to utilize Netdata to collect data and to detect anomalies in a set of resource metrics of Kubernetes clusters taking into account several metrics. By contrast, Netdata offers an anomaly detection mechanism that only analyzes each metric in an isolated manner. On the other hand, integrating Netdata with an immunological algorithm like Dendritic Cell or Negative Selection for malicious/adversarial/intrusion behavior involves combining the real-time monitoring capabilities of Netdata with the Figure 5 shows a scheme where in each worker node there is a Netdata child and in each cluster container there is a Netdata parent and together with an ML algorithm. Within each instance per application there is a database containing data collected by Netdata.

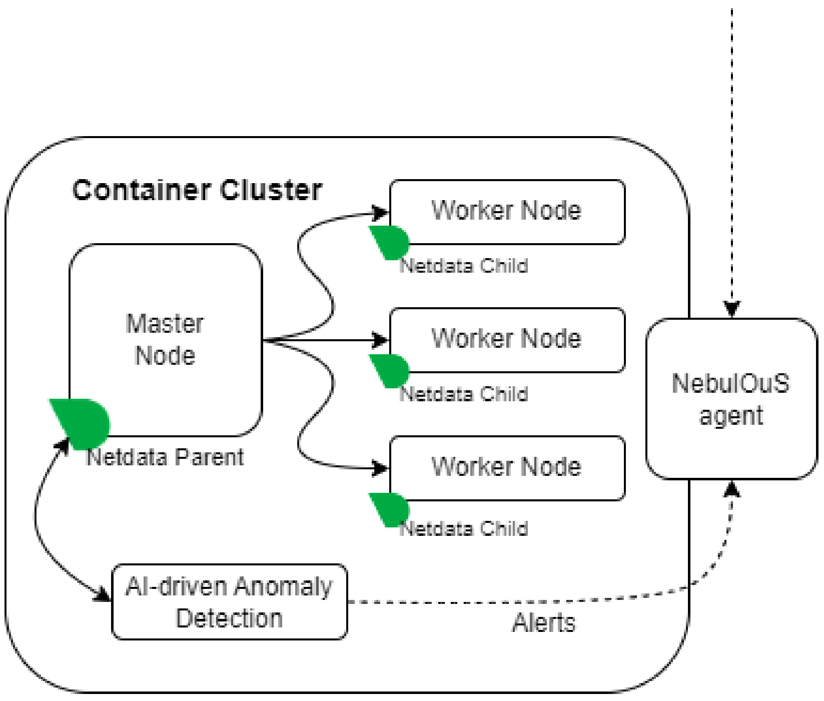


Figure 5: Kubernetes-NetData-ML approach.

3.1 APPROACH/METHODOLOGY/MODELS

Netdata deployment involves deploying Netdata agents on Kubernetes worker nodes using Kubernetes DaemonSets and ConfigMaps for scalable deployment.

A baseline establishment is achieved by establishing normal behavior for resource utilization metrics using historical data, while statistical models like moving averages or standard deviations are utilized to identify deviations from this baseline. The methodology for that involves analyzing historical data of resource utilization metrics, such as CPU, memory, and disk usage, to define normal behavior within each cluster. The amount of historical data required depends on factors like the size of the cluster, the variability of workloads, and the desired level of accuracy. Typically, a sufficient amount of historical data covering at least a few weeks to a month is recommended to capture diverse patterns and variations in workload behavior. Our idea is to check if two weeks are enough for a selected use case and then work accordingly with the rest of the cases. The frequency of updating the baseline should be determined based on the dynamic nature of the cluster's workloads. In environments where workloads change frequently or seasonally, the baseline may need to be updated more frequently to accurately reflect the current typical behavior. Conversely, in more stable environments, less frequent updates may suffice. A common approach is to update the baseline periodically, such as daily or weekly, and adjust the frequency as needed based on observed changes in workload patterns. Automated processes can streamline this updating process, ensuring that the baseline remains relevant and reflective of the cluster's evolving behavior. Our initial idea is to update daily.

When considering machine learning algorithms for anomaly detection, several types of algorithms can be explored, each with its own strengths and suitability depending on the specific characteristics of the data and the desired outcomes. Some of the commonly considered machine learning algorithms include clustering algorithms, supervised learning algorithms, unsupervised learning algorithms, and deep learning algorithms. On the other hand, immunological algorithms, inspired by the principles of the immune system, offer another approach. The immune system is a complex network of cells, tissues, and organs that work together to defend the body against pathogens, such as viruses and bacteria. It achieves this through a process of recognizing foreign entities (antigens) and mounting an immune response against them while tolerating the body's own cells (self).

Immunological algorithms draw inspiration from several key principles of the immune system, such as recognition, learning and adaptation, and diversity. Just as the immune system can distinguish between self and non-self antigens, computational models classify data points as either normal (self) or anomalous (non-self) based on certain features or characteristics. Similarly, the immune system's ability to adapt to new threats by learning from past encounters is mirrored in immunological algorithms, which often incorporate mechanisms for learning and adaptation to dynamically adjust to changes in the data environment. Additionally, like the diverse repertoire of immune cells maintained by the immune system to recognize a wide range of antigens, immunological algorithms employ diverse sets of detectors or classifiers to handle different types of anomalies.

Two algorithms, such as Dendritic Cell Algorithm (DCA) or Negative Selection Algorithms (NSA), mimic the mechanisms of the immune system to recognize and respond to anomalies.

We have evaluated eight algorithm implementations: a k-nearest neighbors algorithm[14], a logistic regression algorithm[15], a random forest algorithm[16], a decision tree classifier algorithm[17], an XGBoost algorithm (which utilizes gradient-boosting decision trees)[18], a LightGBM algorithm (leveraging a histogram-based method that bins data using a distribution histogram)[19], a dendritic cell algorithm [8], [9] and a negative selection algorithm [10]. Our benchmark used the NSL-KDD dataset[20][11]. The XGBoost and LightGBM algorithms demonstrated superior efficiency for our specific task, but the k-nearest clustering algorithm could be a good election if simplicity and interpretability are sought.

The criteria for selecting machine learning algorithms depend on various factors, including the nature of data (its dimensionality and distribution), the scalability of the algorithm (capability of process large volumes of data), the robustness of the algorithm (adaptability to changes in the data distribution). The inclusion of immunological algorithms in the selection process offers unique advantages such as self-regulation, self-organization, and self-repair, making them robust and adaptable to changes. Additionally, these algorithms can handle high-dimensional and noisy data effectively. In the next iteration we will work / test with real data, the LightGBM algorithm or a

[14] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[15] https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model

[16] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier

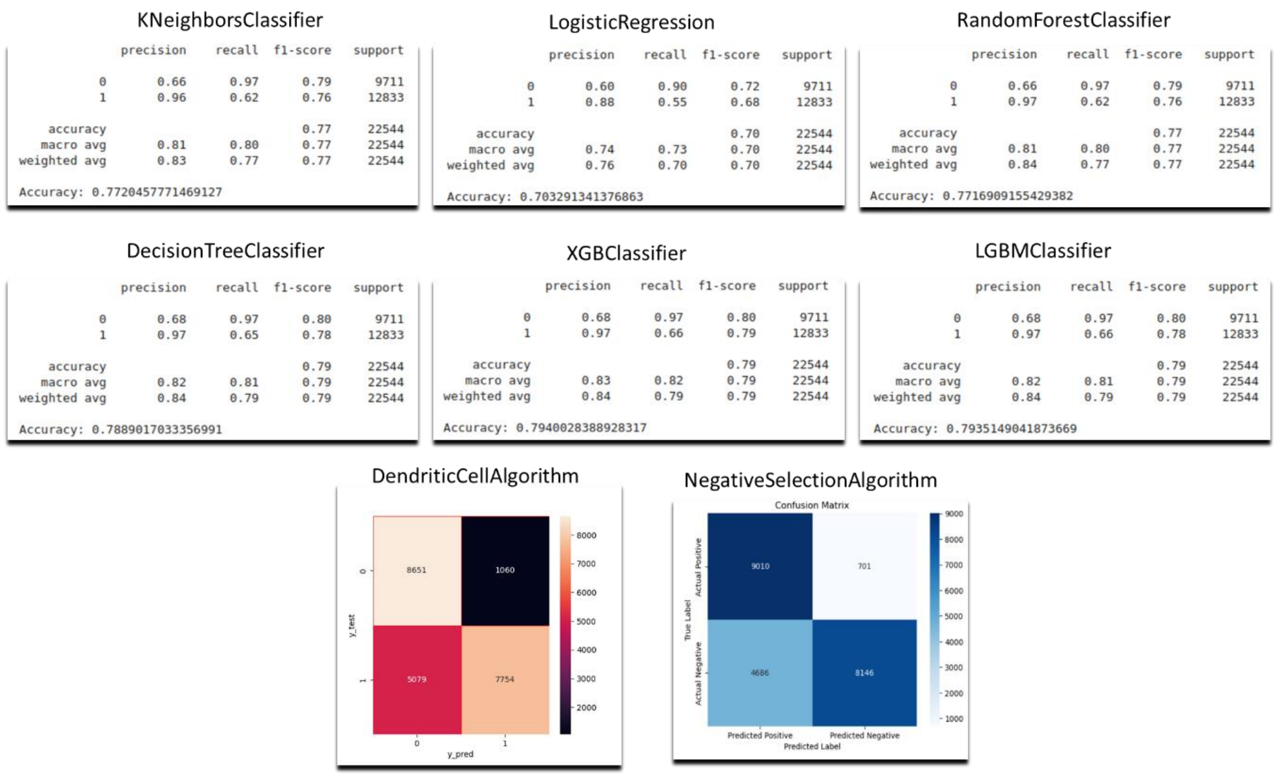
[17] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn-tree-decisiontreeclassifier

[18] https://xgboost.readthedocs.io/en/latest/python/python_api.html

[19] https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html

[20] https://www.unb.ca/cic/datasets/nsl.html

clustering algorithm and some of the immunological ones. We are deeply interested in gaining knowledge and analyzing the potential of these latest algorithms.



KNeighborsClassifier

```
              precision    recall  f1-score   support

           0       0.66      0.97      0.79      9711
           1       0.96      0.62      0.76     12833

    accuracy                           0.77     22544
   macro avg       0.81      0.80      0.77     22544
weighted avg       0.83      0.77      0.77     22544

Accuracy: 0.7720457771469127
```

LogisticRegression

```
              precision    recall  f1-score   support

           0       0.60      0.90      0.72      9711
           1       0.88      0.55      0.68     12833

    accuracy                           0.70     22544
   macro avg       0.74      0.73      0.70     22544
weighted avg       0.76      0.70      0.70     22544

Accuracy: 0.703291341376863
```

RandomForestClassifier

```
              precision    recall  f1-score   support

           0       0.66      0.97      0.79      9711
           1       0.97      0.62      0.76     12833

    accuracy                           0.77     22544
   macro avg       0.81      0.80      0.77     22544
weighted avg       0.84      0.77      0.77     22544

Accuracy: 0.7716909155429382
```

DecisionTreeClassifier

```
              precision    recall  f1-score   support

           0       0.68      0.97      0.80      9711
           1       0.97      0.65      0.78     12833

    accuracy                           0.79     22544
   macro avg       0.82      0.81      0.79     22544
weighted avg       0.84      0.79      0.79     22544

Accuracy: 0.7889017033356991
```

XGBClassifier

```
              precision    recall  f1-score   support

           0       0.68      0.97      0.80      9711
           1       0.97      0.66      0.79     12833

    accuracy                           0.79     22544
   macro avg       0.83      0.82      0.79     22544
weighted avg       0.84      0.79      0.79     22544

Accuracy: 0.7940028388928317
```

LGBMClassifier

```
              precision    recall  f1-score   support

           0       0.68      0.97      0.80      9711
           1       0.97      0.66      0.78     12833

    accuracy                           0.79     22544
   macro avg       0.82      0.81      0.79     22544
weighted avg       0.84      0.79      0.79     22544

Accuracy: 0.7935149041873669
```

Figure 6: Comparison of the different algorithms evaluated.

The alert configuration process involves defining thresholds and conditions that trigger alerts based on anomalies detected in the monitored metrics. This process starts by identifying the essential metrics critical for monitoring, in our case, the Kubernetes cluster's health and performance (CPU usage, memory utilization, and network traffic). Baseline thresholds should then be established for each chosen metric, either through historical data analysis or predefined standards. Machine learning models are then incorporated to analyze metric data and identify deviations from established baselines. Sensitivity levels must be adjusted to balance minimizing false positives and false negatives (hyperparameter optimization). Finally, it is necessary to define specific conditions or rules that activate alerts when machine learning models detect anomalies and send notifications if necessary to the event management.

3.2 CONCEPTUAL ARCHITECTURE & IMPLEMENTATION DETAILS

In deploying Netdata within Kubevela as a DaemonSet, each worker node is ensured to have an instance, supported by the creation of ConfigMaps to store configuration settings. Automating the deployment across nodes via DaemonSets streamlines the process, complemented by Kubernetes services for efficient management and exposure of Netdata instances. This setup facilitates comprehensive monitoring coverage and seamless access to monitoring data.

Table 2 : Kubevela Netdata cluster deployment example

Kubevela Netdata cluster deployment example

```
################### SERVICE ##################
apiVersion: core.oam.dev/v1beta1
kind: Component
metadata:
  name: netdata-parent-service
spec:
  type: webservice
  properties:
    image: netdata/netdata
    port: 19999
    expose:
      - port: 19999
        as: NodePort
        nodePort: 30000
################### CONFIGMAP - PARENT - CONFIG ##################
apiVersion: core.oam.dev/v1beta1
kind: Component
metadata:
  name: netdata-parent-configmap
spec:
  type: raw
  properties:
    apiVersion: v1
    kind: ConfigMap
    metadata:
      name: netdata-parent-config
```

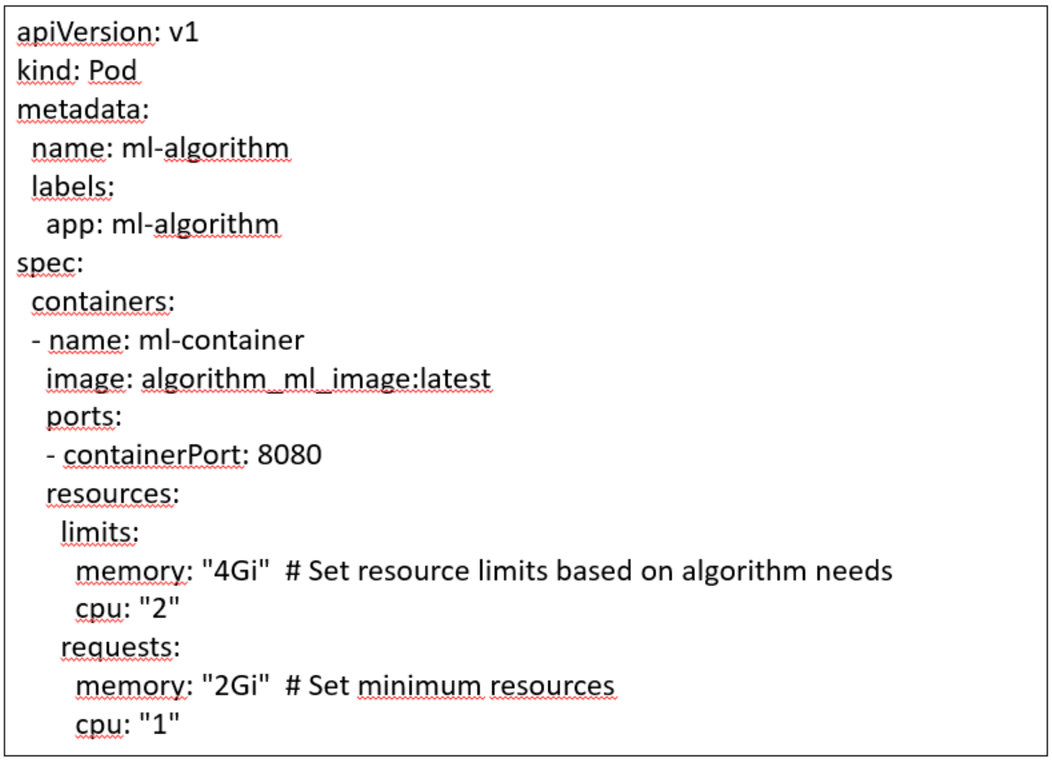
```
#################### NETDATA - PARENT ###################
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: netdata-parent-app
spec:
  components:
    - name: netdata-parent-service
      type: webservice
      properties:
        # Propiedades del servicio
    - name: netdata-parent-configmap
      type: raw
#################### CONFIGMAP - CHILD - CONFIG ###################
apiVersion: core.oam.dev/v1beta1
kind: Component
metadata:
  name: netdata-child-configmap
spec:
  type: raw
  properties:
    apiVersion: v1
    kind: ConfigMap
metadata:
      name: netdata-child-config
    data:
      netdata-streaming.conf: |
        [stream]
            enabled = yes
            destination = netdata-parent-service:19999
            api key = c4ea96fa-1329-4f77-b503-d607db19be52
```

```
################### NETDATA - DAEMONSET - CHILD ##################
apiVersion: core.oam.dev/v1beta1
kind: Component
metadata:
  name: netdata-child-daemonset
spec:
  type: worker
properties:
    image: netdata/netdata
    cmd:
      - ["/bin/bash"]
    ports:
      - containerPort: 19999
################### NETDATA - CHILD ##################
apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: netdata-child-app
spec:
  components:
    - name: netdata-child-daemonset
      type: worker
    - name: netdata-child-configmap
      type: raw
```

Subsequently, it is necessary to integrate a malicious/adversarial/intrusion detection system (a machine learning component) incorporating algorithms such as the LightGBM and/or immunological (seeTable 3 Table 3). Additionally, establish communication channels between the machine learning process and Netdata in an initial phase, followed by another channel between the prediction process (which uses the generated machine learning model) and Netdata, is crucial for ensuring seamless real-time metric transfer on both occasions. These connections will use the capabilities of the "get_data" function implemented in netdata_pandas.data (API calls). This feature acts as a conduit, facilitating the monitoring flow.

Funded by
the European Union

Table 3: Machine learning algorithm deployment example



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ml-algorithm
  labels:
    app: ml-algorithm
spec:
  containers:
  - name: ml-container
    image: algorithm_ml_image:latest
    ports:
    - containerPort: 8080
    resources:
      limits:
        memory: "4Gi"  # Set resource limits based on algorithm needs
        cpu: "2"
      requests:
        memory: "2Gi"  # Set minimum resources
        cpu: "1"
```

3.3 INTERFACES OFFERED/REQUIRED

1. Netdata interface:

The interface with Netdata APIs (or data sources) allows real-time metrics to be retrieved for analysis using a machine learning algorithm.

Key system metrics captured by Netdata include CPU, memory, disk usage, network bandwidth, process counts, and more. Container metrics include resource utilization metrics for containers and virtual machines using /sys/fs/cgroup. Advanced metrics include using eBPF, kernel-level metrics like file descriptors, virtual filesystem I/O, and process management are captured.

2. Machine learning detection system interface:

The interface with the machine learning detection system allows to configure, monitor and retrieve information from learning models integrated with Netdata.

3.4 BEYOND STATE-OF-THE-ART

Exploring the potential of combining the immunological algorithm with other machine learning techniques presents an exciting opportunity for a hybrid approach that harnesses the strengths of both bio-inspired and traditional methods. By integrating the immunological algorithm with established machine learning techniques such as deep learning, reinforcement learning, or ensemble methods, we can create a robust framework capable of tackling complex security challenges in diverse environments. This hybrid approach allows us to leverage the adaptability and self-learning capabilities inspired by biological immune systems, while also harnessing the predictive power and scalability offered by traditional machine learning algorithms. Moreover, combining these techniques opens avenues for enhanced feature extraction, pattern recognition, and anomaly detection, ultimately leading to more accurate and effective intrusion detection systems. Through rigorous experimentation and optimization, we can unlock the full potential of this hybrid approach, paving the way for advanced threat detection and mitigation strategies in cybersecurity.

3.5 NEXT STEPS FOR THE FINAL ITERATION OF THE MECHANISMS

Through the development of Kubernetes Helm charts, streamlined deployment and management of Netdata within Kubernetes are facilitated, enabling easier installation, updates, and configuration management across the Kubernetes environment. Performance optimization efforts are directed towards ensuring the integrated system efficiently handles large-scale and dynamic Kubernetes environments.

Additionally, establishing a feedback loop between Netdata and the machine learning algorithm not only enables continuous adaptation to new patterns but also offers several potential benefits:

- Enhances the ability to detect emerging threats or anomalies in real-time by dynamically adjusting detection criteria based on changing data patterns.
- Improves the accuracy and effectiveness of intrusion detection by feedback.
- Enables proactive threat mitigation by identifying and responding to suspicious activities before they escalate.

By incorporating these potential benefits, the feedback loop between Netdata and the machine learning algorithm becomes fundamental in malicious/adversarial/intrusion detection accuracy. Here's a succinct outline of the steps involved in implementing the feedback mechanism:

- Define a function to capture the model's predictions and their corresponding outcomes (true positives, false positives, etc.).
- Analyze the captured data to identify misclassifications or false alarms and their underlying causes.
- Integrate the feedback into the ML model to adjust its parameters or update its training data.
- Periodically evaluate the performance of the ML model using metrics such as precision, recall, F1-score, etc.
- Monitor the model's capability to recognize new patterns and identify emerging threats over time.
- Incorporate mechanisms to track changes and improvements made to the model based on feedback received.

Furthermore, integration with alerting mechanisms or automated responses ensures timely reactions to identified anomalies, enhancing overall security. By leveraging Netdata's real-time monitoring capabilities and the adaptive nature of the chosen machine learning algorithms, this approach aims to strengthen intrusion, adversarial or malicious detection in Kubernetes clusters by recognizing deviations from normal behavior in a natural and simple way.

4 INTEROPERABLE IOT/FOG DATA MANAGEMENT

As described in deliverable D2.1 Requirements and Conceptual Architecture of the NebulOuS Meta-OS, The Data Collection and Management component is related to data management, both in the context of managing the monitoring data collected from the IoT/Edge/Cloud compute resources, as well as with respect to the data exchanged internally between the NebulOuS components. Regarding the IoT data management, three distinct sub-components were envisioned to comprise this module: A pub/sub mechanism, a time-series IoT data store, an IoT data flow pipelines orchestration mechanism.

4.1 IOT/FOG PUB/SUB MECHANISM

Distributed micro-service oriented applications that can benefit from the NebulOuS resource brokerage capabilities require some communication mechanism to exchange messages between its components. Often, this kind of architectures rely on the publish/subscribe paradigm of communication. To facilitate the adoption of NebulOuS by such applications, the Data Collection and Management module offers a pub/sub mechanism. Its goal is to allow the communication between modules of the applications running on top of NebulOuS and serve as an entry point for IoT data required by these applications. This pub/sub mechanism articulated using Apache ActiveMQ allows components of the application to exchange messages using well known protocols such as AMQP[21], MQTT[22] , STOMP[23] and REST[24]. On top of that, we offer extensions for the ActiveMQ message broker to automatically collected metrics about the communication patterns between the application components (number of messages per second, total size of the messages, message processing latency, etc..). These metrics can then be used by application owners as part of the application definition to detect SLO violations and decide on the need for re-adaptation.

4.1.1 Conceptual Architecture & Implementation details

The proposed pub/sub mechanism utilizes Apache ActiveMQ Artemis[25] as a message broker. Artemis is an open-source project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system[26].
Whilst utilizing a custom protocol for its inter-broker communication in clustered deployments, Artemis is compatible with a wider range of standard and well-known protocols in the IoT landscape. It supports: AMQP 1.0, MQTT (3.1, 3.1.1 and 5), STOMP 1.0, 1.1, or 1.2 and REST, among others. Artemis unifies the different messaging semantics proposed by these standards and allows clients utilizing any of these standards to communicate using a single broker, thus, facilitating the integration of data streams coming different data sources (IoT devices, industrial equipment, etc.).

[21] https://www.amqp.org/resources/specifications

[22] https://mqtt.org/

[23] https://stomp.github.io/

[24] https://activemq.apache.org/components/artemis/documentation/2.25.0/rest.html

[25] https://activemq.apache.org/components/artemis/

[26] https://activemq.apache.org/components/artemis/documentation/

Although Artemis is not bundled as part of the standard deployment of NebulOuS core nor its agents, we propose its usage to articulate the communication between application components. For this, we have developed a reference application that shows how a NebulOuS user (DevOps) can model its application to include the messaging broker.
Apache ActiveMQ Artemis offers the flexibility to be deployed using several strategies[27]. From a centralized broker instance where all clients connect to a central point to a clustered deployment where all brokers are connected conforming a single virtual broker and messages sent to one broker address will be transparently distributed to brokers where consumers for the destination topics are connected. This later approach allows having user application deployments where multiple applications components reside in the same Kubernetes node and, thus, utilize one single instance of the Artemis broker (limiting the number of resources needed) whilst still being able to have other components deployed on separated Kubernetes nodes without affecting the ability to communicate with the rest of the application components. Additionally, having multiple components that exchange messages between them residing in the same Kubernetes node permits to eliminate the transmission of the messages to the network, lowering the bandwidth needed for the application and improving the latency in communications. To leverage the cluster capabilities of ActiveMQ, the proposed deployment relies on the use of Kubernetes DeamonSets[28]. This mechanism allows to define a Pod that will be automatically deployed in each Kubernetes node running any other regular application, Pods. Conversely, when all regular application Pods are removed from a node, the DeamonSets Pod instance is automatically deallocated. Moreover, defining the Artemis broker as a DeamonSet on the user application models (OAM) managed by NebulOuS, simplifies the deployment of the pub/sub nodes (as the user doesn't need to specify the details of each node) and guarantees that one instance of the pub/sub broker will be available locally on each node running any of the user application components Pods.
Figure 7 shows the deployment topology that is making use of the clustering capabilities of Apache Artemis. This application, composed of several components (C1, C2, … C7) is deployed across 3 different pods (POD 1, POD 2, POD 3). On each POD, a message broker is automatically installed by Kubernetes as a daemon set. The brokers conform a cluster that permit not only the communication between components in the same POD (C1 and C2, C3 and C4) but also the communication with components on other PODs.
From a security point of view, the proposed broker deployment works under the assumption that there will exist one Artemis cluster for each end user application that NebulOuS might manage. This guarantees complete isolation of the broker with other applications owned by the same user or other users. However, utilizing the bridging and clustering capabilities of Artemis[29], application owner can freely connect two or more NebulOuS managed applications Artemis cluster to facilitate inter-application communication.
Continuing with the security aspects of the proposed solution, it is left to the owner of the application to decide on the authentication and authorization policies to be enforced by the broker. In this respect, Artemis offers a comprehensive set of configurations that enables it to adapt to the specificities of different usage scenarios[30].

[27] https://activemq.apache.org/components/artemis/documentation/

[28] https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[29] https://activemq.apache.org/components/artemis/documentation/latest/clusters

[30] https://activemq.apache.org/components/artemis/documentation/latest/security.html#authentication-authorization

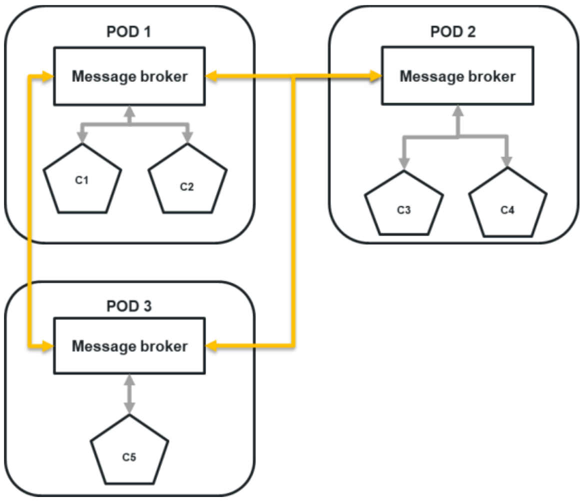


Figure 7: Sample application deployment topology

Table 4 shows a sample application that utilizes the proposed pub/sub mechanism. The example is composed of 5 objects, the master broker, the daemon broker, the producer, and the consumer.

- ActiveMQ Master broker: Necessary for articulating the ActiveMQ cluster.
- ActiveMQ Daemon broker: Defined as a daemon to be deployed along each of the nodes PODs. It is configured to contact the ActiveMQ Master.
- ActiveMQ Demon service: Connects PODs for clients of the ActiveMQ broker to the broker for their POD deployed using daemonsets.
- Producer and Consumer: Dummy ActiveMQ clients that can connect to the host "artemis-daemon-service" to contact the ActiveMQ broker associated with the POD where it is running.

Table 4 : OAM of a sample application using the proposed pub/sub mechanism

```
kind: Application
metadata:
  name: pub-sub-mechanism-deployment
spec:
  components:


    - name: pub-sub-master
      type: webservice
      properties:
        image: activemq/artemis
        volumeMounts:
          configMap:
            - name: broker-config.xml
              mountPath: /broker-path/etc-override/broker.xml
              cmName: broker-config-cmap
      traits:
        - type: expose
          properties:
            port: [8161,61616]

    - name: pub-sub -daemon
      type: daemon
      properties:
        image:  activemq/artemis
        ports:
          - port: 61616
            protocol: TCP
          - port: 8161
            protocol: TCP
    - name: daemon-service
      type: k8s-objects
      properties:
        objects:
          - apiVersion: v1
            kind: Service
            metadata:
              name: artemis-daemon-service
            spec:
              internalTrafficPolicy: Local
              ports:
                - name: msg
                  port: 61616
                  protocol: TCP
                  targetPort: 61616
                - name: http
                  port: 8161
                  protocol: TCP
                  targetPort: 8161
              selector:
```

Funded by the European Union

```
          app.oam.dev/component: pub-sub -daemon
         type: NodePort
 - name: producer
   type: webservice
   properties:
     image: nebulous/iot-producer
   traits:
     - type: "node-affinity"
       properties:
         affinity:
           name: ["node-1"]
 - name: consumer
   type: webservice
   properties:
     image: nebulous/iot-consumer
   traits:
     - type: "node-affinity"
       properties:
         affinity:
           name: ["node-2"]
```

4.1.2 Interfaces offered/required

Utilizing the extensibility features offered by Artemis, we have developed a plugin that automatically publishes to the NebulOuS EMS agent (i.e. EPA) metrics related to the usage of the pub/sub mechanism by the user applications and therefore revealing aspects of the application data exchanged. These metrics can then be used by application owners as part of the application metric model used by NebulOuS to decide on the need for re-adaptation. More precisely, the developed plugin collects the following metrics:

- messages_count: The number of pending messages for any queue.
- max_message_age: Age of the oldest pending message on a given queue.
- consumers_count: The number of active consumers subscribed to a queue.
- group_count: The number of message groupings for a queue. Messages on a queue are grouped by the value of the "JMSXGroupID" attribute associated to each message. This metric is relevant for the IoT Data processing pipeline orchestration tool (described section 4.3).

The metrics are reported per queue and broker on a fixed interval (e.g., each 30 seconds).

Additionally, events related to the lifecycle of messages exchanged between user application components using the proposed pub/sub mechanism are reported. On each step of the message lifecycle (a producer writes a message to the cluster, the message is delivered to a consumer, the consumer acknowledges the message), the plugin generates events with relevant information that can be used for understanding how IoT applications components on NebulOuS are communicating. On each of these lifecycle events, several raw parameters are reported (e.g., pub-sub node receiving/serving the message, id of the message, timestamp of the event, etc…). The details on the parameters reported on each of these events are detailed in Table 5, Table 6 and Table 7.

Table 5: Message published event

Event name	Message published event	
Event description	Event generated when a message is published to the pub/sub system	
Fields		
Name	**Type**	**Description**
messageId	Long	Id of the message
timestamp	Long	Timestamp when the event occurred.
messageSize	Long	Size of the message (in bytes).
messageAddress	String	Address where the message is published.
node	String	The name of the pub/sub cluster node where the message was published.

Table 6 Message delivered event

Event name	Message delivered event	
Event description	Event generated when a message is delivered to a client of the pub/sub system	
Fields		
Name	**Type**	**Description**
messageId	Long	Id of the message.
timestamp	Long	Timestamp when the event occurred.
messageSize	Long	Size of the message (in bytes).
messageAddress	String	Address where the message is consumed.
node	String	Node where the client is connected.
clientId	String	Id of the client receiving the message.
publishNode	String	Node where the message was originally published.

publishAddress	String	Address where the message was originally published.
publishClientId	String	Id of the client that originally published the message.
publishTimestamp	String	Time when the message was published (milliseconds since epoch).

Table 7 Message acknowledged event

Event name	Message acknowledged event	
Event description	Event generated when a message is acknowledged by a client of the pub/sub system	
Fields		
Name	**Type**	**Description**
messageId	Long	Id of the message.
timestamp	Long	Timestamp when the event occurred.
messageSize	Long	Size of the message (in bytes).
messageAddress	String	Address where the message is published.
node	String	Node where the client is connected.
clientId	String	Id of the client receiving the message.
publishNode	String	Node where the message was originally published.
publishAddress	String	Address where the message was originally published.
publishClientId	String	Id of the client that originally published the message.
publishTimestamp	String	Time when the message was published (milliseconds since epoch).
deliverTimestamp	String	Time when the message was delivered to the client.

With these raw parameters, relevant metrics can be derived by the EMS system that can help to decide on the need to scale/downscale the application components responsible for processing messages to adapt to a variation of the workload. In this respect, there are several metrics that can be derived. Some examples are:

- Time difference between a message being originally sent to the broker by a producer and the message being delivered to a consumer.
- Time difference between a message being delivered to a consumer and the consumer acknowledging the message.
- Number of messages per second shared between two components of the user application.
- Volume in megabytes of data shared between two components of the user application.

4.1.3 Next steps for the final iteration of the mechanisms

The current proposed deployment for the pub/sub mechanism relies on the use of static XML configuration files to specify the authentication/authorization aspects. This becomes a burden when working with IoT related application, where the number of clients can grow very fast if each IoT device has its own credentials. The second iteration of the pub/sub mechanism will investigate the integration of external authentication/authorization frameworks like Keyrock[31] and Keycloack[32], utilizing the extensibility offered by Artemis with its Security Settings Plugin[33]

4.2 IOT TIME SERIES DATA STORE

Although the deployment of a IoT time series database was envisaged as a component to be deployed as part of NebulOuS platform, later analysis of the matter has led to conclude that a better approach is to let the application owner decide the most appropriate database technology for their use case and deploy it as a regular NebulOuS application (or part of it). This decision is motivated by the fact that, each application will have specific requirements regarding the solution to be used. These requirements can be related to topics such as how the data is to be stored (wide table vs narrow tables), the high availability requirements and target performance (inserts per second, queries per second, etc…) among others.

Nevertheless, as part of "T5.3 Interoperable IoT/Fog data collection and management" efforts will be put on documenting the approach to integrate TimeScaleDB[34] into an application managed by NebulOuS and connect it to the proposed pub/sub mechanism.

TimescaleDB is an open-source time-series database engine. Built on top of PostgreSQL, it adds several improvements to optimize the storage of time series data on a massive scale (up to tens of TB uncompressed). TimescaleDB is specifically optimized for storing and querying time-series data. It provides a robust set of query functionalities for handling time intervals, downsampling, and other common time-series operations.

[31] https://keyrock-fiware.github.io/

[32] https://www.keycloak.org/

[33] https://activemq.apache.org/components/artemis/documentation/

Among other competitors (Apache Druid[35], InfluxDB[36], OpenTSDB[37], Graphite[38], etc.) TimescaleDB has been chosen as the tool to demonstrate the integration of time series databases in NebulOuS managed applications because it offers similar time-series query capabilities while it requires little administration efforts in comparison with other technologies.

4.3 IOT DATA PROCESSING PIPELINES ORCHESTRATION TOOL

4.3.1 Introduction

NebulOuS provides specific semantics for modelling IoT data processing pipelines by identifying their main components (data sources, transformation operations and data consumers) and interlace them to conform data transformation flows. Once defined by the user, the orchestration of these data transformation pipelines is handled by NebulOuS core, allocating/deallocating computational resources whenever needed to adapt to the current workload.

To better understand better the NebulOuS approach, consider the vehicle fleet monitoring solution depicted in Figure 8: Vehicle fleet monitoring IoT data pipeline example.

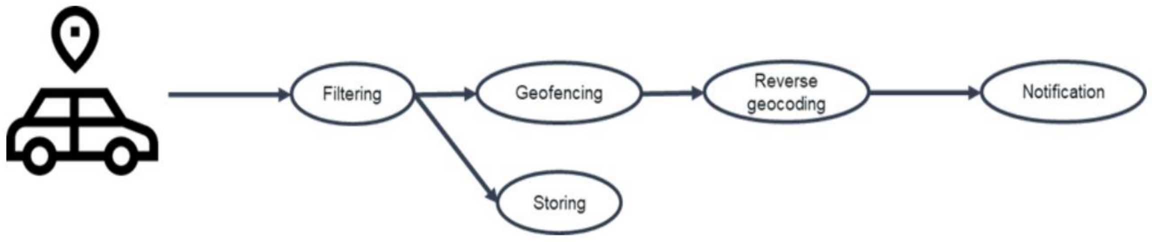


Figure 8: Vehicle fleet monitoring IoT data pipeline example

Monitored vehicles are equipped with a data logger that collects information about GPS position of the vehicle and publishes it to the NebulOuS input queue. GPS monitoring is known to be error prone, so the user wants to perform an outlier detection on the raw data (filtering step). The applied algorithm needs to know the last N readings to classify the Nth+1 reading. As a result, this outlier detection can be parallelized up to one instance of the step per vehicle and must always be guaranteed that readings from a vehicle are processed by the same instance. Next step of the process is to detect if the vehicle leaves a delimited geographic area for longer than a certain period (geofencing step). Again, this operation can be parallelized but the events from one vehicle need to be processed by the same worker. If a vehicle leaves the designated area, a notification is to be sent to the manager of the fleet. This notification contains a human readable message about the location of the vehicle (e.g: street address). For this, a reverse geocoding process needs to be executed on the latitude, longitude data of the egress event (reverse geocoding step). This step can be parallelized infinitely, as each event can be processed separately. With the result of the reverse geocoding, a notification step is responsible for managing the actual sending of the notification via mail (notification step). This

[35] https://druid.apache.org/

[36] https://www.influxdata.com/products/influxdb-overview/

[37] http://opentsdb.net/

[38] https://graphiteapp.org/

operation doesn't allow for parallelization. After the filtering step, processed data is to be stored on a time-series database (storing step). The parallelization level of this step is limited by the number of concurrent connections to the underlying storage system (e.g., 5).

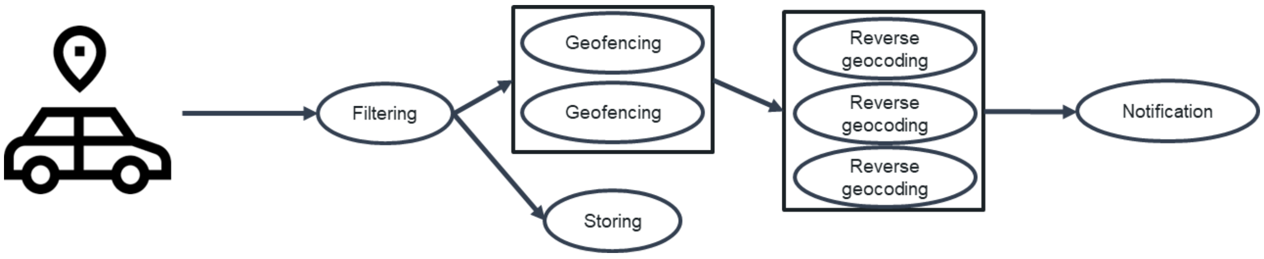


Figure 9: Vertical scalability of the IoT data pipeline example

One of the key requirements of such applications is the ability to scale vertically part of the pipeline depending on the workload. For instance, Figure 9: Vertical scalability of the IoT data pipeline example shows a situation where several steps of the application (Geofencing and Reverse geocoding) have been vertically scaled to parallelize the process of the step workload during a period of high pressure on these steps.

In NebulOuS, we understand a data processing pipeline as a directed acyclic graph of steps. A step is a logic block that receives data from an input stream, does some computation over it and produces results in one or more output streams. In the example above, five steps can be identified: i) filtering, ii) geofencing, iii) reverse geocoding, iv) notification v) storing.

The proposed semantics for modelling IoT data processing pipelines allow the user to indicate the structure of the pipeline, the level of parallelism allowed for each step of the flow and the workload distribution criteria among these parallel instances. On top of that, the user is also able to indicate the SLO constraints/optimization function that governs the actual deployment of the pipeline in terms of number of parallel instances of each step and number of resources (CPU/RAM/GPU, etc…) dedicated to these instances. With this, the user is capable of leveraging the capacity of the meta-OS to i) adjust the number of instances of each of the steps of the pipeline as well as the CPU/RAM dedicated to each of these instances depending on the current workload of the application (e.g. number of messages per second) and ii) automate the deployment of the pipeline and re-configuration when the workload changes.

4.3.1.1 Definition of the data processing pipeline

Data processing pipelines (pipeline so forth) are defined as regular applications managed by NebulOuS. As any other application inside NebulOuS, the user needs to provide its application graph (Open Application Model or OAM), the service level objective (SLO) and the objective function.

The application graph of a pipeline consists of the set of steps. A step is a logic block that receives data from an input stream does some computation over it and produces results in one or more output streams. The solution relies on ActiveMQ queues (offered by the IoT pub/sub mechanism described in 4.1) to articulate the communication between pipeline steps: passing of messages between them and collecting audit logs of their execution. Each step of this pipeline is mapped to a "component" in the OAM file. For each of these steps, the application owner can specify the constraints on the computing node that is going to host that step (max/min RAM, CPU, GPU, disk, etc..) as well as all the horizontal scalability of the step (range of number of instances that can run in parallel).

In order to control how messages are distributed across the different instances of each step. A configuration file is required to be provided by the application owner (devops). This file, in JSON format contains, for each step of the pipeline (identified by its component name in the OAM file), an object with the following information:

- input_stream: The name of the stream from where the step will consume data.
- grouping_key_accessor: Information on how to extract the key for grouping messages.
    - source: property|body_json|body_xml
    - expression: string.
        - In case of source == "property". The grouping_key for a message will be the value of the message property with key "expression".
        - In case of source == "body_json". The grouping_key for a message will be the string serialized value of the JSON path "expression" of the message body.
        - In case of source == "body_xml". The grouping_key for a message will be the string serialized value of the XML path "expression" of the message body.

To define the SLO of each step, user can leverage the metrics automatically reported by the IoT/FOG Pub/Sub mechanism for each queue (messages_count, max_message_age, consumers_count, groups_count) and/or metrics derived from the message lifecycle monitoring (wait time, processing time, volume of data exchanged, etc…) (as described in 4.1) to define independent SLOs for each of the pipeline steps. Usually, these SLOs would indicate the max processing latency of messages for a certain step or the max length of the input queue for the step.

4.3.2 Conceptual Architecture & Implementation details

The proposed approach relies on the use of the IoT/FOG pub/sub mechanism described in section 4.1. It utilizes Artemis to create a cluster of pub/sub brokers that allows the pipeline instances steps to communicate between them. To constrain how messages sent to a certain queue are to be distributed among the different instances of the step that are running, the solution relies on the message grouping capabilities of Artemis[39]. This functionality permits to mark messages with a JMSXGroupID attribute. With this Artemis guarantees that all messages with the same JMSXGroupID value will be sent always to the same client that is subscribed to the queue where the message resides. Should the client be disconnected (due to failure of the client or scale-in/down of the deployment topology), another client subscribed to the queue would be selected and start receiving the messages. At any given time, a client subscribed to a queue can receive messages from many JMSXGroupIDs.

To simplify the process of marking the messages with the appropriate value of JMSXGroupID to achieve the desired workload distribution among parallel instances of each of the IoT transformation pipeline steps, we have developed an Artemis plugin that handles this task (MessageGroupIDAnnotationPlugin). Once the Artemis broker starts, it retrieves the JSON file containing the pipeline definition provided by the application DevOps (as explained in section 4.3.1.1). After that, the broker is ready for receiving new messages from the producers. Once a message is received, the plugin sets the value for the JMSXGroupIDs of the message according to the metadata defined in the pipeline definition file.

Since many steps can consume the output produced by a single step (e.g: in the example APP geofencing and storing steps are consuming the results from the filtering step), but each step might require different groupings for the message. To fulfil this requirement, it is necessary to create an address for each consuming step and divert there the messages sent to the producing step output

[39] https://activemq.apache.org/components/artemis/documentation/latest/message-grouping.html#message-grouping

address. The developed ProcessingPipelineManagementPlugin handles this process automatically on startup of the broker.

4.3.3 Usage Example

This section utilizes the example application described Figure 8: Vehicle fleet monitoring IoT data pipeline example to provide a summary of how the proposed Data processing pipeline orchestration approach can be applied to an user application.

For this application, the NebulOuS user (DevOps) should register a new OAM similar to the draft definition described in Table 8 8.

Table 8: Draft of OAM for the Vehicle fleet monitoring APP.

```
kind: Application
metadata:
  name: vehicle-fleet-monitoring
spec:
  components:
    - name: pub-sub-master
      type: webservice
      properties:
        image: activemq/artemis
        volumeMounts:
          configMap:
            - name: broker-config.xml
              mountPath: /broker-path/etc-override/broker.xml
            - name: iot_dpp_libs
              mountPath: /broker-path/libs
            - name: pipeline.json
              mountPath: /broker-path/pipeline.json
    - name: filtering-step
      type: webservice
      properties:
        image: vehicle-fleet-monitoring/filtering-step
        traits:
            - type: "scaler"
                properties:
                  replicas: [1,7]
```

```
   - name: geofencing-step
     type: webservice
     properties:
       image: vehicle-fleet-monitoring/geofencing-step
       traits:
           - type: "scaler"
             properties:
               replicas: [1,7]
   - name: reverse-geocoding-step
     type: webservice
     properties:
       image: vehicle-fleet-monitoring/reverse-geocoding-step
       traits:
           - type: "scaler"
             properties:
               replicas: [1,7]
…
```

For brevity reasons, this draft OAM definition doesn't include many of the steps from the sample application. Also, instead of using DaemonSets to deploy a cluster of Apache Artemis brokers, it relies on a centralized broker.

In this case, the OAM file contains the components relative to three steps of the pipeline: filtering, geo-fencing and reverse-geocoding. For each step, an OAM component is registered. For each step, the trait "scaler" is assigned a range from 1 to 7. This indicates that the step can have from 1 to 7 instances running in parallel.

Another key aspect of the example is how the Apache Artemis component is registered. There, volume mounts are used to provide: The broker configuration file, the JAR containing the plugin implementation and the pipeline definition file.

The broker configuration file is an XML that contains all the configurations for the broker as described in the Apache Artemis documentation[40]. There, the user must register the following plugins:

- **eut.nebulouscloud.iot_dpp.QueuesMonitoringPlugin:** Apache Artemis plugin that periodically collects usage metrics from the queues of the broker (messages_count, max_message_age, consumers_count, group_count) and publishes them to EMS.
- **eut.nebulouscloud.iot_dpp.monitoring.EMSMessageLifecycleMonitoringPlugin:** ActiveMQ Artemis plugin for tracking the lifecycle of messages inside an ActiveMQ cluster. On

[40] https://activemq.apache.org/components/artemis/documentation/latest/configuration-index.html

Funded by the European Union

each step of the message lifecycle (a producer writes a message to the cluster, the message is delivered to a consumer, the consumer ACKs the message), the plugin generates events with relevant information that can be used for understanding how IoT applications components on NebulOuS are communicating.

- **eut.nebulouscloud.iot_dpp. MessageGroupIDAnnotationPlugin:** ActiveMQ Artemis plugin that sets, for any incoming message, the appropriate value for the JMSXGroupID according to the configuration found on the pipeline definition file.
- **eut.nebulouscloud.iot_dpp. ProcessingPipelineManagementPlugin:** Apache Artemis plugin responsible for creating necessary diverts for message addresses to allow multiple pipeline processing steps to consume the outputs from the same preceding step but using different grouping strategies.

The registration of the plugins is achieved including the following section in the XML configuration file of the broker.

Table 9: Sample section for registering Apache Artemis plugins

```
<broker-plugins>
  <broker-plugin class-name="eut.nebulouscloud.iot_dpp.QueuesMonitoringPlugin"></broker-plugin>
  <broker-plugin class-name="eut.nebulouscloud.iot_dpp.EMSMessageLifecycleMonitoringPlugin
"></broker-plugin>
  <broker-plugin class-name="eut.nebulouscloud.iot_dpp.MessageGroupIDAnnotationPlugin "></broker-
plugin>
  <broker-plugin class-name="eut.nebulouscloud.iot_dpp.
ProcessingPipelineManagementPlugin"></broker-plugin>
</broker-plugins>
```

Regarding the JAR containing the plugin implementations, its source code can be found on https://review.opendev.org/admin/repos/nebulous/iot-dpp-orchestrator,general

Regarding the pipeline definition file, the DevOps must provide a file with the following contents:

Table 10: Pipeline definition for the Vehicle fleet monitoring APP

```
{
 "filtering_step": {
  "input_stream": "raw_data",
  "grouping_key_accessor": {
   "source": "body_json",
   "expression": "vehicle_id"
  }
 },
 "geofencing_step": {
  "input_stream": "filtering_step_output",
  "grouping_key_accessor": {
   "source": "body_json",
      "expression": "vehicle_id"
    }
  },
  "reverse_geocoding_step": {
    "input_stream": "geofencing_step_output"
  }
}
```

There, information on the input of each step and how messages are grouped is provided:

- **filtering_step**: that consumes messages published on the "raw_data" address (there is where vehicles should publish the GPS readings). Filtering of incoming messages can be parallelized but it must be guaranteed that all messages from the same vehicle are processed by the same step instance. For this, this step states that messages should be grouped by the value of the attribute "vehicle_id" found in the body of the messages.
- **geo_fencing_step**: consumes from "filtering_step_output" address (that is where the workers from the "filtering_step" publish their output). Again, multiple instances of "geo_fencing_step" can exist, but messages from the same vehicle must always be processed by the same instance of the step. For this reason, the "grouping_key_accessor" is provided.
- **reverse_geocoding_step:** consumes from "geofencing_step_output". This time, no special requirements on the distribution of messages needs to be imposed, for this, "grouping_key_accessor" is not informed.

With this configuration, once the application is deployed, the metrics and events described in section 4.1.2 Interfaces offered/required are published on the EMS and can be used to define the SLO for the application and the optimization criteria.

4.3.4 Next steps for the final iteration of the mechanisms

Having implemented the fundamentals for modelling IoT data processing pipelines, efforts for the task "Task 5.3: Interoperable IoT/Fog data collection and management" will focus on validating the proposed approach. This will be handled following two different directions. First, a laboratory-scale test application will be modelled using the proposed approach and workload simulations will be conducted to evaluate how NebulOuS core handles the re-adaptation needs. In parallel, the approach will also be validated using use cases already present in the project (Crisis management use case) and other use cases that might join the project as part of the Open Calls. The findings obtained from these activities will help to identify new metrics that could be beneficial for expressing the SLO's of the applications and the optimisation function. Also, the laboratory scale application can be made publicly available to let NebulOuS adopters better understand how IoT data processing pipeline oriented applications can be modelled.

5 SELF-ADAPTIVE RECONFIGURATION ENACTMENT

Reactive application management means that there is a Service Level Objective (SLO) violation event triggering the adaptation, and then all the metric values representing the application's current execution context is frozen while the Solver finds a configuration that is optimized for the measured application execution context. This approach can always be used; however it takes time to compute the optimized configuration and to reconfigure the deployed components. This may lead to a significant lag between the time point when the measurements were taken, and the time point when the optimized application is ready and running. The benefit is that the reactive adaptation is computed on exact information available at the time of the SLO violation detection. Alternatively, one may predict the measurements and the SLO violation events to a future time point long enough into the future to complete the optimization and the adaptation of the adaptation before this time point is reached. This is obviously better provided that the prediction is exact. But since it must be based on machine learning, which assumes that the future will look similar to the past, there will always be a discrepancy between the predictions and the real execution context when the reconfigured application is ready for use, and this discrepancy increases with the length of the prediction horizon.

Proactive adaptation was piloted in the MORPHEMIC[41] platform developed in the Horizon 2020 project of the same name. The process introduced in MORPHEMIC is similar in NebulOuS. First, predicted monitoring metric values are being received through the communication broker and are being stored in an InfluxDB database. Each application features its own bucket, therefore allowing for the isolation of data between applications. Then, forecasters are able to query the database, create the datasets which are needed for their operation and create predictions. In NebulOuS we aim to reuse at least one forecaster from MORPHEMIC, which is based on exponential-smoothing available here[42]. After, the Prediction Orchestrator collects and ensembles these predictions to a final predicted value. This value is then used by the SLO Violation Detector to create a reconfiguration alert whenever necessary. The process is illustrated in Figure 10 showing the detection of an SLO violation event at t_3 and where the reconfigured application is only available for use at t_{28}. The Solver in MORPHEMIC used an external module to calculate the utility value for a proposed configuration, and the Utility Generator used again a Performance Predictor to forecast the effect of the proposed configuration on performance indicator values depending on the configuration. For instance, the response time of an application is directly measurable and influenced by the current application configuration and execution context. The proactive version predicted the future application execution context vector for t_{28} denoted $\hat{\boldsymbol{\theta}}(t_{28})$ and this was used in the optimization loop leading to the deployment of the optimized configuration $\boldsymbol{c}^*(t_{28})$.

It is necessary to distinguish between configuration independent metric values that can be trivially measured and predicted by time series prediction methods, and the performance indicators that must be calculated using regression on the independent metrics and the running configuration. The estimation of the regression functions will be done by the Performance Module in NebulOuS, and the evaluation of both the utility function and the regression values can then be directly computed by the Solver. More details are available in the NebulOuS deliverable D3.1 [12]. The prediction of the

[41] https://www.morphemic.cloud/

[42] https://opendev.org/nebulous/exponential-smoothing-predictor

independent metric values and the prediction of a possible future SLO violation event is discussed in the following.

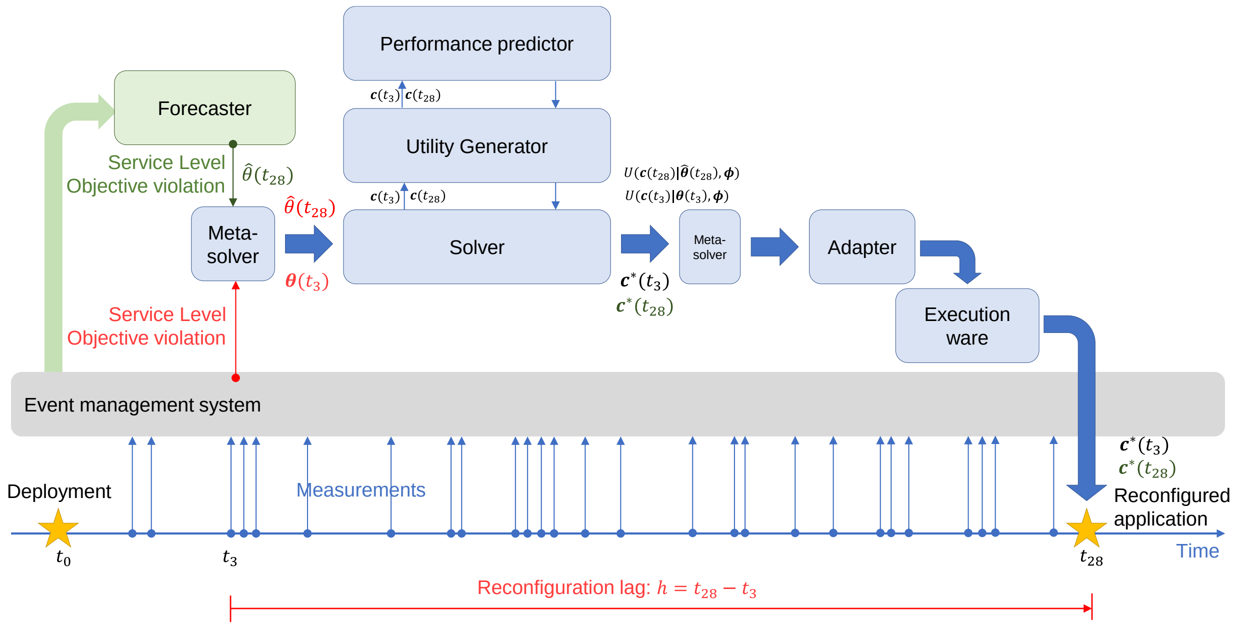


Figure 10: The application optimisation inherited from the MORPHEMIC project showing both the reactive optimisation to a Service Level Objective violation at time t_3 leading to the new reconfiguration deployed at t_{28}, and the proactive version where the metric values are predicted for t_{28}.

Once the new configuration has been calculated, it must be enacted. This is done by two critical components: The Adapter and the Deployment Manager called the "Executionware" in MORPHEMIC. The Adapter computes the difference between the running application and the configuration to be enacted, keeps the resources the two configurations have in common and instructs the Deployment Manager to create the missing resources and free the resources that will no longer be used. The Adapter is a part of the Optimiser Module and the initial version of the Deployment Manager is discussed in *D4.1 Initial Orchestration Layer & Security-enabled Overlay Network Deployment*.

5.1 SLO VIOLATION DETECTOR

5.1.1 Introduction

The Severity-based SLO Violation Detector, is a component which is responsible for the recognition of the need for a reconfiguration. It allows an application to function smoothly, by preserving at runtime the constraints the DevOps has provided. Using its help, the platform can save resources by not continuously running resource-hungry optimization algorithms, but focusing instead on the points in time, when the operational thresholds the DevOps has set have been violated. The role of the component can be better understood by using the following diagram:

Funded by the European Union

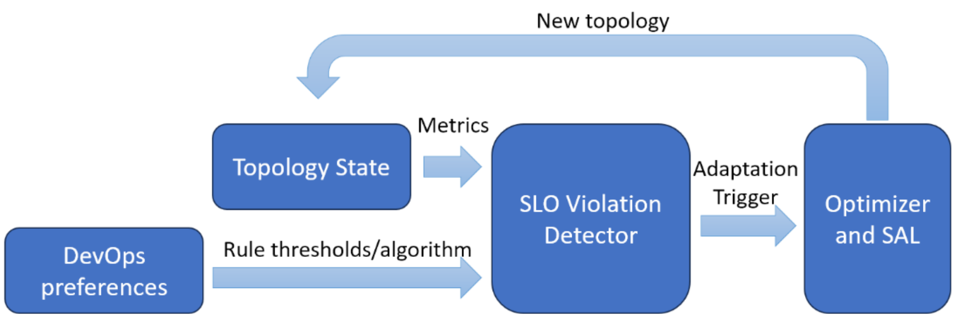


Figure 11: The interactions of the Severity-based SLO Violation Detector within the Nebulous platform

The SLO Violation Detector (or SLOViD) draws its roots directly from the respective component which was introduced in the Morphemic project. Of course, a lot has changed in this component, which most importantly has been reworked and re-architected to satisfy the demanding multi-application needs of Nebulous.

5.1.2 Approach

The most important challenge which is presented in the context of Nebulous for the SLO Violation Detector, second only to dynamically improving its behaviour, is the support for multiple applications. To actually handle the case, a major refactoring was required from the source code which was inherited. The general architecture of the component will be outlined in the following section 5.1.3. order to extend the functionality of the component, the mavenized structure which was employed for the Morphemic version, was mostly kept intact. Additional or more refined concepts were introduced in existing classes, but where needed additional classes were created. However, code packages were kept the same.

Seizing the opportunity of the code refactoring, a transition to a new AMQP communication library introduced by EXN[43] was performed. Moreover, some new functionality in the form of elementary support for a Spring Boot API was added. Therefore, the SLO Violation Detector can now spawn new detection engines both through a REST API and through ActiveMQ.

5.1.3 Conceptual Architecture

The architecture of the SLO Violation Detector appears in Figure 10. In the above figure, we present the internal architecture of the SLO Violation Detector, and the main input and output of the component. As it can be seen, the SLO Violation detector receives functional input from the NebulOuS platform, in the form of three messages-events arriving in the Nebulous broker: The "SLO Rule" message, the "Metrics list" message and the "Device lost" message. Out of these, the most important is the "SLO Rule" message, which contains the actual SLOs which should be respected by the application. The "Metrics list" message defines the metrics which should be predicted for the particular application, and therefore indirectly indicates to the SLO Violation Detector some metrics which will be exposed by the application and will possibly need a subscription. It is very important, as it now (unlike Morphemic) contains information on the estimated maximum and minimum values of a metric, therefore minimizing the need for coarse, generic estimations. Finally, the "Device lost"

[43] https://opendev.org/nebulous/exn-connector-java

message indicates that a device was lost from the platform and therefore it is necessary to signal that a reconfiguration will be (possibly) needed.

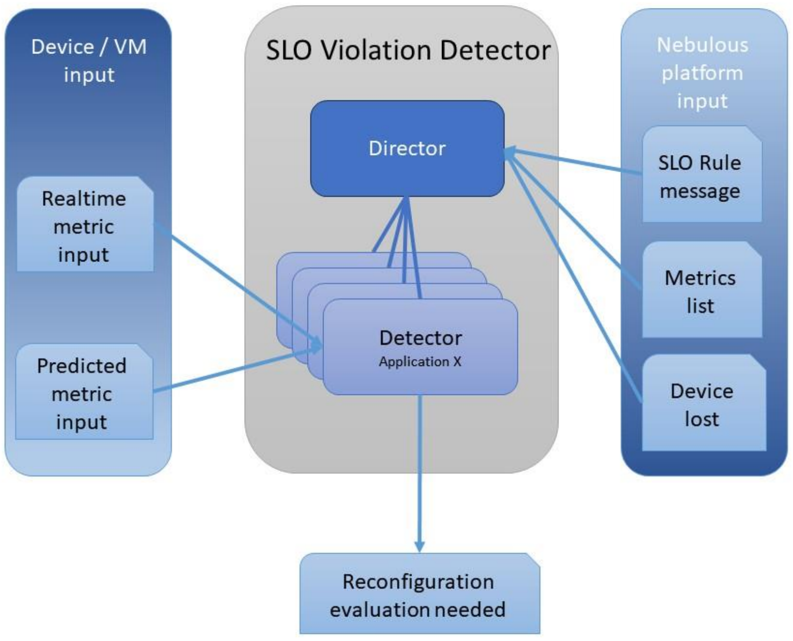


Figure 12 The interfaces of the SLO Violation Detector and its internal conceptual architecture

As underlined in the previous section, the main change from the Morphemic platform which should be underlined in this deliverable is the refactoring of the SLO Violation Detector, in order to be able to support multiple applications. In the architectural view of Figure 10 above, this is implied by the handling of all communication with actual VM instances and edge devices spawned by NebulOuS, by different detector instances.

The complete functionality of the component, from the registration of an application to it, until a reconfiguration is sent, is the following:

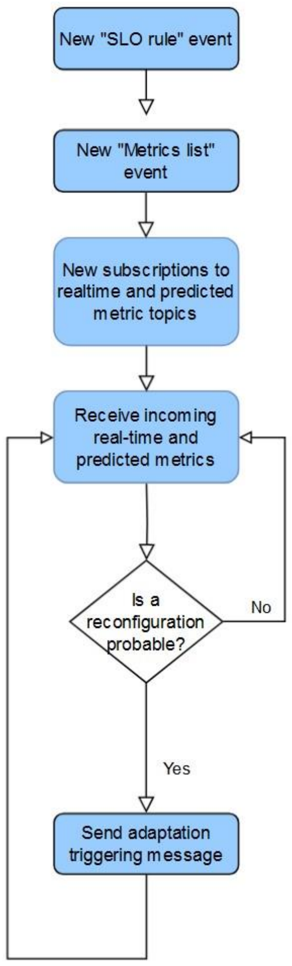

New "SLO rule" event
New "Metrics list" event
New subscriptions to realtime and predicted metric topics
Receive incoming real-time and predicted metrics
Is a reconfiguration probable?
No
Yes
Send adaptation triggering message

Figure 13 The flow of processing for a single SLO rule within the SLO Violation Detector, starting from the reception of its definition, and ending with the publication of the alert event and the reception of new real-time and predicted monitoring data

As Figure 11 portrays, for an application to be considered by Nebulous, it is first necessary to receive an "SLO rule" event. While in Morphemic this event was received by a monolithic SLO Violation Detector, and triggered the functionality described in steps 2,3 and 4, in Nebulous the same functionality is accomplished by two different sub-components - the Director and the Detector subcomponents respectively. In the context of Nebulous, one Director is always assumed, which guides one or more Detector subcomponents. When a new "SLO rule" message arrives, the Director subcomponent instructs the creation of a new Detector subcomponent. Each Detector is then responsible to subscribe to real-time and predicted metrics which are relevant to the SLOs for the particular application, which are expected to be received from EMS clients and the Nebulous forecasting output (therefore, if the same metric is used in different applications the SLO Violation detector can keep values distinct per application, associated to their own Detector). Real-time and predicted values can be received both in regular and irregular time intervals.

To understand whether a reconfiguration is necessary, the SLOViD Detector subcomponent calculates the Severity [13] of the current situation, as this is perceived through (at most) three different meta-metrics: PrConf, Delta and Rate of Change. All of these meta-metrics take into account not only the real-time values of monitoring metrics but also the predicted values which are provided by Nebulous forecasting. PrConf is the probability confidence associated with a prediction for a particular monitoring metric, multiplied by the normalized confidence interval width. Delta, refers to the differential between the predicted value and the real-time value of the metric (the real-time value is calculated as the average of a configurable number of past observations). Rate of change is self-explanatory – it calculates the rate of change of a metric, by calculating the normalized ratio of the difference between the prediction and the real-time value of a metric, over the current real-time value. When using the Prconf-Delta method to calculate Severity, only the product of the two meta-metrics (PrConf and Delta) is used. On the other hand, calculating Severity using the All-metrics method involves a typical calculation of Severity using all three meta-metrics. For more details, and illustrative examples, the reader is referred to the documentation provided as part of the relevant work in Morphemic project [14].

If the All-metrics method has been selected to calculate the final Severity value, the Severity value which is calculated is used as the probability of an SLO violation (capped at 100%). On the other hand, if the PrConf-Delta method has been chosen, the Severity value is compared with the overall median Severity value which can be calculated for all inputs (equal to 0.0652), assuming normalized Severity values from 0 to 1. This median value is used as the 50% reconfiguration probability value, and a 100% probability value is assigned to any Severity value equalling (or being greater than) 1. Although this assignment of a probability to a Severity value is justified, it is not the sole possible assignment. Therefore, we plan to improve this assignment of probabilities to Severity values as future work for this component (see also the suggested approach in Section 5.1.5).

For more details on the calculation of the Severity value, the reader is directed to the documentation of the component provided in D2.2 of the Morphemic project [14].

5.1.4 Interfaces offered and required

The Severity-based SLO Violation Detector component requires a number of interfaces in order to process the messages which are illustrated in Figure 10. In the following table 11, the interfaces offered and required by the component are described. The majority of the interfaces of the component are kept the same as in Morphemic; Still, they are briefly repeated here for completeness, with the addition of the Device lost interface. It should be noted that the 'Device lost' interface is an inbound interface for the SLO Violation Detector, and when a message is received in it a new reconfiguration message is triggered and published.

Table 11: The interfaces offered and consumed by the SLO Violation Detector

Interface name	Description	Related topics	Sample event
Monitoring Data – real-time metrics	This interface is related to the acquisition of real-time and predicted monitoring metrics	eu.nebulouscloud.monitoring.realtime.{METRIC_NAME}	`{ "metricValue": 12.34, "level": 1, "component_id":"postgresql_1", "timestamp": 163532341 }`
Monitoring Data – predicted metrics	This interface is related to the acquisition of predicted monitoring metrics	eu.nebulouscloud.monitoring.predicted.{METRIC_NAME}	`{ "metricValue": 12.34, "level": 1, "timestamp": 163532341, "probability": 0.98, "confidence_interval" : [8,15] "predictionTime": 163532342, }`

Metrics list	This interface is related to the acquisition of information on the metrics which will be monitored	eu.nebulouscloud.monitoring.metric_list	{
			"name": "_Application1",
			"version": 1,
			"metric_list": [
			{
			"name": "metric_1",
			"upper_bound": "100.0",
			"lower_bound": "0.0"
			},
			{
			"name": "metric_2",
			"upper_bound": "Infinity",
			"lower_bound": "-Infinity"
			},
			{
			"name": "metric_3",
			"upper_bound": "10.0",
			"lower_bound": "-4.0"
			}
			]
			}

Device lost	This interface is related to the acquisition of information on the need of Nebulous to perform a reconfiguration if a used device is lost at runtime	eu.nebulouscloud.monitoring.device_lost	{
			"device_id": "device_100",
			"application_name": "_Application1",
			"timestamp": 1626181860
			}

Reconfiguration alert	This interface provides an alert to the NebulOuS Optimizer of a possible need to reconfigure the application	eu.nebulouscloud.monitoring.slo.severity_value	{
			"severity": 0.9064,
			"predictionTime": 1626181860,
			"probability": 0.92246521
			}

New SLO rule	This interface is related to the need of the SLO Violation Detector to aid the enforcement of SLOs for a particular application	eu.nebulouscloud.monitoring.slo.new	{ "name": "_Application1", "operator": "OR", "constraints": [{ "name": "cpu_and_memory_or_swap_too_high", "operator": "AND", "constraints": [{ "name": "cpu_usage_high", "metric": "cpu_usage", "operator": ">", "threshold": 80.0 }, { "name": "memory_or_swap_usage_high", "operator": "OR", "constraints": [{

			`"name": "memory_usage_high", "metric": "ram_usage", "operator": ">", "threshold": 70.0 }, { "name": "disk_usage_high", "metric": "swap_usage", "operator": ">", "threshold": 50.0 } ] } ] } ] }`

5.1.5 Next steps

While a fair number of changes have been implemented for the SLO Violation Detector, the introduction of the capability to change the Severity threshold triggering an adaptation, or equivalently being able to change the probability of an adaptation dynamically (and based on the feedback of the topology), still needs to be implemented. Until now, the Q-learning technique has been evaluated and is the technique which will be most probably used.

By comparing the current state of the platform with the desired goals, it has been established that in order to achieve the dynamic adaptation of the behaviour of the component, a data-driven approach, or a control-theoretic approach needs to be implemented. To illustrate, let us assume a scenario in which there is a rule stating that an SLO violation should exist when the CPU usage is over 70% or the RAM usage is over 70%. Then, the following table 12 can be created (assuming that the all-metrics calculation method has been chosen).

Table 12: Scenario of a predicted SLO violation

Realtime		Predicted		Rate of change		PrConf		Delta		Severity	Probability	Success
cpu	ram	cpu	ram	cpu	ram	cpu	ram	cpu	ram			
0.7	0.8	0.8	0.9	0.14	0.13	0.9	0.9	0.33	0.66	0.5611	56.11%	100%
0.7	0.8	0.75	0.95	0.07	0.95	0.9	0.9	0.16	0.83	0.5794	57.94%	80%

Based on our preliminary analysis, we have considered that when the component performs a calculation involving an SLO rule for a particular application, the real-time and predicted values for the metrics which are involved in the rule, as well as the calculated meta-metrics, can be stored. Then, when another situation (where situation is defined as the vector of SLO rule version and the current real-time and predicted metrics for a particular application), similar to the older one has been encountered, knowledge can be extracted to derive the next action of the SLO Violation detector (announce need for reconfiguration or not). The decision can be taken by factoring in both the previous success of following the action which would otherwise be deterministically chosen (i.e., to announce the need for reconfiguration, whenever the probability is over X%) and an exploration factor which would be considered only when the success rate is less than 100% for the particular situation. The success rate itself could be defined as the capability of the platform to reach an estimated Utility function score by adapting the current topology.

It can be understood from the above example that Q-learning fits the vision to implement adaptive behavior within the SLO Violation Detector. However, the final decision on the exact technology to be used along with the relevant implementation details is something which will be reported in D5.2

5.2 PREDICTION ORCHESTRATOR

The Proactive Adaptation Approach is central to enhancing the precision of predictive models through the integration of a wide array of forecasting tools. This methodology is predicated on the principle that a broader spectrum of forecasting inputs contributes to more accurate and reliable predictions across various metrics. The linchpin of this approach is the Prediction Orchestrator, which fulfils several pivotal functions:

- Allocation of forecasting tasks: The Prediction Orchestrator delineates the specific metrics that each forecaster must predict, including the necessary frequency of these predictions, thereby ensuring a structured and systematic forecasting process.
- Centralization of forecasts: By consolidating the forecasts from diverse sources into a unified repository, the Prediction Orchestrator facilitates a comprehensive overview of predictive insights.
- Oversight and validation: The Prediction Orchestrator rigorously evaluates the aggregated forecasts to ascertain their relevance to future timeframes while securely archiving this data, thus ensuring its integrity and applicability.
- Synthesis of predictions: Employing advanced analytical techniques, the Prediction Orchestrator amalgamates the disparate forecasts into a singular, refined prediction for each metric and temporal juncture, thereby enhancing the overall predictive accuracy.
- Dynamic response to updates: In instances where forecasters revise their predictions, the Prediction Orchestrator swiftly recalibrates the consolidated forecast to reflect these updates, maintaining the currency and relevance of predictive insights.

The foundational assumption of this approach is the existence of at least one forecasting method and a Prediction Orchestrator. In scenarios where a singular forecasting method yields direct 'final' predictions, the conventional procedural sequence is bypassed. Here, the 'final' predictions hinge on pre-established parameters for the prediction horizon and the quantity of future forecasting intervals, necessitating merely a singular initiation message (start_forecasting, type I).

Each forecasting method is tasked with generating predictions for designated metrics, extending into the future as specified. The inception of forecasting, termed 'epoch start', alongside the 'prediction horizon', are critical components. The prediction horizon is defined by the minimal duration requisite for an adaptation, rendering forecasts for intervals shorter than this horizon as extraneous.

For instance, should the epoch start be set at 1705046500, with a prediction horizon of 120 seconds (2 minutes) and a requirement for 5 sequential forecasts, a method is expected to instantaneously generate forecasts for timestamps 1705046620, 1705046740, 1705046860, 1705046980, and 1705047100. If these forecasts are produced at 1705046750, the initial two would be obsolete, and the third too proximate to the prediction horizon (110 seconds), leaving only the latter two forecasts as relevant.

This process is facilitated through various message types:

Type I Messages initiate the forecasting process, delineating the metrics to be predicted by a specific method and are conveyed via the designated topic.

The forecasting initiation involves a start_forecasting event, which outlines the metrics to be forecasted by a specific method. This message is sent to the eu.nebulouscloud.forecasting.start_forecasting.{prediction_method_name} topic, where {prediction_method_name} is a placeholder.

An example of a Type I message structure is:

```
"application_name"  "_Application1"
"metrics"   "cpu_usage"
"timestamp"  1705046535
"epoch_start"  1705046500
"number_of_forward_predictions"  5
"prediction_horizon"  120
```

- metrics: A list specifying which metrics are to begin forecasting.
- timestamp: The epoch time when the message is sent.
- epoch_start: The starting point for forecasting.
- number_of_forward_predictions: How many forecasts are to be made per metric.
- prediction_horizon: The time interval in seconds between each forecast.

Type II Messages encapsulate the predictions formulated following a Type I message and are published to a specific monitoring data interface topic.

Predictions should be formatted as Type II messages for the monitoring data interface and published to the: eu.nebulouscloud.monitoring.preliminary_predicted.{prediction_method_name}.{metric_name} topic, with placeholders for {prediction_method_name} and {metric_name}.

Type III Messages, while not obligatory, signal the termination of a forecasting method's activity, typically due to suboptimal performance, and specify the metrics for which predictions should cease.

The Prediction Orchestrator will play an instrumental role in this ecosystem, orchestrating the forecasting activities by directing methods on the metrics and timing for predictions, collating and scrutinizing forecasts, and refining these into more accurate predictions through sophisticated techniques. These final predictions will be then relayed to the SLO Violation Detector and the Optimiser ensuring that the system's adaptive responses are informed by the most accurate predictive insights available.

5.3 ADAPTER

The Optimizer module consists of multiple internal components, see the deliverable *D3.1 Initial NebulOuS Brokerage and Resource Management*. The predicted values for metrics that are used in the utility function and the problem constraints are collected by the Solver. When the Solver is triggered by the SLO Violation Detector to produce a solution, the Solver produces the solution that maximizes the utility of the application for the farthest time point for which it has a prediction assuming zero-order hold for all other metrics, i.e., that all metric values predicted are valid until the next prediction arrives.

The Adapter is an internal component of the Optimiser Controller receiving the next application configuration produced by the Solver. This configuration is compared to the running configuration, and a list of nodes to add and a list of nodes to remove are produced to plan the adaptation. Note that both horizontal and vertical scaling are supported: the solver might decide to vary the number of replicas an application component runs on (horizontal scaling), and/or might choose to redeploy a component on a node with different requirements (vertical scaling).

After calculating the necessary changes, the Optimiser Controller calls endpoints on the Scheduling Abstraction Layer (SAL) to create and shut down nodes as needed. After this process, the node configuration for the application will correspond to the new application configuration computed by the Solver. The adapter creates new nodes, submits the KubeVela file to deploy the containers of the application on the new node configuration, then shuts down the retired nodes after KubeVela has computed the difference between the running application pods and the requested pods and restart deployed the application components according to the new scenario.

Concretely, the sequence of Adapter actions performed during the reconfiguration of the application is as follows:

1. The adapter updates the KubeVela application model file by replacing values of resource requirements with the values produced by the Solver.
2. It scans the updated Kubevela file to gather Node Candidates requirements.
3. It calls the Cloud/Fog Service Broker to get the available Node Candidates fulfilling the requirements. The Cloud/Fog Service Broker returns the list of Node Candidates together with their ranking. Node Candidates are ranked based on the preferences which are submitted for the particular application, in terms of a selection of criteria with different weights. Details of Cloud/Fog Service Broker and how the ranking is calculated are provided in *D3.1 Initial NebulOuS Brokerage and Resource Management*.
4. The Adapter chooses the Node Candidates with the highest rank. Node Candidates can be either Edge or Cloud Nodes.
5. It compares the chosen Node Candidates with the Nodes that are currently used.
6. It asks SAL to create new nodes and add them to the application cluster.
7. It sends the updated KubeVela file to SAL. SAL reconfigures the application: components are moved to the newly created nodes.
8. Once the application is running on the new configuration, the Adapter calls SAL to remove any superfluous nodes from the cluster and stop them.

This behaviour ensures that the time when the application is not operating is minimized to be only while the KubeVela framework updates the application, without waiting for the nodes to be started.

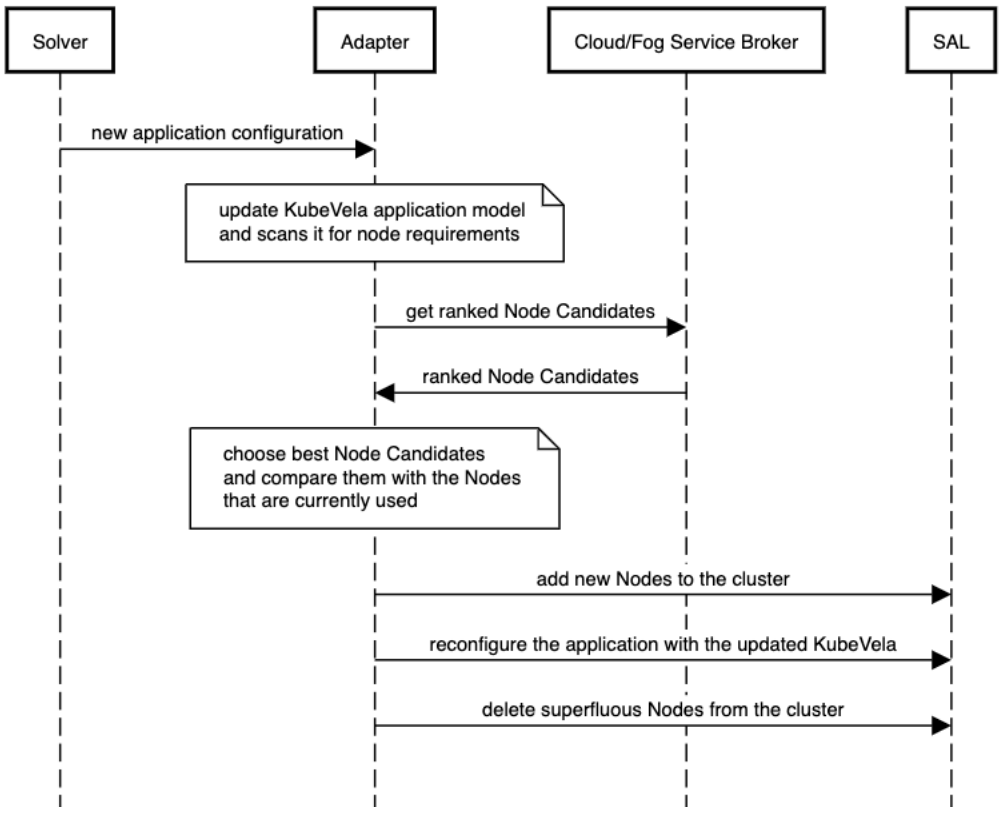

Solver
Adapter
Cloud/Fog Service Broker
SAL
new application configuration
update KubeVela application model and scans it for node requirements
get ranked Node Candidates
ranked Node Candidates
choose best Node Candidates and compare them with the Nodes that are currently used
add new Nodes to the cluster
reconfigure the application with the updated KubeVela
delete superfluous Nodes from the cluster

Figure 14: The sequence of reconfiguration actions performed by Adapter

5.4 RECONFIGURATION ENACTMENT MECHANISM

The *Deployment Manager (SAL*[44]*)* component serves as the foundational element within the *NebulOuS* solution, with its core functionalities outlined in *D4.1: Initial Orchestration Layer & Security-enabled Overlay Network Deployment*. By adhering to the principles of monitoring, decision-making, execution, verification, and adaptation requested by the Optimizer and the Scheduler, SAL ensures the seamless adaptation of resources and services to changing demands within *NebulOuS* cloud and edge computing environment. SAL employs reconfiguration enactment mechanisms to enable dynamic adjustments within the Kubernetes cluster. These mechanisms encompass actions such as scale-in/scale-out processes through designated endpoints, updating labels of pods, and component replication. This orchestration mechanism plays a pivotal role in facilitating efficient resource utilization, effective load balancing, fault tolerance, and scalability.

5.4.1 Scale-in / Scale-out mechanism

Process for scaling in and out of the number of nodes in a cluster is supported by the SAL endpoints.

[44] https://github.com/ow2-proactive/scheduling-abstraction-layer

Scaling out involves adding new nodes to the existing cluster dynamically to accommodate increased workload or demand. The endpoint for scaling out is represented by a POST request to a specific URL. The request body contains JSON data specifying the details of the new node(s) to be added to the cluster. In this case, the JSON array includes information about the new node such as its name (e.g., worker-node), node candidate ID, and cloud ID. The request also includes a session ID header for authentication or session management purposes. Upon successful execution, the response returns JSON data confirming the addition of the new node to the cluster. The response includes details such as the cluster ID, cluster name, master node, and a list of all nodes in the cluster including the newly added one.

Scaling in involves removing existing nodes from the cluster dynamically to optimize resource usage or reduce costs. The scaling in process involves a similar mechanism where a request is made to remove specific nodes from the cluster. Upon successful execution, the response will confirm the removal of the specified node(s) from the cluster.

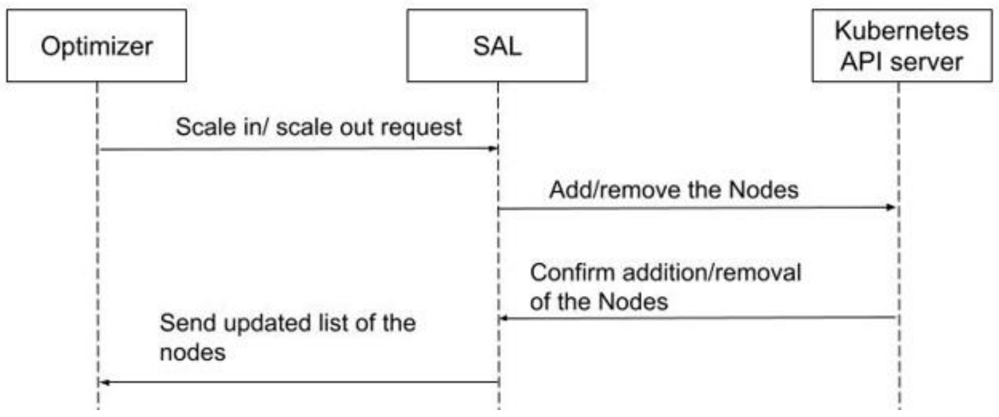

Optimizer
SAL
Kubernetes API server
Scale in/ scale out request
Add/remove the Nodes
Confirm addition/removal of the Nodes
Send updated list of the nodes

Figure : ScaleIn/Out mechanism

5.4.2 POD migration

Pod migration in a Kubernetes cluster is the dynamic process of relocating pods (containers) from one node to another, often prompted by events such as node failure, maintenance tasks, or scaling operations. When a pod migrates to a new node, it must undergo labels update to accurately reflect its updated location. Labels provide metadata regarding specific attributes of nodes, preferences or conditions to avoid during the placement, or the pod requirement such as memory, CPU or storage. The Kubernetes scheduler is responsible for determining where to deploy pods within the cluster based on various factors such as resource availability and constraints, affinity, and anti-affinity rules[45]. Labelling of a pod forces the scheduler to find feasible Nodes for a Pod and picks the most feasible ones to run the Pod according to the given label. The scheduler then notifies the API server about this decision in a process called binding. Initially, the Kubernetes cluster comprises multiple nodes, each

[45] https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

labelled with attributes indicating for instance hardware specifications or geographical location. For instance, Node A might be tagged with attributes like "zone=us-west" and "ram=4" to signify its geographical location and the amount of RAM it possesses. However, when a trigger event signals the need to decommission or repurpose Node A, its existing labels are removed through the process, which involves updating the metadata associated with Node A via the Kubernetes API server. Following the successful removal of labels of Node A, if there's a demand for additional resources in a specific geographical zone, Node B is appropriately labelled with attributes representing that zone. This process involves updating Node B's metadata with new labels, such as "zone=us-east" and "ram=8" signifying its new geographical location and increased RAM capacity. With Node B appropriately labelled, Kubernetes components can efficiently consider it for workloads requiring resources matching those attributes.

Subsequently, the redeployment process is initiated. Redeploying a Kubernetes deployment impacts the cluster's workload distribution and resource allocation, potentially leading to pods migrating to different nodes based on scheduling decisions. During redeployment, Kubernetes may terminate existing pods associated with the deployment and schedule the creation of new pods based on the updated configuration. This process ensures continuous service availability and optimal resource utilization within the cluster. Overall, pod migration, relabelling, and redeployment activities in Kubernetes clusters are vital for adapting to changes in application versions, configuration settings, or maintenance requirements, while maintaining service continuity and resource efficiency.

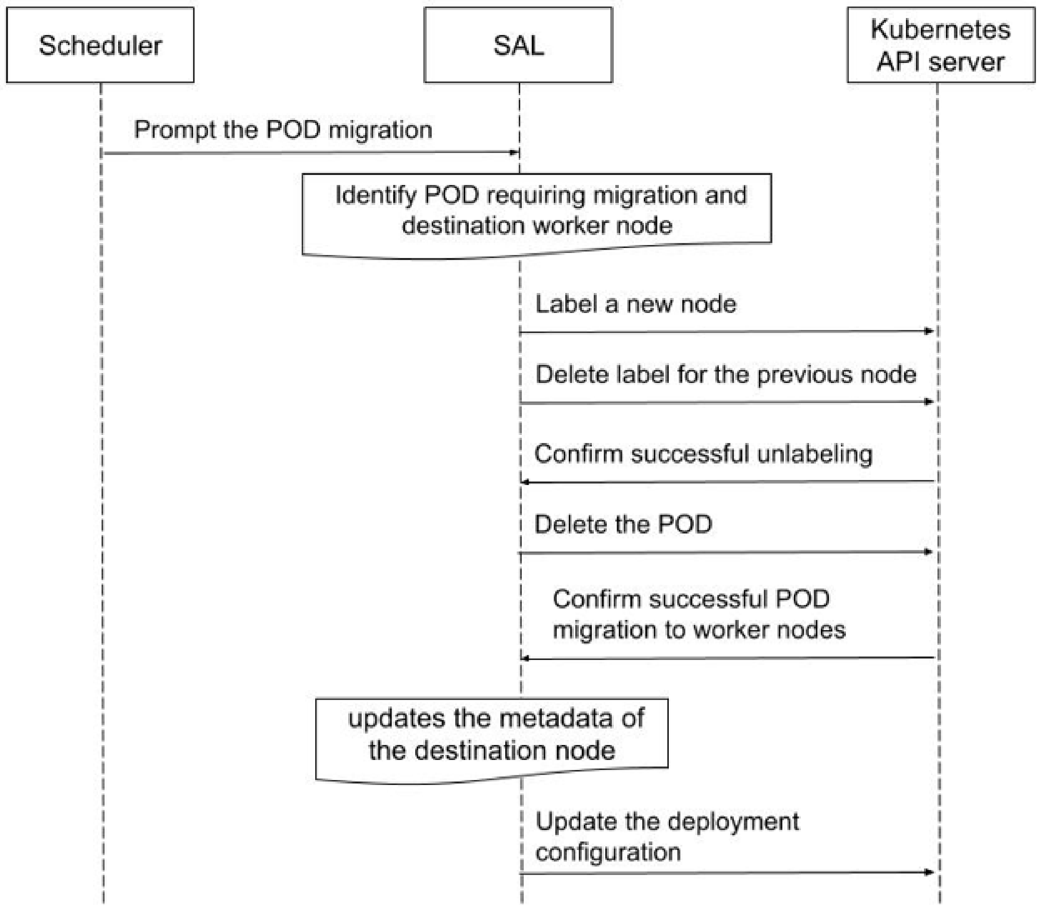

Scheduler
SAL
Kubernetes API server
Prompt the POD migration
Identify POD requiring migration and destination worker node
Label a new node
Delete label for the previous node
Confirm successful unlabeling
Delete the POD
Confirm successful POD migration to worker nodes
updates the metadata of the destination node
Update the deployment configuration

Figure : POD migration mechanism

5.4.3 Component replication

In Kubernetes cluster management, component replication involves creating multiple replicas of an application or service to improve availability, scalability, and fault tolerance. Initially, a single instance of the application is deployed within the cluster using an application deployment endpoint provided by SAL. Through this endpoint, the Optimizer configures the deployment to include multiple replicas as needed. SAL ensures the continuous maintenance of the specified number of replicas within the cluster. In response to increased demand or workload, the Optimizer dynamically scales up the number of replicas. SAL allocates these replicas to available nodes based on resource availability, constraints, and scheduling policies. The creation of additional replicas is initiated to meet the desired replication count, with SAL instantiating new pods based on the application's container image and deploying them to selected nodes. Continuous monitoring of the replicas' health and status is crucial to maintain their availability and reliability. In the event of replica failure or deterioration, automatic mechanisms are essential to replace them with new instances and sustain the desired replication count. Conversely, during decreased workload or reduced resource usage, the Optimizer scales down the number of replicas. SAL automates the termination of surplus replicas using the Kubernetes kube component, thereby optimizing resource utilization across the cluster.

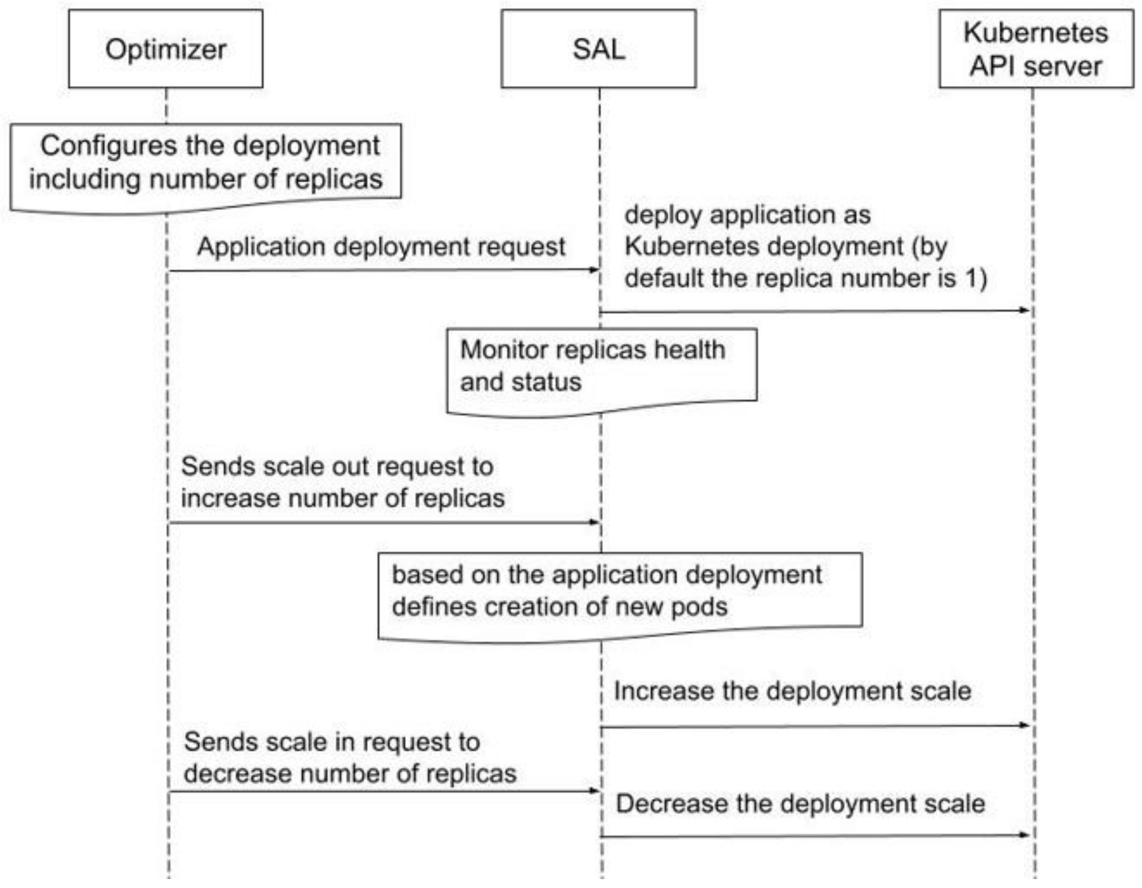

Optimizer
SAL
Kubernetes API server
Configures the deployment including number of replicas
Application deployment request
deploy application as Kubernetes deployment (by default the replica number is 1)
Monitor replicas health and status
Sends scale out request to increase number of replicas
based on the application deployment defines creation of new pods
Increase the deployment scale
Sends scale in request to decrease number of replicas
Decrease the deployment scale

Figure : Component replication

6 ASYNCHRONOUS MESSAGE-BASED API

This section describes the architecture, as well as the approach taken to ensure an event driven asynchronous architecture across the NebulOuS components. Using an event driven architecture, we aimed to promote component decoupling through separation of concerns, component scalability through asynchronous communication, and communication coherency through message address conventions and message payload structure definition and homogenization [15].

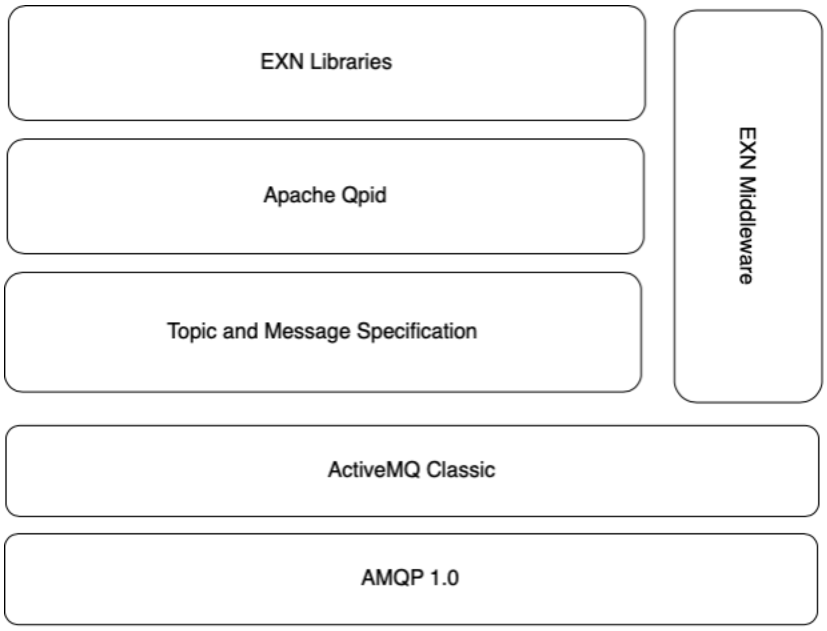


Figure 15: Event driven asynchronous architecture across the NebulOuS components Message Broker

A core component of asynchronous communication is the message broker. A message broker is a software system that enables communication between different applications, systems, or services by translating messages between formal messaging protocols. It acts as an intermediary that facilitates the exchange of messages by receiving a message from a sender (producer) and routing it to the appropriate receiver (consumer), potentially transforming or processing the message along the way. This allows for decoupling of the producer and consumer, meaning they do not need to be aware of each other's existence or be directly connected.

Amongst the most important features provided by a message broker are:

- **Message Queuing**: Temporarily holding messages until they are processed by the receiver.
- **Publish/Subscribe Model**: Allowing messages to be published to a topic and received by all subscribers to that topic.
- **Message Routing**: Directing messages from one or more producers to one or more consumers based on routing rules.
- **Message Transformation**: Converting messages from one format to another to ensure compatibility between different systems.
- **Reliability and Durability**: Ensuring messages are not lost in case of processing failures or network issues.
- **Scalability**: Handling increasing loads by distributing messages across multiple consumers or instances of the broker.

Popular message brokers include RabbitMQ[46] , Apache Kafka[47] , ActiveMQ[48] , and Amazon SQS[49] , each with its own set of features and use cases. We opted for ActiveMQ Classic[50] which provides a solid and proven message broker fully supporting the AMQP 1.0[51] specification , which is the main communication protocol.

6.1 MESSAGE PROTOCOL (AMQP 1.0)

There are several message protocols supported by the message brokers. We examined two of the most common message protocols in event driven architectures, the Java Message Service (JMS) protocol, given the fact that it was already adopted by several components in the platform, such as the EMS (Section 2.2), and the Advanced Message Queuing Protocol (AMQP) protocol, and specifically version 1.0.

We opted for AMQP 1.0 given its interoperability across different platforms and languages. JMS is a set of interfaces and associated semantics defined specifically in Java, which can limit interoperability to Java or Java Virtual Machine (JVM)-based languages, unless additional bridging software is used. AMQP's protocol-level standardization makes it inherently more interoperable between different systems and languages. As part of our task, we needed to allow components in a multitude of programming languages to adopt the asynchronous messaging paradigm through a unified protocol.

Furthermore, and an important aspect of opting for AMQP is that it offers support for both brokered and brokerless messaging architectures providing flexibility in deploying distributed systems and can offer scalability advantages in certain scenarios. For example when a centralized broker is not required and producers and consumers can communicated directly between each other, whilst keeping the same implementation API

This allows us to future proof our communication component, and allow for a version of the platform which does not rely on a message broker.

Finally, AMQP in its core implementation offers messaging functionality which would require 3rd party plugins or further configuration in the case of the JMS protocol, such as message annotations, filtering, and settlement and delivery tracking at the protocol level.

6.2 ADDRESS & PAYLOAD SPECIFICATION

As part of the main goal of this task, which was to allow components to seamlessly integrate and communicate between each other, we defined a topic naming convention and payload specification wrapping of a core AMQP message, which set the basis upon which the functionality would be

[46] https://www.rabbitmq.com/

[47] https://kafka.apache.org/

[48] https://activemq.apache.org/

[49] https://aws.amazon.com/sqs/

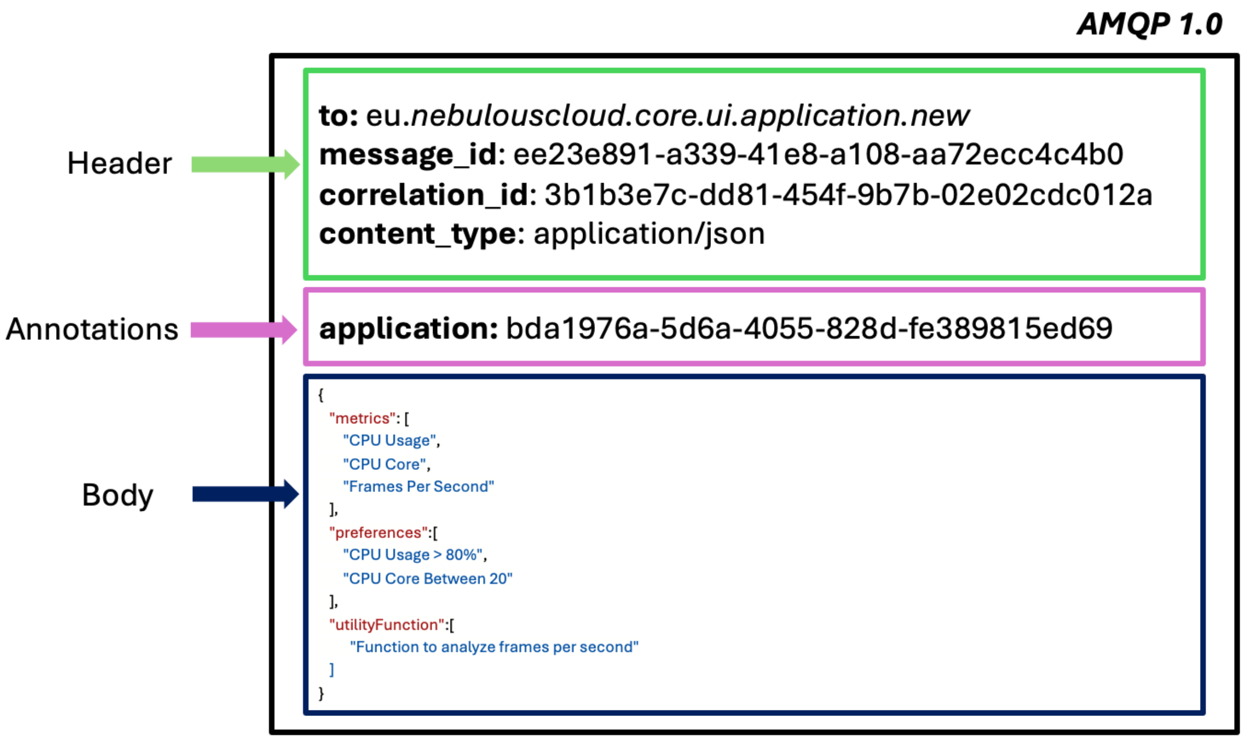
[50] https://activemq.apache.org/components/classic/

[51] https://www.amqp.org/specification/1.0

implemented. The scope of such specification was to ensure future proof extensibility of the platform, by allowing other components to be integrated in the future in a known matter.

6.2.1 Anatomy of a message

The message will adhere to the AMQP 1.0 protocol, whereby through convention components in the platform need to abide by in order to seamlessly integrate with each other.



```
to: eu.nebulouscloud.core.ui.application.new
message_id: ee23e891-a339-41e8-a108-aa72ecc4c4b0
correlation_id: 3b1b3e7c-dd81-454f-9b7b-02e02cdc012a
content_type: application/json

application: bda1976a-5d6a-4055-828d-fe389815ed69

{
  "metrics": [
    "CPU Usage",
    "CPU Core",
    "Frames Per Second"
  ],
  "preferences":[
    "CPU Usage > 80%",
    "CPU Core Between 20"
  ],
  "utilityFunction":[
     "Function to analyze frames per second"
  ]
}
```

Figure 16: the AMQP 1.0 Message

The following table describes each part of the AMQP 1.0 Message.

Table 13. AMQP 1.0 Message Explained

to	This is the intended address of the message, commonly referred to as topic, however in AMQP it is referred to as the address, which can be either a topic destination or a queue. By default, the address is considered a queue unless otherwise specified
message_id	In order to better track the message across logs or even in the actual broker UI each message has a unique identifiable UUID v4 ID.
correlation_id	In order to support the request/reply paradigm of the Enterprise Integration Pattern, the correlation-id is propagated by the consumer of a message in case of a reply, so that the initial request can be matched
content_type	By definition all message are of type JSON.

application	This is set in the message annotations, and it allows component to further filter messages intended for a specific application. The application UUID is generated upon application create by the UI Controller
body	This can be set to any content that is of the correct content-type, in this case JSON (application/json)

6.2.2 Address naming convention

Besides the message payload convention, it is important to determine an address naming convention, which provides several advantages during integration:

- Allows consumers to subscribe to all the nebulous events regardless of component (producer) by subscribing to base of the naming convention e.g., eu.nebulouscloud.*
- Subscribe to all the events coming from a specific component regardless of the scope of the message. For example, if the component is called ui subscribing to eu.nebulouscloud.ui.*
- Subscribe to a predefined action which is used across all components, for example component state, by subscribing to eu.nebulouscloud.*.state.*

In order to achieve this, there are several parts to this convention.

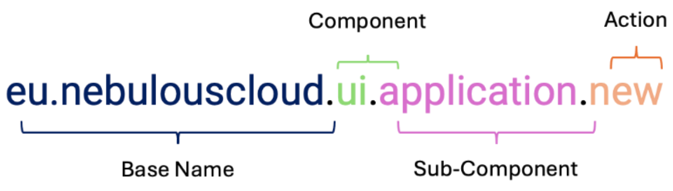


Figure 17: Events Naming Convention

Base Name

Channel names should always start with **eu.nebulouscloud** in order to differentiate that the message traveling through message broker is internal and part of NebulOuS. This will allow the possible use of a message broker in the future, which is not provided by NebulOuS, but provided on-premises by a possible user of our system.

Whether a queue or a topic is required as the channel type will not be depicted on the channel name.

Component

Defines the component producing the message, for example the *ui*, *ems* or *optimizer*. The component does not necessarily need to map on a one-to-one basis with the existing components of the system.

So, the channel name of a message being generated from the ui will be prefixed as:

eu.nebulouscloud.**ui**

However, if a component wishes to further differentiate internal structure, such as *UI Application*, that can be done using as such:

eu.nebulouscloud.**ui**.**application**

This allows expanding the channel name filtering pattern either eu.nebulouscloud.ui* or eu.nebulouscloud.ui.application.* for a specific one.

Action

Each message communicates an action, the action could be of an informative scope like new application, or of a demanding scope, such as request. The scope of the message is open to definition by the components communicating in the platform and can be further customized with their conventions.

On top of the message and naming conventions, we also specified default actions which need to be supported by each component.

Component Lifecycle

In an asynchronous architecture the availability of a component may be optional, regarding its availability, or its bootstrapping phase.

In the case of two coupled components, one needs to be aware of the availability of the other. It is for this reason that we are introducing a reserved word state.

Producers are responsible of sending the appropriate messages which correlate with its application **state**.

Table 14. Component State Message

eu.nebulouscloud.{component}.**state**	It should be produced by a component, during the lifecycle
Available States	starting, ready, stopping, stopped
Custom States	A component may send its own custom state, e.g., "forecasting" which needs to be contractually agreed between components exchanging this state.

There is one more state which will allow the system to know the availability of a component during runtime, which does not deal with the lifecycle of the component, called ping.

Table 15. Component Health Message

eu.nebulouscloud.{component}.health	This should be produced by the component at regular intervals. The internal timestamp can be defined at a later stage.

Funded by the European Union

6.3 EXN MESSAGING LIBRARIES

In order to support components integrating with the library, we created a set of libraries across Java and Python to offer a simplified API on top of Apache Qpid[52], abstracting the complexity of direct AMQP protocol manipulation.

The core aim was to facilitate the adherence to the predefined message and payload specifications. The libraries include functionality for error handling, and communication patterns, effectively reducing the initial implementation effort required by developers when working directly with Apache Qpid for common messaging tasks.

A configuration helper module was included to ease the further customization of the libraries so that these could be used outside the scope of NebulOuS. Simplifying the setup of connections, exchanges, and message filtering according to predefined address and payload conventions. Flexibility was a key consideration, and while the library offered sensible defaults and simplifications for common tasks, it also allowed users to override specific behaviours to suit their unique requirements. This flexibility included customizing threading behaviour, event handling, and message processing logic to accommodate specific application needs.

Implementations for common Enterprise Integration Patterns (EIP), especially the request-reply pattern, were incorporated alongside configurable message filtering based on headers and application UUIDs. This allowed developers to leverage advanced messaging functionalities with minimal effort. Additionally, ready-made publishers for frequent messaging scenarios such as health checks, state updates, and synchronized publishing were made available, ensuring that users could quickly integrate standard messaging functionalities into their applications.

Community involvement and feedback were embraced by open-sourcing the libraries and establishing clear contribution guidelines, greatly enhancing the libraries' quality and adaptability. Continuous integration and deployment pipelines were set up, ensuring that the libraries were reliably tested and updated, maintaining their usability and relevance in the face of evolving messaging needs and practices.

6.3.1 Technical Implementation

There are *five key classes* of the EXN Libraries which provide the core functionality.

Context

The context class is an atomic singleton which is shared across consumers and producers, and it handles state sharing as well as utilities to create and match addresses.

Furthermore, through the context, the state of consumers and producers can be manipulated at runtime.

Consumer

This defines a consumer, using a key and address.

Other properties determine whether the consumers register on a topic or queue.

[52] https://qpid.apache.org/

A consumer can adhere to the address naming conventions, or define an arbitrary queue name, by manipulating the FQDN property.

Finally, a consumer can determine at instantiation whether to filter for specific applications.

Publisher

This class defines the producers in the system. The publisher like the consumer is created by specifying a key and an address. Through properties it can determine whether it publishes on a topic or a queue.

Unlike the consumer, which determines the application filter at instantiation, a producer can publish events for multiple applications.

There are three extensions of the Publisher class available through the library:

StatePublisher – provides methods for publishing the application state as defined in the specification.

ScheduledPublisher – provides methods to continuously publish an event at a selected interval.

SyncPublisher – This publisher abstracts the request-reply EiP using the correlation-id property of the message header, and provides a sendSync method, that allows for synchronous message exchanges.

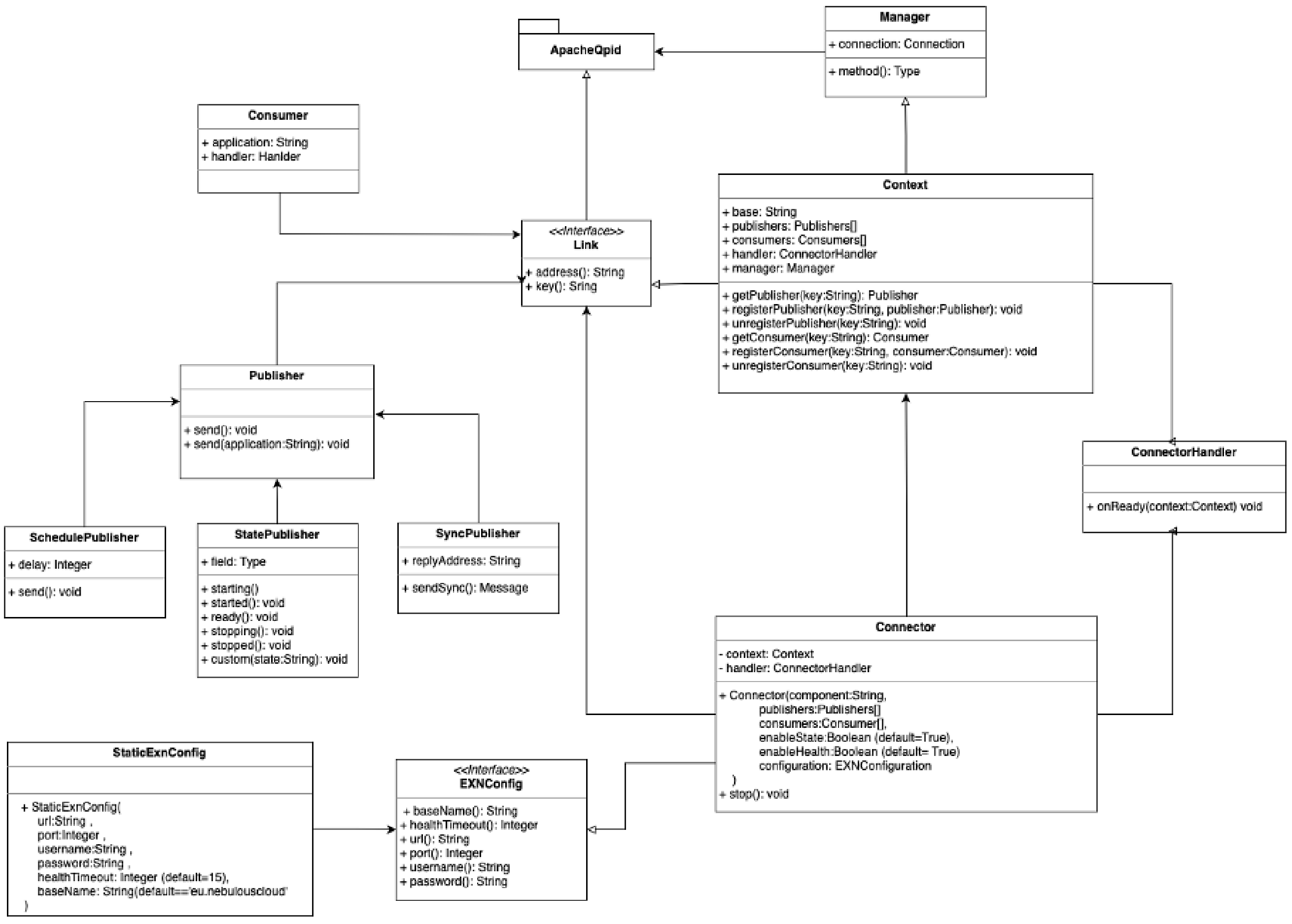


Figure 18: Key classes of the EXN Libraries

ConnectionHandler

This class should be abstracted and defined by all users of the library, it provides the main entry point of the application, once all consumers and publishers are set up.

Connector

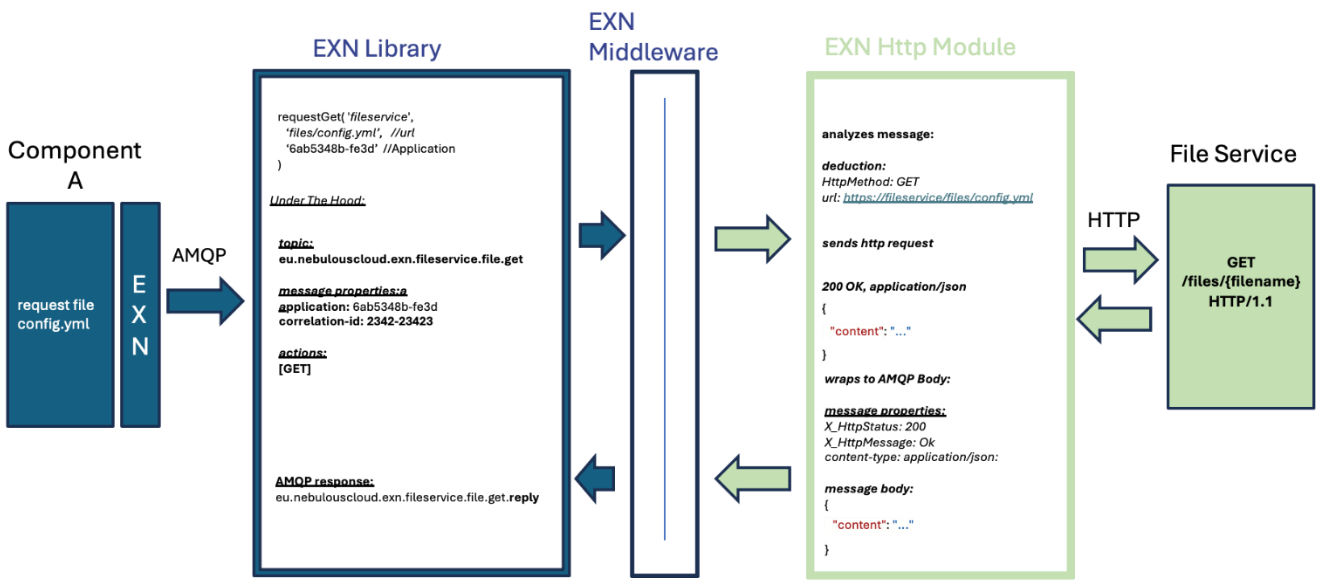
The connector is the main definition class, where all consumers and producers are defined, as well as the message broker connection properties.

It provides the start and stop methods.

6.4 EXN MIDDLEWARE

The EXN Middleware is used as a critical architectural element within the communication framework, providing a service integration middleware which provides communication across varied systems and protocols. Using the EXN Libraries, this component ensures dependable asynchronous communication the AMQP protocol.

A core function of the EXN Middleware is to allow the communication between components are readily available, but do not adhere to the asynchronous event driven architecture, for example all functionality is provided through synchronous HTTP calls. Furthermore, message payload translation, error handling, are amongst the features that need to be handled by the middleware.



```
requestGet( 'fileservice',
  'files/config.yml',  //url
  '6ab5348b-fe3d'  //Application
)

Under The Hood:

topic:
eu.nebulouscloud.exn.fileservice.file.get

message properties:a
application: 6ab5348b-fe3d
correlation-id: 2342-23423

actions:
[GET]

AMQP response:
eu.nebulouscloud.exn.fileservice.file.get.reply
```

```
analyzes message:

deduction:
HttpMethod: GET
url: https://fileservice/files/config.yml

sends http request

200 OK, application/json
{
  "content": "..."
}
wraps to AMQP Body:

message properties:
X_HttpStatus: 200
X_HttpMessage: Ok
content-type: application/json:

message body:
{
  "content": "..."
}
```

Figure 19: EXN Middlware's HTTP abstraction service

In the EXN Middlware we created a core HTTP abstraction service and integrated for example two core services SAL and the Cloud Fog Service Broker (CFSB). The HTTP abstraction service builds on top of the address naming conventions providing the following specification:

- The URL endpoints after the base URL of the HTTP interface, are converted to dot notation
- HTTP Method names are appended to the end of the topic name following the mapping in Table 16.
- Replies are sent using the calling address name and appending **.reply**

This request-reply paradigm, was achieved by using the standard AMQP header property **correlation-id**. Components using the EXN Libraries are able to call methods on any endpoint in a pseudo-sync manner.

Table 16. Topic names and mapping to HTTP methods

HTTP Method	Address Action
GET	nebulouscloud.eu.exn.[service].[action] nebulouscloud.eu.exn.[service].[action]**.reply**
POST	nebulouscloud.eu.exn.[service].[action]**.post** nebulouscloud.eu.exn.[service].[action]**.post.reply**
PUT	nebulouscloud.eu.exn.[service].[action]**.put** nebulouscloud.eu.exn.[service].[action]**.put.reply**
DELETE	nebulouscloud.eu.exn.[service].[action]**.delete** nebulouscloud.eu.exn.[service].[action]**.delete.reply**

6.5 NEXT STEPS & FUTURE WORK

As a next step of task T5.5 is to provide the EXN Libraries in other programming languages such as NodeJS and GO. Whilst having validated the robustness of the address naming conventions and message payload specification.

Further developing both the Java and Python libraries, to include fixes and/or functionality extensions that may come up in the next development cycle of the platform.

Finally, extend the EXN Middleware to provide extended services, enhancing the existing HTTP transformation protocol, and allow for the integration of other message payloads such as XML or EDIs through other communication protocols such as SOAP, JMS, etc.

Funded by the European Union

7 CONCLUSIONS

In this deliverable, we have provided a detailed report on the research and development efforts undertaken as part of NebulOuS Work Package 5. We focused on the Meta-OS features that enable self-adaptive and proactive reconfigurations enactment, considering efficient and resilient monitoring, anomaly detection and predicted severity of imminent SLO violations of applications deployed in cloud computing continuums.

The innovative technology discussed in this deliverable, is designed to enhance autonomous application reconfiguration enactment capabilities within complex and dynamic cloud computing environments, encompassing multi-cloud, fog, and edge resources. Key highlights include the development of a sophisticated, resilient Event Management System that leverages distributed complex event processing to provide critical QoS monitoring. Additionally, we considered AI-driven anomaly detection featured, employing a hybrid approach that combines immunological algorithms with machine learning techniques to address network security challenges effectively. Furthermore, the development of an interoperable IoT/Fog data management system further enhances the ability to manage and propagate application data across a distributed cloud continuum, facilitating the orchestration of IoT data processing pipelines for improved data stream management. The exploration of autonomous application adjustments through mechanisms like the SLO Violation Detector represents a pivotal step towards achieving seamless application reconfigurations and optimizations across the cloud continuum. Lastly, the establishment of an asynchronous message-based API ensures effective communication and interoperability among the various elements of the NebulOuS Meta-OS system.

As already mentioned, this work reports on the first iteration of all these mechanisms that are to be tested and evaluated as part of the NebulOuS pilot demonstrators, but also through the financial support for third parties' program that aims to attract additional real use cases that will test the platform in all meaningful scenarios. The second iteration of all the WP5 mechanism will focus on further enhancing NebulOuS reconfiguration capabilities by incorporating forecasted monitoring metrics, allowing the proactive reconfiguration of applications deployed on cloud computing continuum.

REFERENCES

[1] O. Etzion, P. Niblett, and D. C. Luckham, Event processing in action. Manning Greenwich, 2011

[2] W. A. Higashino, "Complex event processing as a service in multi-cloud environments," 2016

[3] M. Hirzel, "Partition and compose: parallel complex event processing," in Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems. ACM, 2012, pp. 191–200

[4] V. Stefanidis, Y. Verginadis, I. Patiniotakis, G. Mentzas. Distributed Complex Event Processing in Multiclouds. In Proceedings of the 7th European Conference on Service-Oriented and Cloud Computing, 12 - 14 September, 2018, Como, Italy.

[5] S. Veloudis, E. Barmpas, I. Paraskakis, Y. Verginadis, A. Tsagkaropoulos, I. Patiniotakis, G. Horn, M. Różańska, A. Sarros. D2.2 - Initial Semantic Models and Resource Discovery Mechanism. NebulOuS Deliverable 2023.

[6] A. P. Achilleos et al., 'The cloud application modelling and execution language', J. Cloud Comput., vol. 8, no. 1, p. 20, Dec. 2019, doi: 10.1186/s13677-019-0138-7.

[7] Y. Verginadis, I. Patiniotakis, A. Tsagkaropoulos, J.-D. Totow, A. Raikos, G. Mentzas. D2.1-Design of a self-healing federated event processing management system at the edge. Morphemic deliverable 2021.

[8] J. Greensmith, «The Dendritic Cell Algorithm,» Thesis Submitted for the Degree of Doctor of Philosophy, University of Nottingham., Nottingham, 2007.

[9] J. Greensmith y U. Aickelin, «The Deterministic Dendritic Cell Algorithm,» de International Conference on Artificial Immune Systems, Lecture Notes in Computer Science , Phuket, Thailand, 2008.

[10] S. a. P. A. a. A. L. a. C. R. Forrest, «Self-nonself discrimination in a computer,» de Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy, 1994.

[11] M. Tavallaee, E. Bagheri, W. Lu y A. Ghorbani, «A Detailed Analysis of the KDD CUP 99 Data Set,» de Submitted to Second IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), 2009.

[12] G. Horn, M. Różańska, R. Schlatte, Y. Verginadis, D. Apostolou, G. Koronakos, F. Paraskevopoulos. D3.1 - Initial NebulOuS Brokerage and Resource Management. NebulOuS deliverable 2024.

[13] A. Tsagkaropoulos, Y. Verginadis, N. Papageorgiou, F. Paraskevopoulos, D. Apostolou, and G. Mentzas, "Severity: a QoS-aware approach to cloud application elasticity," J Cloud Comp, vol. 10, no. 1, p. 45, Dec. 2021, doi: 10.1186/s13677-021-00255-5.

[14] Y. Verginadis, I. Patiniotakis, F. Paraskevopoulos, A. Tsagkaropoulos, D. Tzormpaki, V. Stefanidis, J.-D. Totow, P. Skrzypek, A. Warno, M. Riedl, M. Rozanska, I. Bizid. D2.2- Implementation of a holistic application monitoring system with QoS prediction capabilities. Morphemic deliverable 2022

[15] L. Lazzari και K. Farias, 'Uncovering the Hidden Potential of Event-Driven Architecture: A Research Agenda'. arXiv 2023. doi: 10.48550/arXiv.2308.05270.

[16] G. Hohpe, B. Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., 2003, ISBN: 978-0321200686

CONSORTIUM

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.