



A Meta Operating System
for Brokering Hyper-Distributed Applications
on Cloud Computing Continuums

Initial Semantic Models and Resource Discovery Mechanism

Project Acronym/Title	NebulOuS: A Meta Operating System for Brokering Hyper-Distributed Applications on Cloud Computing Continuums
Grant Agreement No.:	101070516
Call	World leading data and computing technologies (HORIZON-CL4-2021-DATA-01)
Project duration	36 months 1 September 2022 - 31 August 2025
Deliverable title	Initial Semantic Models and Resource Discovery Mechanism
Deliverable reference	D2.2
Version	1.2
WP	2
Delivery Date	24/10/2023
Dissemination level	PU
Deliverable lead	SEERC
Authors	Simeon Veloudis, Evangelos Barmpas, Iraklis Paraskakis, Yiannis Verginadis, Andreas Tsagkaropoulos, Ioannis Patiniotakis, Geir Horn, Marta Różańska, Alexandros Sarros
Reviewers	Radosław Piliszek (7bull.com), Dimitris Kardaras (ICCS)
Abstract	This document discusses NebulOuS' approach to resource discovery. It presents the models underpinning NebulOuS' resource discovery mechanism, including: a declarative model of expressing hyper-distributed components and their deployment characteristics; a declarative model for capturing QoS requirements attached to application components; a semantic representation of the capabilities and characteristics of a pool of CC resources; a model for semantically representing QoS requirements; and finally, a model for determining optimal application component placement based on the current available resource capabilities and characteristics, the component's QoS requirements, as well as any user-expressed preferences regarding deployment. The document also presents an initial description of a prototype of NebulOuS resource discovery mechanism.
Keywords	Fog computing, Cloud Continuum, application deployment, resource discovery, declarative description, semantic models, brokerage, QoS, quality assurance
Dissemination Level	
PU	Public, fully open
Type	
R	Document, report (excluding the periodic and final reports)

Version	Date	Owner	Author(s)	Changes to previous version
0.1	2023-07-13	SEERC	Simeon Veloudis	Initial table of contents
0.2	2023-07-15	UBI	Christos Alexandros Sarros	Addition of KubeVela to the table of contents
0.2	2023-07-15	ICCS	Ioannis Patiniotakis	Addition of the YAML metric model section to the table of contents
0.2	2023-07-19	ICCS	Andreas Tsagaropoulos	Addition of the TOSCA/CAMEL section to the table of contents
0.3	2023-08-25	SEERC	Simeon Veloudis	Initial writing about CoCoOn
0.3	2023-08-27	SEERC	Evangelos Barmpas	Initial writing about OWL-Q and Q-SLA
0.4	2023-09-8	SEERC	Simeon Veloudis	Completed content writing about CoCoOn
0.4	2023-09-10	SEERC	Evangelos Barmpas	Completed content writing about OWL-Q and Q-SLA
0.5	2023-09-12	SEERC	Simeon Veloudis	Editing of the OWL-Q and Q-SLA sections
0.6	2023-09-13	UBI	Christos Alexandros Sarros	Completed content on KubeVela
0.6	2023-09-13	ICCS	Ioannis Patiniotakis	Completed content on the YAML metric model
0.6	2023-09-13	ICCS	Andreas Tsagaropoulos	Completed the TOSCA/CAMEL section to the table of contents
0.7	2023-09-15	SEERC	Evangelos Barmpas	Formatting, addition of references and enumeration of tables, figures, and listings.
0.8	2023-09-21	ICCS	Yiannis Verginadis	Addition of the Metadata Schema discussion
0.9	2023-09-24	UiO	Geir Horn	Addition of content on optimisation
1.0	2023-10-06	ICCS	Yiannis Verginadis	Added section on resource discovery mechanism
1.1	2023-10-13	SEERC	Evangelos Barmpas	Final formatting and fixing for various visual issues
1.2	2023-10-24	EUT	María Navarro	Final Review

Table of Contents

Table of Contents	4
Figures	6
Listings	7
List of Acronyms.....	8
Executive Summary	9
1. Introduction	10
1.1 Model-Driven Resource Discovery	11
1.2 Semantic Technologies for Optimisation and Quality Assurance	11
1.3 NebulOuS Resource Discovery	12
2. Describing Application Placement in the Cloud Continuum	14
2.1 Existing Work	14
2.2 Application Definition and Deployment in NebulOuS.....	15
2.2.1 Open Application Model introduction	15
2.2.2 KubeVela Model details.....	16
3. Metric Model	22
3.1 Metrics	22
3.1.1 Windows	23
3.1.2 Sensors.....	24
3.1.3 Output.....	24
3.1.4 References	25
3.2 Requirements	25
3.3 Metric Model structure	26
3.4 Language and Style.....	27
4. Optimisation DSL.....	29
4.1 Parameterised application topology model	29
4.2 Optimisation model.....	32
4.3 NebulOuS Integration	35
5. Resource Discovery Mechanism	36
5.1 Registering Fog/Edge devices to NebulOuS.....	36
5.2 Unregistering Fog/Edge devices to NebulOuS.....	38
5.3 Resource Discovery	39
6. Semantic Modelling	43
6.1 Asset Modelling.....	43
6.1.1 IoT Ontologies.....	43
6.1.2 IaaS Ontologies.....	44
6.1.3 CoCoOn	45

6.1.4	The NebulOuS Approach	50
6.2	QoS Requirements.....	51
6.2.1	Service Quality Meta-Models	51
6.2.2	OWL-Q	52
6.2.3	Q-SLA.....	56
6.2.4	Metadata Schema.....	60
7.	Conclusions	63
	References	64

Figures

Figure 1: Overview of the NebulOuS approach	13
Figure 2: OAM architecture	16
Figure 3: Application main abstractions	17
Figure 4: Application modelling abstractions in KubeVela	21
Figure 5: Overview of NebulOuS Resource Discovery mechanism	36
Figure 6: Sequence diagram for registering Fog/Edge devices	37
Figure 7: Sequence diagram for unregistering Fog/Edge devices	39
Figure 8: Resource Discovery mechanism dashboard	39
Figure 9: List of open user's registration requests (currently empty)	40
Figure 10: New device registration request form	40
Figure 11: List of open user's registration requests (with a new request)	40
Figure 12: List of open user's registration requests – Collecting device capabilities	41
Figure 13: Device capabilities as collected by Resource Manager and stored in database – Request has been updated....	41
Figure 14: List of open user's registration requests (Request awaits authorization, after capabilities collection)	42
Figure 15: Admin view of open registration requests awaiting authorization	42
Figure 16: List of open user's registration requests (device is being onboarded)	42
Figure 17: List of open user's registration requests (successful device onboarding)	42
Figure 18: CoCoOn v1.0.1	46
Figure 19: OWL-Q facets	52
Figure 20: OWL-Q specification facet	53
Figure 21: OWL-Q Attribute facet	54
Figure 22: OWL-Q Metric facet	55
Figure 23: OWL-Q Unit facet	56
Figure 24: OWL-Q Value Type facet	56
Figure 25: OWL-Q Q-SLA facet	57
Figure 26: Metadata Schema overview	61
Figure 27: The UML class diagram for the Processing domain	62

Listings

Listing 1: video surveillance application modelling and deployment using the Open Application Model and KubeVela	20
Listing 2: Example of metrics	23
Listing 3: Example of grouping processing	24
Listing 4: Example of output specification	25
Listing 5: Example of references	25
Listing 6: SLO requirement specification	25
Listing 7: Example of scopes	26
Listing 8: Metric model example (reduced)	27
Listing 9: Examples of specification styles in detailed format.....	28
Listing 10: The resource requirements represented as decision variables with ranges of possible values	29
Listing 11: The parameterised placement instructions as index variables with value ranges.....	30
Listing 12: A KubeVela defined component for facial detection with resource requirements in red boxes and component placement instructions in the green box.....	31
Listing 13: Representing the application deployment requirements.....	33
Listing 14: The total multiplicity of the various worker types broken down in workers per location	33
Listing 15: Deployment node identifiers and number of worker instances per node	33
Listing 16: The cost constraints of the optimisation problem.....	34
Listing 17: The utility calculations for the facial detection component workers problem	35
Listing 18: Example compute service specification	47
Listing 19: Example storage service specification	47
Listing 20: Example load balancing service specification.....	48
Listing 21: Example pricing specification	49
Listing 22: Example storage pricing specification.....	49
Listing 23: Downlink speed specification	50
Listing 24: Example of an attribute and its metrics.....	59
Listing 25: Example of a simple SLA	59
Listing 26: Example of a SL, its pricing model, and SL transition	59
Listing 27: Example of an SLO, its penalty, and compensation.....	60

List of Acronyms

AMPL	A Mathematical Programming Language
API	Application Programming Interface
ARM	Architectural Reference Model
CAMEL	Cloud Application Modelling and Execution Language
CC	Cloud Continuum
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
DB	Database
DNS	Domain Name Service
DSL	Domain-specific Language
EaaS	Edge-as-a-Service
EDMM	Essential Deployment Metamodel
EMS	Event Management System
EPL	Event Processing Language
GUI	Graphical User Interface
IOPS	Input/Output Operations per Second.
IP	Internet Protocol
JSON	JavaScript Object Notation
NGSI-LD	Next Generation Service Interfaces Linked Data
MDS	Metadata Schema
MQ	Message Queues
OAM	Open Application Model
OS	Operating System
OWL	Web Ontology Language
QUDT	Quantities, Units, Dimensions, and Types
RAM	Random Access Memory
RDF	Resource Description Framework
SAREF	Smart Application Reference
SL	Service Level
SLA	Service-level Agreement
SLO	Service-level Objective
SOSA	Sensor, Observation, Sample, and Actuator
SQL	Structured Query Language
SSN	Semantic Sensor Networks
IoTMA	Internet of Things Model and Analytics
TOSCA	Topology and Orchestration Specification for Cloud Applications
UX	User Experience
VM	Virtual Machine
W3C	World-Wide Web Consortium
WP	Work Package
XML	Extensible Markup Language
XSD	XML Schema Definition
YAML	YAML Ain't Markup Language

Executive Summary

NebulOuS aims at designing and implementing a novel meta-Operating System (OS) that will enable secure and optimal deployment of hyper-distributed applications across the Cloud continuum. It aspires to provide a *resource management platform* that will transparently create and maintain an adaptive environment for hosting hyper-distributed application components (in the form of containerised workloads), whilst satisfying any QoS requirements, as well as any security and privacy constraints, attached to them. The platform will provide facilities for resource discovery, allocation, and provisioning, as well as for dynamic application component placement and scheduling.

This document focuses on resource description with the aim of resource discovery i.e., on the identification of suitable CC resources for deploying hyper-distributed application components. In NebulOuS this is done by matching the QoS requirements attached to the components against the capabilities and characteristics of a pool of available CC resources. NebulOuS embraces a model-driven and user-centric approach to resource discovery that enables users to declaratively define:

- Application compositions and deployments (i.e., hyper-distributed apps).
- QoS requirements attached to application components.
- Deployment preferences.

NebulOuS assures the quality of the resource discovery process, hence the quality of the provided CC brokerage, by ensuring the quality of the QoS requirements used in this process. To this end, it relies on a semantic model that describes the declaratively-defined application component QoS requirements. By describing QoS requirements ontologically, we pave the way for a quality assurance mechanism that relies on *semantic reasoning* for assessing the correctness of these requirements by comparing them against a set of semantically-captured application consumption policies. These are policies that operate at a higher level of abstraction and express a broader set of business and security requirements that characterise an application component (as opposed to an application component workload or instance).

As its title suggests, this document provides an initial account of the semantic and declarative models underpinning the NebulOuS discovery mechanism. More specifically, the following models are proposed:

- The Open Application Model as the de-facto standard for describing hyper-distributed applications, and the KubeVela software as a tool for application composition and deployment. KubeVela has been chosen due mainly to its out of the box support for Kubernetes.
- A custom model (based on the metric model from CAMEL [1]) for capturing the QoS requirements associated with hyper-distributed applications and for addressing their monitoring aspects.
- A model based on AMPL for describing the constraints and the objectives according to which application components are managed throughout their lifecycles.

These models are underpinned by ontological descriptions of CC resource capabilities and characteristics such as storage capacities, network connectivity, geolocation, and price. The final version of these models, and of the discovery mechanism, will be reported in D2.3 (due in by M34).

1. Introduction

Accessing remote computing resources in data centres provisioned by cloud providers is today the de facto model for Internet-based applications [2]. According to this model, data generated by a wide range of devices, spanning from smartphones and wearables to smart city cameras and factory sensors, are transferred to typically geographically distant clouds for processing and storage. Nevertheless, this computing model is swiftly becoming unviable [2]. Firstly, it precludes applications with hard real-time requirements that cannot tolerate the high communication latencies incurred by long-distance data transfers. Secondly, as the number of interconnected devices continues to increase at astonishing rates¹, communication latencies also increase, degrading the Quality of Service (QoS) even for applications with relaxed real-time requirements [3], [4].

An alternative computing model that can alleviate these problems advocates the decentralisation of a portion of the pooled resources available in cloud data centres and their distribution across the Cloud Continuum (CC) i.e., towards the edge of the network and closer to end users, actuators, and data sources (sensors) [5]. These resources, henceforth referred to CC *resources*, typically take the form of dedicated micro-data centres, or of Internet nodes such as routers, gateways, and switches augmented with processing capabilities.

Contrary to cloud resources, fog and edge resources are [2]: (i) constrained – they typically have less compute and storage capacity compared to cloud resources; (ii) mutually heterogeneous – they feature different machine architectures and have different capabilities; (iii) highly dynamic – their workloads vary widely. These characteristics render the task of managing CC resources and ensuring their optimal usage particularly complex [2].

Managing CC resources entails several sub-processes including resource discovery, allocation, provisioning, scheduling, and placement [6]. In this deliverable we focus on *resource discovery* i.e., on the identification of CC resources capable of deploying components of hyper-distributed applications with associated and varying QoS requirements, and based on user-expressed preferences.

In the literature, two main approaches to fog resource discovery have been proposed. In [7], the authors propose the Edge-as-a-Service (EaaS) platform that provides, amongst others, a lightweight discovery protocol that operates across homogeneous fog resources. The protocol is based on a master node that executes a manager process on each resource and interacts with it by issuing commands. A major drawback is that the protocol cannot operate in federated fog environments with heterogeneous resources; moreover, the security implications of installing and executing a manager process on remote resources are ignored.

Closer to our work, the authors in [8] propose an algorithm that discovers fog resources by matching the QoS requirements of an application against the capabilities of available fog resources. The protocol relies on a programming infrastructure called Foglets and assumes that fog resources are publicly known or available for use; a join algorithm that selects one resource from among a set of available fog resources that are equidistant from the user is also provided. Nonetheless, the proposed protocol ignores interoperability issues stemming from the inherent heterogeneity of fog resources and the non-standardised naming conventions used for describing the characteristics of these resources and the QoS requirements of their workloads. Moreover, the protocol makes no provisions for assuring the quality of the provided brokerage function.

¹ According to IoT Analytics (<https://iot-analytics.com/number-connected-iot-devices/>), the Internet of Things has increased from around 3 billion interconnected devices in 2015 to more than 16 billion interconnected devices in 2023 generating more than 300 quintillion bytes per day.

1.1 Model-Driven Resource Discovery

Following [8], NebulOuS discovers CC resources by matching their capabilities against the QoS requirements attached to application components. It embraces a *model-driven* and *user-centric* approach that focuses on facilitating the overall process of defining hyper-distributed applications and their workloads and deploying them over CC resources. In particular, it enables users to *declaratively* define:

- **Application component compositions and deployments** based on the Open Application Model (OAM²) as a de-facto standard for modelling cloud-native application deployment, and on the Kubernetes-native KubeVela software³ as the main tool for describing application composition and deployment.
- **QoS requirements attached to application components** through the use of a *custom model* –based on the metric model of CAMEL⁴– that enables the definition of *metrics* over arbitrary *user defined QoS attributes* i.e., any attributes for which a source of values (i.e., a sensor) can be specified.
- **Optimisation goals** i.e., preferences regarding application deployment, through the use of an optimisation model based on AMPL⁵.

1.2 Semantic Technologies for Optimisation and Quality Assurance

NebulOuS relies on semantic technologies for describing CC resources, and for assuring the quality of the resource discovery process, *hence the quality of the provided CC brokerage*. To this end, two distinct –albeit interrelated– ontological models are provided:

1. The *asset model*, for describing common traits encountered in infrastructural CC services, including compute and storage capacities, network connectivity, geolocation, and price.
2. The *application component QoS requirements model* that is populated with the information described in the metric model outlined above.

The asset model provides a basis for determining how an application deployment is to be realised across a pool of available resources given a set of user-expressed preferences, and providing that the QoS requirements attached to the application’s components are satisfiable. More specifically, the asset model forms the basis of NebulOuS’s *optimisation model* that describes the constraints and the objectives according to which application components are managed throughout their lifecycles. This includes *optimised* application component placement that considers the current capacities and capabilities of a pool of available CC resources, the component’s QoS requirements, as well as any user-expressed preferences regarding application consumption.

The application component QoS requirements model provides the basis of NebulOuS’ quality mechanism. More specifically, by ontologically describing QoS requirements, we pave the way for a quality assurance mechanism that relies on *semantic reasoning* for assessing the correctness of these requirements by comparing them against a set of semantically captured *application consumption policies*. These are policies that operate at a higher level of abstraction and express a broader set of business and security requirements that characterise an application component as opposed to an application component workload (instance). For example, a policy may impose minimum limits on the compute capacity that must be assigned to an application; any QoS requirement attached to a component of this application must respect these limits. In other words, we are envisaging a situation whereby QoS requirements potentially vary across different deployed workloads of an application,

²<https://oam.dev/>

³<https://kubvela.io/>

⁴<https://camel-dsl.org/>

⁵<https://ampl.com/>

whilst abiding by an overarching set of application consumption policies⁶. Evidently, our approach transforms the process of assuring the quality of CC brokerage into one of semantic reasoning, bringing about the following advantages: (i) reasoning based on knowledge that is *semantically inferred* and not necessarily available at the syntactic level; (ii) reliance on a standards-based approach that avoids ad-hoc solutions.

1.3 NebulOuS Resource Discovery

Figure 1 provides an overview of the NebulOuS approach to resource discovery. Users define hyper-distributed application components and their deployments using KubeVela; they describe the QoS requirements attached to each component using an adequate for the CC metric model. QoS requirements are also mapped to the application component QoS requirements ontology for interoperability purposes, and for assessing their correctness⁷. Moreover, users specify optimisation goals regarding application deployment. Based on these goals and on the capabilities and characteristics of the available resources (described in the asset model⁸), the AMPL-based optimisation model yields the optimal application deployment.

The rest of this deliverable is structured as follows: Section 2 outlines cloud-application description languages and introduces the Open Application Model and KubeVela. Section 3 describes the NebulOuS custom metric model and Section 4 provides an overview of the AMPL-based optimisation model. Section 5 provides an overview of the NebulOuS resource discovery mechanism. Section 6 outlines semantic modelling in the IoT, overviews ontologies for QoS specification, and presents the two ontological models of NebulOuS: the asset model and the application component QoS model. Finally, Section 7 outlines conclusions.

⁶ Consider, for instance, the following scenario. An organisation develops an IoT application and sets an application consumption policy that imposes minimum limits on the CPU cores and RAM size that must be available to an application execution. The organisation then deploys application instances at different locations to serve the needs of its customers (one deployed instance per customer is assumed). Customers are free to set their own QoS requirements on their application instances if these abide by the overarching application consumption policy.

⁷ Assessing the correctness of QoS requirements is beyond the scope of this deliverable.

⁸ Capabilities and characteristics are dynamically obtained from the NebulOuS monitoring system which is outside the scope of this report.

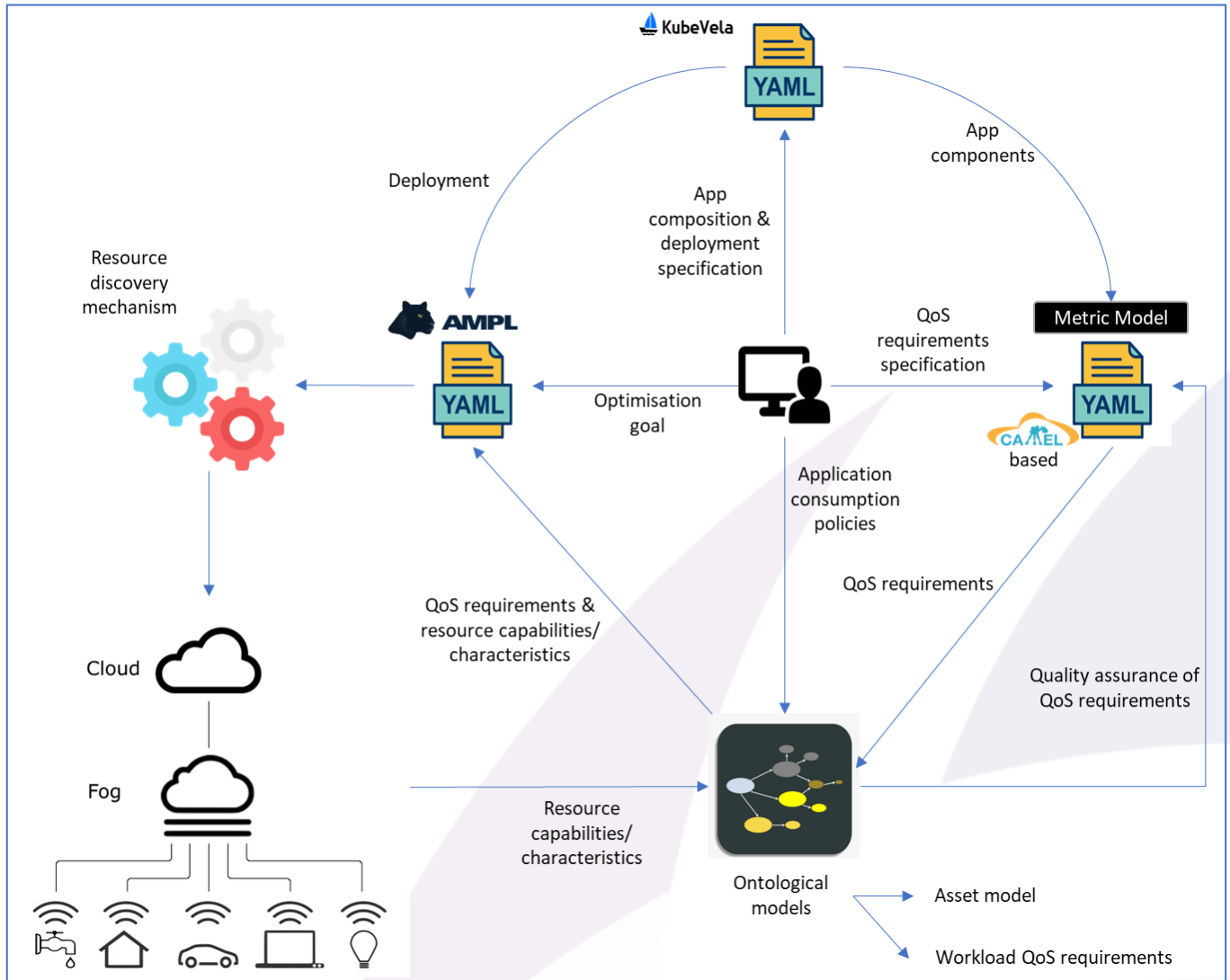


Figure 1: Overview of the NebulOuS approach

2. Describing Application Placement in the Cloud Continuum

2.1 Existing Work

Two prominent domain specific languages (DSLs) for the description of cloud native applications are TOSCA⁹ and CAMEL¹⁰. TOSCA is an open OASIS standard for describing application components, their interrelationships, their topologies, and the processes through which they are managed. It has recently been extended to provide support for edge and serverless deployments [9], [10] but lacks an (official) orchestrator implementation. To fill this gap, several TOSCA-compliant orchestrators have been proposed (see Table 1 for an overview). It can be seen, from the contents of Table 1 that the most actively maintained TOSCA approaches are the Unfurl¹¹ and the Infrastructure Manager¹² orchestrators, although based on the Github stars Cloudify¹³ is the most popular solution. These, however, are either not adequately supported, or target prior versions of the language, or are not open source, impeding their exploitation and/or further development. Moreover, none of these orchestrators provides support for Kubernetes¹¹ – the current de-facto standard for automated deployment and management of containerised applications.

Table 1: TOSCA orchestrators

<i>Solution</i>	<i>Open Source</i>	<i>Public Cloud support (claimed)</i>	<i>Targeted TOSCA</i>	<i>Last commit - last year commits</i> ¹²	<i>GitHub stars</i>
OpenTOSCA ¹³	Yes	Low: only one cloud supported for spawning VMs	1.3	4/7/2023 - 10	52
EDMM ¹⁴	Yes	Excellent: large number of clouds supported for spawning VMs (through translating technologies)	TOSCA light [11]	14/2/2023 - 1	10
Unfurl ¹⁵	Yes	Very good: moderate number of clouds supported for spawning VMs	1.3	23/9/2023 - 803	90
OpenTOSCA Vintner ¹⁶	Yes	Very good: moderate number of clouds supported for spawning VMs (through subordinate orchestrators)	1.3	25/7/2023 - 206	2
xOpera ¹⁷	Yes	Very good: moderate number of clouds supported for spawning VMs	1.3	27/12/2022 - 1	32
Infrastructure Manager ¹⁸	Yes	Excellent: large number of clouds supported for spawning VMs	1.0	21/9/2023 - 450	53
Cloudify ¹⁹	Partially ²⁰	Excellent: large number of clouds supported for spawning VMs	Custom ²¹	18/9/2023 - 319	141

⁹ <https://www.oasis-open.org/committees/tosca>

¹⁰ <https://camel-dsl.org/>

¹¹ <https://kubernetes.io/>

¹² Data as of 28/9/2023

¹³ <https://github.com/OpenTOSCA/container>

¹⁴ <https://github.com/UST-EDMM/edmm>

¹⁵ <https://github.com/onecommons/unfurl>

¹⁶ <https://github.com/opentosca/opentosca-vintner>

¹⁷ <https://github.com/xlab-si/xopera-opera>

¹⁸ <https://github.com/grycap/im>

¹⁹ <https://github.com/cloudify-cosmo/cloudify-manager>

²⁰ Cloudify offers some its capabilities under either a community edition or a paid premium edition. The source code in the provided repository has no documentation on the setup of Cloudify (or the completeness of the provided components), while the license of both editions also dictates that they can be used only in binary form when downloaded in a compiled form.

²¹ <https://github.com/cloudify-cosmo/cloudify-manager-blueprints/blob/master/simple-manager-blueprint.yaml>

CAMEL enables users to specify a wide range of aspects related to multi- and cross-cloud applications, including domains of deployment, requirements, monitoring metrics, scalability, security, organisation, and execution. It supports the `models@runtime` [12] approach so all application updates, including reconfiguration decisions, are reflected and stored in the model. CAMEL also supports complex user preferences specifications including utility functions for expressing the goodness of a deployment configuration from an application owner's perspective. CAMEL exhibits a significantly richer bundle of capabilities than TOSCA, especially in the multi-clouds deployment management domain. Nevertheless, its high degree of verbosity and its lack of support for Kubernetes renders its use in NebulOuS cumbersome. Moreover, the software supporting CAMEL models is limited to the Java-based Eclipse CDO models²² repository and the CDO client, which impedes the use of these models through C++ or Python components.

2.2 Application Definition and Deployment in NebulOuS

NebulOuS embraces a model-driven and user-centric approach to deploying hyper-distributed applications in the Cloud-Edge continuum. It aims at offering a simple yet powerful means for end users to define customised application compositions and deployments. NebulOuS leverages state-of-the-art technical approaches and specifications used by the cloud native computing community for distributed application composition and deployment. By relying on such widely adopted standards and tools, as opposed to ad-hoc solutions, we facilitate the use of the NebulOuS platform and promote its adoptability.

The centrepiece of our approach is the adoption of the Open Application Model (OAM)²³ as the de-facto standard for describing hyper-distributed applications, and the use of the KubeVela²⁴ software as a tool for application composition and deployment. KubeVela is a Cloud Native Computing Foundation²⁵ (CNCF) incubation project, and it is increasingly adopted by the community and industry²⁶. Although its use parallels that of TOSCA and CAMEL, KubeVela does not offer a DSL for implementing OAM, it relies instead on the diffused and lightweight YAML for describing application compositions and deployments. Contrary to CAMEL and TOSCA, KubeVela is an official orchestrator for OAM applications, much more popular than any TOSCA/CAMEL orchestrator (over 5600 Github stars) while also being actively developed (654 commits in 2023 up until 28/9/2023). KubeVela also provides out-of-the-box support for Kubernetes enabling applications using it to take advantage of any cloud provider supporting Kubernetes. The rest of this section outlines the Open Application Model and its KubeVela implementation aiming to shed light on the capabilities it offers to NebulOuS.

2.2.1 Open Application Model introduction

OAM²³ was originally created by Alibaba and Microsoft as a collection of high-level abstractions for modelling cloud-native applications. Although the model is designed to be agnostic to any underlying infrastructure technology, its implementations are focused on Kubernetes. KubeVela²⁴ is one such implementation. Others include the Rudr²⁷ and Crossplane²⁸ projects that were, however, discontinued giving way to KubeVela as the de-facto OAM implementation that currently drives the specification forward.

OAM seeks to address the problem of how to compose distributed applications in the context of (micro)service-oriented architectures. Its main goal is to devise a generic, infrastructure-agnostic way to describe application deployment across hybrid environments; this absolves developers from having to understand low-level infrastructure details, thus allowing them to focus on the architecture and actual development of

²² <https://eclipse.dev/cdo/>

²³ <https://oam.dev/>

²⁴ <https://github.com/kubevela/kubevela>

²⁵ <https://www.cncf.io/>

²⁶ Indicated by the large number of GitHub stars that the KubeVela project received - more than 5400 at the time of writing this report

²⁷ <https://github.com/oam-dev/rudr>

²⁸ <https://github.com/crossplane/oam-kubernetes-runtime>

distributed applications. To this end, OAM separates the application definition from operational details. This separation of concerns is based on a clear distinction between the platform engineer and application developer organisational roles. It is achieved through a higher-level abstraction that decouples the description of distributed applications from the underlying infrastructure details, using a model that is self-contained and allows the definition of an application’s components and operational behaviours. This approach enables clarity and extensibility, enabling re-use of application components. Figure 2 provides an overview of the OAM architecture.

As of September 2023, the latest official OAM specification is v0.3.0 (released in June 2021). Nevertheless, since then a series of KubeVela versions (with the latest one being v1.9.0 released in June 2023) have overwritten and outdated parts of the official specification. Our account of OAM is based on the KubeVela version 1.9.0.

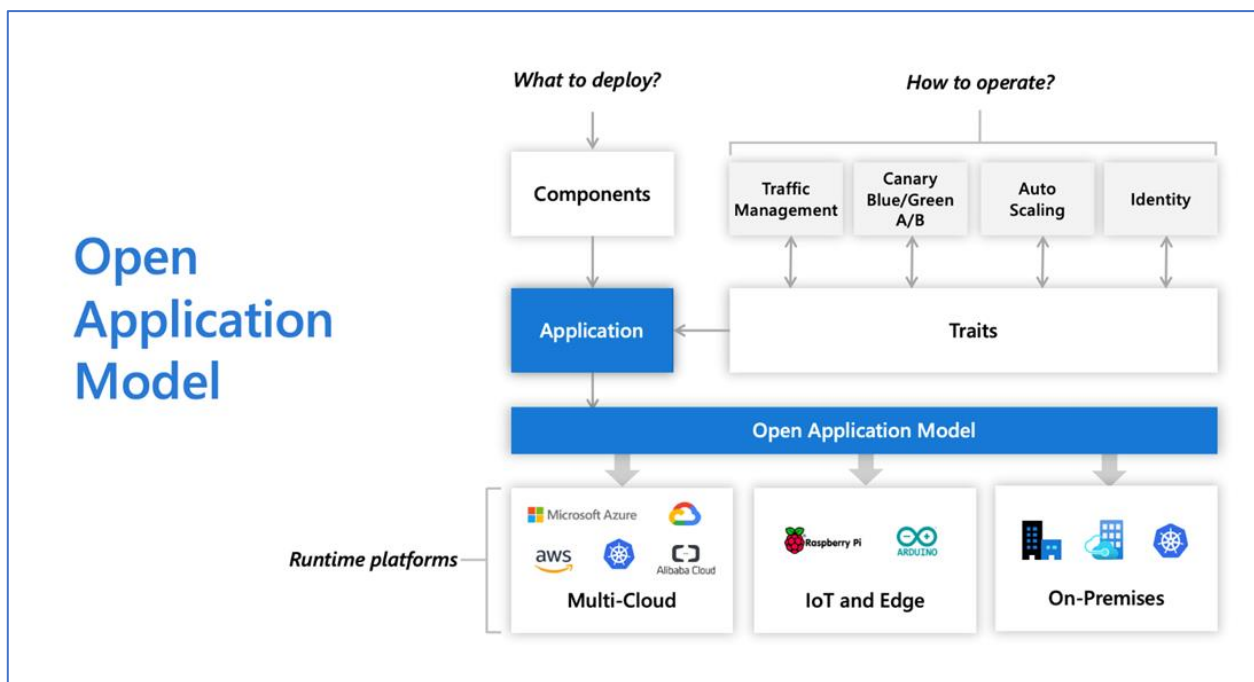


Figure 2: OAM architecture

2.2.2 KubeVela Model details

The main abstraction used in KubeVela, is that of an “Application”. An application is defined as: “a collection of interrelated, but discrete components (services, tasks, workers) that, when coupled with configuration and instantiated in a suitable runtime infrastructure, together accomplish a unified functional purpose.”²⁹ Application deployments are captured through user-defined deployment plans which are in turn defined as Directed Acyclic Graphs (DAGs). KubeVela applications use four main abstractions:

- **Component:** Defines the delivery artefact (binary, Docker image, Helm Chart, etc.), or cloud service included in an application. In KubeVela, an application typically takes the form of a microservice and, as a result, it is recommended that it includes less than 15 components: a core service and its dependencies (e.g., database, cache, pub/sub, etc.).
- **Trait:** A characteristic defined on a single component; for example: scale and rollout strategy, persistent storage claim, gateway endpoint, etc. Traits may be used for expressing user preferences/requirements regarding component placement.
- **Policy:** Defines a strategy for a certain aspect of an application (e.g., multi-cluster topology, configuration overrides, security/firewall rules, etc.). Policies bear some similarity with traits, but they affect the entire application (as opposed to a single component trait).

²⁹ https://github.com/oam-dev/spec/blob/master/2.overview_and_terminology.md

- **Workflow:** Allows to define every step in the delivery process. Some typical steps are manual approval, partial deploy, notification, etc.

Each abstraction introduces a programmable module that can be referenced by an application entity (see Figure 3). Applications are expressed declaratively in YAML.

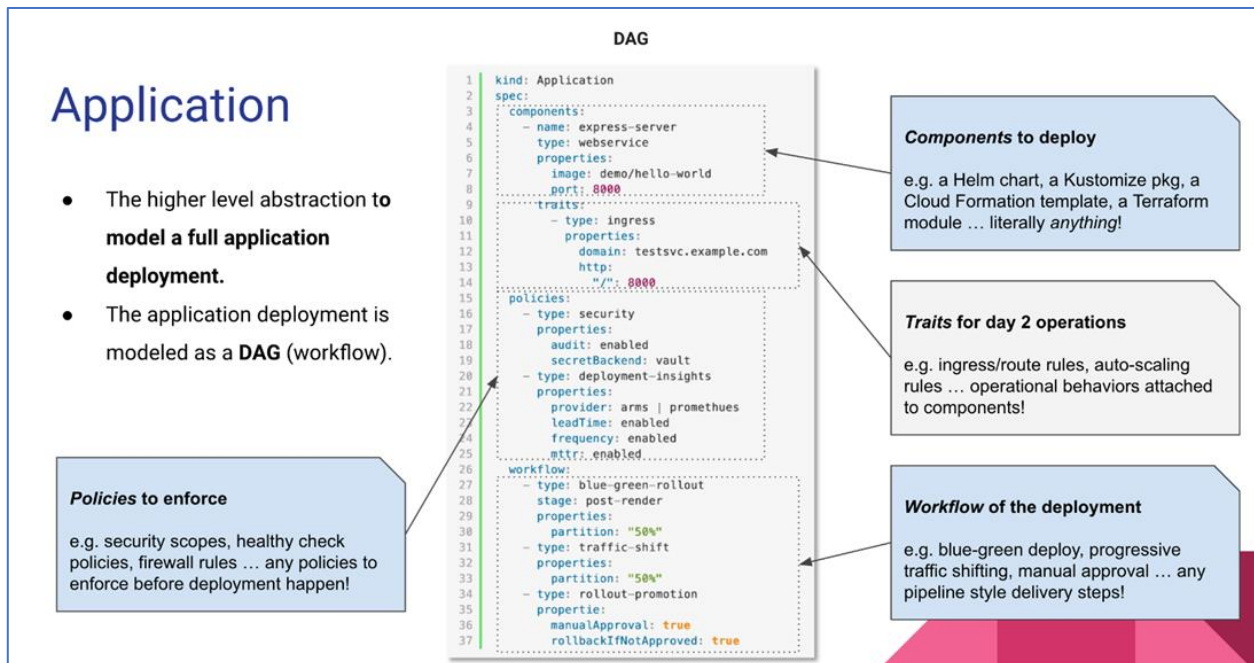


Figure 3: Application main abstractions

For illustration purposes, we provide an example of a simple video surveillance application modelled and deployed using the Open Application Model and KubeVela (see Listing 1). The application is composed of 4 distinct components: *Kafka Server*, *Kafka UI*, *Video Capture* and *Video Player*. All components are modelled as webservices (a specific KubeVela application type³⁰), with their relevant properties and traits attached. As an example of the capabilities offered by KubeVela, consider the affinity and geoLocation traits that are attached to the face-detection component. Using these traits, a NebulOuS user can explicitly denote preferences and/or requirements regarding component placement, based on particular aspects that need to be accommodated by the NebulOuS Meta-OS (e.g., affinity/anti-affinity constraints, geographical requirements, etc.).

³⁰ <https://kubvela-docs.oss-cn-beijing.aliyuncs.com/docs/v1.1/end-user/components/cue/webservice>

```

apiVersion: core.oam.dev/v1beta1
kind: Application
metadata:
  name: surveillance-demo
  namespace: default
spec:
  components:
  - name: kafka-server
    type: webservice
    properties:
      image: confluentinc/cp-kafka:7.2.1
      hostname: kafka-server
      ports:
        - port: 9092
          expose: true
        - port: 9093
          expose: true
        - port: 29092
          expose: true
      cpu: "1"
      memory: "2000Mi"
      cmd: [ "/bin/bash", "/tmp/run_workaround.sh" ]
      env:
        - name: KAFKA_NODE_ID
          value: "1"
        - name: KAFKA_LISTENER_SECURITY_PROTOCOL_MAP
          value: "CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT"
        - name: KAFKA_LISTENERS
          value: "PLAINTEXT://0.0.0.0:9092,PLAINTEXT_HOST://0.0.0.0:29092,CONTROLLER://0.0.0.0:9093"
        - name: KAFKA_ADVERTISED_LISTENERS
          value: "PLAINTEXT://kafka-server:9092,PLAINTEXT_HOST://212.101.173.161:29092"
        - name: KAFKA_CONTROLLER_LISTENER_NAMES
          value: "CONTROLLER"
        - name: KAFKA_CONTROLLER_QUORUM_VOTERS
          value: "1@0.0.0.0:9093"
        - name: KAFKA_PROCESS_ROLES
          value: "broker,controller"
    traits:
      - type: storage
        properties:
          configMap:
            - name: kafka-init
              mountPath: /tmp
              data:
                run_workaround.sh: |-
                  #!/bin/sh
                  sed -i '/KAFKA_ZOOKEEPER_CONNECT/d' /etc/confluent/docker/configure
                  sed -i 's/cub zk-ready/echo ignore zk-ready/' /etc/confluent/docker/ensure
                  echo "kafka-storage format --ignore-formatted -t NqnEdODVKKiLTfJvqd1uqQ== -c
/etc/kafka/kafka.properties" >> /etc/confluent/docker/ensure
                  /etc/confluent/docker/run
          - name: kafka-ui
            type: webservice
            properties:
              image: provectuslabs/kafka-ui:cd9bc43d2e91ef43201494c4424c54347136d9c0
              exposeType: NodePort
              ports:

```

```

- name: kafka-ui
  type: webservice
  properties:
    image: provectuslabs/kafka-ui:cd9bc43d2e91ef43201494c4424c54347136d9c0
    exposeType: NodePort
    ports:
      - port: 8080
        expose: true
        nodePort: 30001
    cpu: "0.3"
    memory: "512Mi"
    env:
      - name: KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS
        value: "kafka-server:9092"

- name: video-capture
  type: webservice
  properties:
    image: registry.ubitech.eu/nebulous/use-cases/surveillance-dsl-demo/video-capture:1.1.0
    cpu: "0.2"
    memory: "100Mi"
    env:
      - name: KAFKA_URL
        value: "kafka-server:9092"
      - name: KAFKA_DETECTION_TOPIC
        value: "surveillance"
      - name: CAPTURE_VIDEO
        value: "False"
      - name: CAPTURE_DEVICE
        value: "/dev/video0"
      - name: DEBUG
        value: "True"
      - name: HOSTNAME
        value: "docker-capture"
    volumeMounts:
      hostPath:
        - name: video
          mountPath: "/dev/video1"
          path: "/dev/video0"
    traits:
      - type: affinity
        properties:
          nodeAffinity:
            required:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: "kubernetes.io/hostname"
                      operator: "In"
                      values: ["nebulousk8s-worker-1"]

- name: face-detection
  type: webservice
  properties:
    image: registry.ubitech.eu/nebulous/use-cases/surveillance-dsl-demo/face-detection:1.2.0
    edge:
      cpu: "1.2"
      memory: "512Mi"
    env:
      - name: KAFKA_URL
        value: "kafka-server:9092"
      - name: KAFKA_DETECTION_TOPIC
        value: "surveillance"
      - name: THREADS_COUNT
        value: "1"

```

```

- name: THREADS_COUNT
  value: "1"
- name: STORE_METRIC
  value: "False"
- name: DEBUG
  value: "True"
traits:
- type: affinity
  properties:
    podAntiAffinity:
      required:
        - labelSelector:
            matchExpressions:
              - key: "app.oam.dev/component"
                operator: "In"
                values: [ "video-capture" ]
            topologyKey: "test"
- type: nodePlacement
  properties:
    cloudWorkers:
      count: 6
      nodeSelector:
        - name: node1
          value: 2
        - name: node2
          value: 1
        - name: node3
          value: 3
    edgeWorkers:
      count: 3
      nodeSelector:
        - name: node4
          value: 2
        - name: node5
          value: 1
- type: geoLocation
  properties:
    affinity:
      required:
        - labelSelector:
            - key: "continent"
              operator: "In"
              values: ["Europe"]

- name: video-player
  type: webservice
  properties:
    image: registry.ubitech.eu/nebulous/use-cases/surveillance-dsl-demo/video-player:1.1.0
    exposeType: NodePort
    env:
      - name: KAFKA_URL
        value: "kafka-server:9092"
      - name: DEBUG
        value: "True"
      - name: SERVER_PORT
        value: "8081"
  ports:
    - port: 8081
      expose: true
      nodePort: 30002

```

Listing 1: video surveillance application modelling and deployment using the Open Application Model and KubeVela

KubeVela uses Custom Resource Definitions (CRDs) to implement OAM abstractions on top of Kubernetes. It manages application components and cloud resources by leveraging Kubernetes' capability to set up control loops (Controllers) for keeping the actual state of component instances in line with a desired configuration, thus avoiding configuration drift³¹. Desired configurations are expressed declaratively.

KubeVela uses the concept of OAM "Definitions" to provide automation for custom capabilities that may be attached to an Application. A Definition is written in the CUE language³², and it is then shared, discovered and used to compose an Application. Its goal is to hide complexity from developers by abstracting away significant chunks of underlying logic and allowing them to reuse out-of-the-box elements that have been created to implement a particular piece of functionality. There are four different types of definitions – *ComponentDefinition*³³, *TraitDefinition*³⁴, *PolicyDefinition*³⁵, and *WorkflowStepDefinition*³⁶ – each extending the corresponding application. KubeVela does provide built-in definitions that are readily available upon installation – examples include *cron-task*, *webservice* (*ComponentDefinitions*), *affinity*, *gateway*, *hpa* (*TraitDefinitions*), *apply-once*, *override*, *take-over* (*PolicyDefinitions*), *apply-component*, *build-push-image* and *deploy-cloud-resource* (*WorkflowStepDefinitions*). Available definitions can be found in the project's GitHub repo³⁷. A set of OAM definitions, along with their CRD controllers can be grouped into a KubeVela Addon. An Addon is a scenario-oriented extension of KubeVela, which is uploaded to a community-maintained registry and installable by any user. NebulOuS aims to take advantage of Definitions to automate interaction with underlying resources and thus offer to its users an intuitive way to deploy applications.

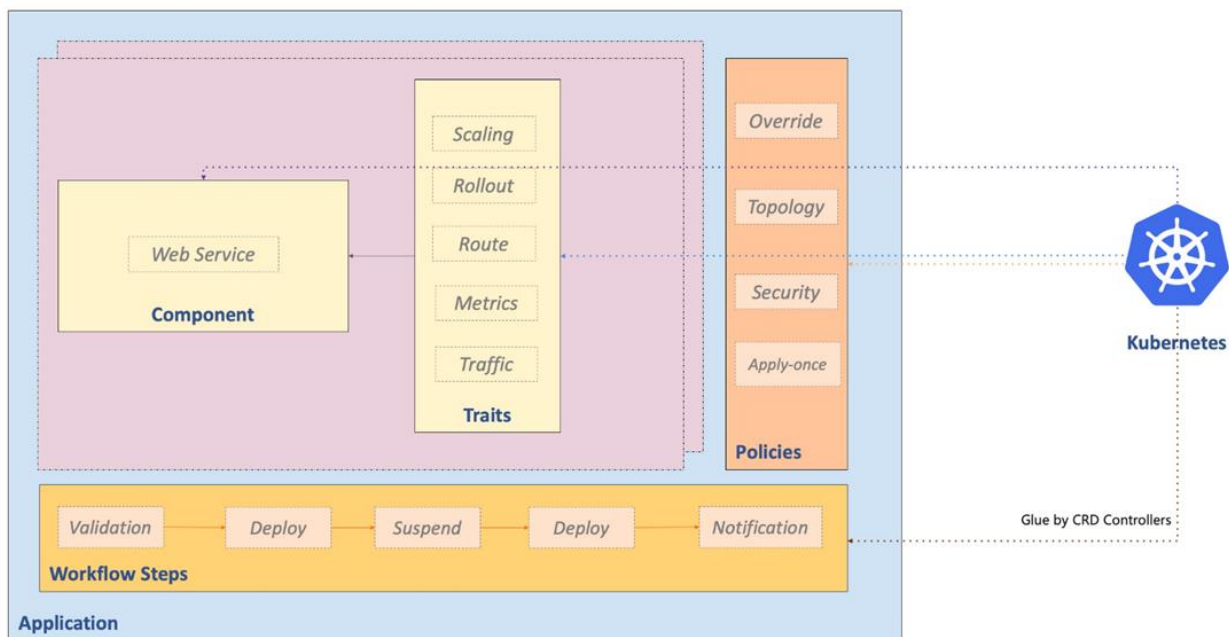


Figure 4: Application modelling abstractions in KubeVela

³¹ When deploying app components across different environments, changes are made according to each individual use case. This can lead to deployments drifting away from the original baseline configuration. As these changes add up, systems eventually start behaving inconsistently across environments. These issues are often difficult to diagnose and fix, especially since the changes are often undocumented. This process is commonly known as configuration drift.

³² <https://cuelang.org/>

³³ <https://kubvela.io/docs/end-user/components/references/>

³⁴ <https://kubvela.io/docs/end-user/traits/references/>

³⁵ <https://kubvela.io/docs/end-user/policies/references/>

³⁶ <https://kubvela.io/docs/end-user/workflow/built-in-workflow-defs/>

³⁷ <https://github.com/kubvela/kubvela/tree/master/vela-templates/definitions>

3. Metric Model

The *NebulOuS metric model* provides a technology-agnostic modelling framework for capturing the QoS requirements associated with hyper-distributed applications and for addressing their monitoring aspects. It is based on the metric model offered by the CAMEL DSL which it extends, amends, and rationalises for shifting its attention from multi-cloud apps to hyper-distributed ones in the Cloud computing continuum.

The NebulOuS metric model is underlain by the tenet that the different parts, or *components*, comprising a hyper-distributed app (e.g., app servers, databases, load balancers, etc.) may have different QoS requirements in terms of attributes being measured and attributes being computed. For this reason, it organises QoS-related artefacts around component **types**, or logical groupings of component types, called **scopes**. These artefacts include:

- **Metrics.** Quantifiable attributes for monitoring a property. Metrics can be *raw*, collected directly by monitoring sensors (also referred as *measurements*), or *composite*, computed using raw metrics or other composite metrics. Metrics are further elaborated in Section 3.1.
- **Requirements.** Top-level constructs used to express performance "expectations" from the application and the monitoring system. They take the form of metric constraints that are referred to as *service level objectives* (SLOs). Metric constraints are acceptable/desired or unacceptable/undesired metric values and value ranges. Requirements can be specified either per component type or per component *scope*, or at the application level (see Table 2). They are further specified in Section 3.2.

Scopes are introduced to facilitate the definition of common or combinatorial constraints that apply across different component types. Moreover, applications may be viewed as universal scopes, comprising all component types.

Table 2: Different types of scopes

Scope Types	Requirements	Metrics
Component	pertaining per component type	used in component type requirements
Scope	pertaining per scope	used in scope requirements
Application-scope	pertaining application-wide	used in application-wide requirements

3.1 Metrics

A *metric* is a measure that quantifies an application/system/environment attribute, which can be used to characterise the progress, performance, or status of an application aspect. Metrics are used to track specific data points over time¹⁵. Metric *values* (either *raw* or *computed*) are recorded as *events* and exchanged through event streams. Events are immutable, time-stamped, real-world facts (for a raw metric, e.g., a sensor measurement) or computations thereof (for a composite metric, e.g., the average over the last-minute sensor measurements).

Metric specifications implicitly define namesake *event streams* for conveying metric values. They also include several parameters describing how a metric is captured or computed, thus providing crucial information for configuring monitoring agents. These parameters are collectively referred to as the **metric context** and include: the sensor specification; the input metric *windows* in case of composite metrics (see below); the location/level where computations occur; the sampling rates of measurements or the times at which computations are performed, as different numbers and types of parameters may apply. For instance, a composite metric must include a **formula** that specifies the computation expression. On the other hand, a raw metric must include a

sensor definition. In case parameters are left unspecified, the corresponding default values will be used, however not all parameters have defaults, in which case they must be explicitly specified.

Listing 2 specifies a composite and a raw metric. The composite metric is calculated by averaging the `cpu_util_instance` raw metric values included in the corresponding data window. The `formula` parameter gives the computation expression (in the next example is `'mean(cpu_util_instance)'`), and the `window` section specifies how many and which input metric values (of `cpu_util_instance`) will be used in the computation (in the next example, it is all `cpu_util_instance` values arrived in the last 5 minutes). The `output` section specifies how often the resulting average will be calculated (i.e., every 30 seconds to `cpu_util_prct`). The raw metric collects its values using the sensor specified in the `sensor` section. Again, the `output` section specifies how often the measured values will be sent to the output event stream (every 30 seconds).

The rest of this subsection elaborates on the main metric context parameters.

```
metrics:
- name: cpu_util_prct
  type: composite
  formula: 'mean(cpu_util_instance)'
  window:
    type: sliding
    size: '5 min'
  output:
    type: all
    schedule: '30 sec'
- name: cpu_util_instance
  type: raw
  sensor:
    type: netdata
    affinity: netdata__system__cpu__total
  output:
    type: all
    schedule: '30 sec'
```

Listing 2: Example of metrics

3.1.1 Windows

Windows are finite sets of metric values, received from one or many input event stream(s) and retained based on a set of criteria. For instance, a window can contain the M latest events emitted on an event stream, or the events that arrived in the last N seconds. Windows can be either sliding or batch (tumbling). The former type has *moving* bounds and events can be added and removed as time passes (for example, events received in the last 10 seconds are retained in a 10-sec sliding window and older events are automatically removed). The latter type has *fixed* bounds and events can be added as long as the window is open; all events are automatically removed when the window closes/expires (for example, all events received during the interval 00:00:00 – 00:00:30 are retained until 00:00:30 and then automatically removed). Windows can be time-based or length-based or a combination of them. Time-based windows typically have a fixed time span (for example last 10 seconds), while length-based windows typically have a maximum allowed number of events (for example the last 10 events). Windows are required for operations such as aggregations, and pattern matching where several metric values (spanning time) are needed (for example an average over last 10 minutes, or detecting if an event occurred after another, while other events might have occurred in between).

Window Processings

Window processings are operations on windows that rearrange retained events based on a set of criteria. Three processing types are currently considered: *Grouping*, *Sorting* and *Ranking*. The first type segments window events into groups; the second type sorts window events and retains the top/bottom of them; the third type is similar to *Sorting* but retains only the most recent occurrence of an event based on uniqueness criteria. Any number of processings can be defined on a window but grouping processings take precedence. It is to be stressed that

processings are *not* data operations; for example, they do not aggregate or join data. They may influence, however, the scope of data operations. For instance, a Grouping processing may segment window data per IP address, and a subsequent averaging operation may be applied per window segment as opposed to the entire window. Sort and rank processings can influence the outcome of a data join (order of joined events) or the outcome of *first* / *last* event operations.

Listing 3 gives the specification of a grouping processing included in the window of `vc_instance_number` metric. The criteria parameter specifies how the window events will be grouped. In this case they will be grouped per application component instance, which can be implemented using the instance IP address. A number of predefined criteria are provided; namely `PER_INSTANCE`, `PER_HOST`, `PER_ZONE`, and `PER_REGION`. Apart from them, custom criteria can also be specified.

```
metrics:
- name: vc_instance_number
  formula: 'add(vc_instance_number_raw)'
  window:
    type: sliding
    size: '5 min'
    processing:
      - type: grouping
        criteria: PER_INSTANCE
```

Listing 3: Example of grouping processing

3.1.2 Sensors

Sensors are software units that measure an attribute or set of related attributes. They can be classified in various ways e.g., system or application-specific sensors (depending on their provider and what attributes they measure) or pull or push sensors. Pull sensors provide solicited-only measurements (e.g., in response to queries from the monitoring sensor), whereas push sensors actively emit unsolicited measurements. All sensor measurements are converted to timestamped events and sent to the event stream attached to the defining metric. For system-specific attribute measurements the well-known Netdata³⁸ monitoring agent will be used. Application-specific metrics can be obtained in a number of ways, including posting exporting them to Netdata, exposing them as Prometheus endpoint, or sending them to the monitoring system directly using AMQP protocol³⁹. Additional methods might be added if required by use cases.

3.1.3 Output

Output specifies how often events are sent to the metric's output stream. Metric values can be collected or calculated at any rate based on the availability of input data (either sensor measurements or input events); however, it may be desirable that their rate of emission is limited (throttling). Output is an optional parameter that specifies such a rate limit. In the example of Listing 4, the `schedule` parameter specifies the rate for sending metric events to the corresponding output stream. The `type` parameter is meaningful if more than one output events can be generated (measured or calculated) per period: `'all'` value causes all collected/calculated events to be sent, `'first'` value causes only the first one to be sent (the rest are discarded), and `'last'` value causes only the last one to send. If the output parameter is left unspecified, metric values are immediately relayed to the event stream upon collection from the corresponding sensor, or upon calculation.

³⁸ <https://www.netdata.cloud/>

³⁹ <https://www.amqp.org/>


```
metrics:
  - name: cpu_util_prct
    .....
  output:
    type: all
    schedule: '30 sec'
```

Listing 4: Example of output specification

3.1.4 References

References are pointers from one metric model building block to another that are used to avoid repetition of identical specifications. References use the name of the referenced block to refer to it; their effect is equivalent to replacing the references by the referenced block specifications. In the example of Listing 5 the *ref* value must be the name of another metric. The name must be fully qualified if the referenced metric is defined in another component or scope; i.e., <component/scope_name>.<metric/requirement_name>. Otherwise, the component or scope part can be omitted.

```
metrics:
  - name: VideoCaptureCardinality
    ref: '[video-capture].[instances]'
```

Listing 5: Example of references

3.2 Requirements

Requirements are named metric model structures defined per component type or scope, or application-wide. The content and interpretation of these structures depend on their subsequent usage (beyond the metric model). Currently, a single kind of requirements is provided, namely *Service-Level Objectives (SLOs)*, but more can be added in the future (e.g., if the NebulOuS use cases require it). SLOs are the target values or value ranges for a service level that is measured by a service-level metric. They provide a means for measuring the performance of a service. They are modelled as named *constraints*, where each constraint is a boolean expression evaluating a condition, typically whether a metric (usually computed) falls within an acceptable (or non-acceptable) value range. When the metric values fall within the acceptable range, the service performs as expected and the corresponding SLO is *fulfilled*. Otherwise, the corresponding SLO is *violated*, which is an indication that the service is not functioning at an adequate performance level. Such an event will generate a signal, which will also be a metric; the metric value is not important, only the presence of the violation metric is significant. This signal is called an *SLO violation event*.

```
requirements:
  - name: cpu_slo
    type: slo
    constraint: 'cpu_util_prct > 80'
  - name: ram_slo
    type: slo
    constraint: 'ram_util_prct > 80'
```

Listing 6: SLO requirement specification

Listing 6 illustrates an example specification of two SLO requirements. Their names (*cpu_slo* and *ram_slo*) implicitly specify two named event streams (with the same names, or with names deriving from them) for sending any SLO violation events. Their constraints are simple threshold checks involving a single metric (e.g., *cpu_util_prct* or *ram_util_prct* respectively), but more complex expressions are possible. The specifications of the metrics are given in the metric section of the metric model (see below).

3.3 Metric Model structure

A metric model document loosely follows the Kubernetes resource files style, and it typically comprises the following sections:

- Header and Metadata
- Components
- Scopes

Header

A **header** includes vital information for identifying the document type and determining how the remaining contents will be processed and interpreted. It must include at least the type and version of the model.

Metadata

The **metadata** section typically follows the header and includes additional information such as a display name for the model, as well as tags/labels characterizing the model (among many metric models) that can be used in conjunction with selectors.

Components

The **components** section contains a list of entries with each entry including a component name, a requirements subsection, and/or a metrics subsection. The requirements subsection encompasses component-specific requirements (SLOs), whereas the metrics subsection encompasses component-specific metric specifications. All component names must be defined in the corresponding application model documents (i.e., in KubeVela specifications).

```
scopes:
- name: app-wide-scope
  requirements:
  .....
  metrics:
  .....
- name: app-wide-scope
  components: [ MySQLdb, AppServer ]
  requirements:
  .....
  metrics:
  .....
```

Listing 7: Example of scopes

Scopes

The **scopes** section contains a list of named scopes. As already mentioned, **scopes** are logical groupings of two or more component types that facilitate the definition of requirements and metrics pertaining to more than one component type. A metric model specification may use the metrics from any component participating in the scopes section. Thus, scopes facilitate the specification of metrics combining input data from different component types. Each scope has a unique name and optionally a list of participating components. It also contains a requirements sub-section (with SLOs), and/or a metrics sub-section. The former subsection includes requirements that conjunctively bind all scope components. The latter subsection encompasses metrics available to the components participating in the scope. The example of Listing 7 gives the specification of two scopes: an application-wide scope, and one encompassing only the MySQLDB and AppServer components. The *components* parameter is an array declaring the components participating in the scope. Omitting this parameter means all components are included, hence it is an application-wide scope.

Listing 8 provides an example of a metric model (repetitive parts have been omitted to reduce length).

```

apiVersion: nebulous/v1
kind: MetricModel
metadata:
  name: face-detection-deployment
  labels:
    app: surveillance-demo-app
spec:
  components:
  - name: face-detection
    requirements:
      - name: cpu_slo
        type: slo
        constraint: 'cpu_util_prct > 80'
      - name: ram_slo
        type: slo
        constraint: 'ram_util_prct > 80'
      .....
    metrics:
      - name: cpu_util_prct
        type: composite
        template: &prct_tpl
          id: 'prct'
          type: real
          range: [0, 100]
          formula: 'mean(cpu_util_instance)'
          window:
            type: sliding
            size: '5 min'
          output:
            type: all
            schedule: '30 sec'
      - name: cpu_util_instance
        type: raw
        template: *prct_tpl
        sensor:
          type: netdata
          affinity: netdata__system__cpu__total
        output:
          type: all
          schedule: '30 sec'
      .....
  scopes:
  - name: app-wide-scope
    components: [ face-detection, ..... ]
    requirements:
      - name: sample_slo_combining_data_across_components
        type: slo
        constraint: ' sample_metric_combining_data_across_components > 10'
      - name: sample_optimisation_goal
        type: slo
        constraint: utility_var
      .....
    metrics:
      - name: sample_metric_combining_data_across_components
      .....

```

Listing 8: Metric model example (reduced)

3.4 Language and Style

The metric model provides two ways of defining a building block: (a) single-line/compact definition, and (b) multi-line/detailed definition. The former requires writing the definition as a string following a building-block-

specific convention/format, hence enabling its automatic conversion into a detailed definition (case b). The latter requires each setting to be provided on separate lines. The compact definition significantly improves readability, but manual editing can be error prone. The detailed definition is the one that should be used internally by the NebulOuS components. Listing 9 provides an example in compact format.

The metric model may be expressed in any popular serialisation format including YAML, JSON, and XML. In NebulOuS, we opt for YAML as it is the most human-readable syntax.

```
window: 'sliding 5 min'  
output: 'all 30 sec'
```

The same example in detailed format:

```
window:  
  type: sliding  
  size: '5 min'  
output:  
  type: all  
  schedule: '30 sec'
```

Listing 9: Examples of specification styles in detailed format

4. Optimisation DSL

Any operating system deals primarily with the management and distribution of resources to satisfy the needs of running applications. Seen from one application, NebulOuS is therefore fundamentally about automatic application management across several underlying infrastructures hosting the resources necessary for running the application. Automating application management requires combining knowledge in four different areas: the application topology, the information about available resources, measurements of the current state of the application and the environment, and the definition of the goals and objectives for the application.

4.1 Parameterised application topology model

Figure 3 showed the fundamental structure of a KubeVela YAML application specification. For deployment, this model must be entirely specified and unique. However, this implies that all necessary decisions have been made à priori. Hence, this model cannot be used as input for the automatic application management since there is no information about the possible alternatives for the application configuration.

As an example, consider the description of one application component labelled "video surveillance" from a video stream application shown in Listing 1. There are resource requirements for the component in lines 116-118 and 130-132 of Listing 10. These requirements restrict the choices for the resources necessary to deploy and execute the component. Furthermore, there are placement restrictions for the component and the number of component instances in lines 157-172 of Listing 11. In their current form, none of these requirements exhibit any variability.

To use the KubeVela description with NebulOuS, one must indicate where *decisions* are to be made and values from the decision process inserted. Adopting the notation that square braces mean items to be specialised, and a dot notation to bind it to the right semantic understanding of the meaning of the variables, one adopts a notation for the KubeVela file like the one illustrated in Listing 1 for the resource requirements with ranges for the possible values the application component can properly use. The initial application deployment will always happen with the least possible resources.

```

116     edge:
117         cpu: {faceDetection.edge.cpu in {1.2..3.0} }
118         memory: {faceDetection.edge.memory in {250..1000} }
119         env:
-----
130     cloud:
131         cpu: {faceDetection.cloud.cpu in {3.0..6.0} }
132         memory: {faceDetection.cloud.memory in {1000..4000} }

```

Listing 10: The resource requirements represented as decision variables with ranges of possible values

The same idea and illustrative syntax can be adopted for the deployment specifications as shown in Listing 13. This example specification is incomplete without the supporting constraints: there should be constraints among the proposed variables since there is now nothing preventing the maximum possible Cloud worker instances to be deployed on every possible Cloud node, even if the total number of Cloud nodes in that case would exceed the upper limit for the total number of Cloud workers.

```

157 ▼ cloudWorkers:
158     count: {faceDetection.cloudWorkers.count in {2..10} }
159 ▼     nodeSelector:
160 ▼         - name: {faceDetection.cloudWorkers.cloud.node.label[ faceDetection.cloudWorkers.cloud.node.count ];
161             faceDetection.cloudWorkers.cloud.node.count in {1..3} }
162 ▼         value: {faceDetection.cloudWorkers.cloud.node.instances[i] in {0..faceDetection.cloudWorkers.count};
163             i in {1..faceDetection.cloudWorkers.cloud.node.count} }
164
165
166 ▼     edgeWorkers:
167     count: {faceDetection.edgeWorkers.count in {0..5} }
168 ▼     nodeSelector:
169 ▼         - name: {faceDetection.edgeWorkers.edge.node.label[ faceDetection.cloudWorkers.edge.node.count ];
170             i in {1..faceDetection.cloudWorkers.edge.node.count} }
171 ▼         value: {faceDetection.edgeWorkers.edge.node.instances[i] in {0..faceDetection.edgeWorkers.count};
172             i in {1..faceDetection.cloudWorkers.edge.node.count} }

```

Listing 11: The parameterised placement instructions as index variables with value ranges

It should be emphasized that the terms and modelling concepts used in the parameterised KubeVela model must be anchored in the semantic asset (see section 6) model describing the available resources and vocabulary across all application models. Furthermore, the ranges for the different variable domains indicated in the example of Listing 13 are defined in the user interface and exported from there to the optimisation model. Therefore, they need not be a part of the parameterized component model. However, leaving the domains out may be confusing since some of the domains may have secondary index variable definitions, like the `faceDetection.cloudWorkers.edge.node.count`, and then it may not be possible to validate the parameterized model for debugging purposes. On the other hand, since all the information necessary for the parameterized model will be collected in the user interface, and explicit export of this knowledge as a parameterized model file may not be necessary at all.

```

112 - name: face-detection
113   type: webservice
114   properties:
115     image: registry.ubitech.eu/nebulous/use-cases/surveillance-dsl-demo/face-detection:1.2.0
116     edge:
117       cpu: "1.2"
118       memory: "512Mi"
119     env:
120       - name: KAFKA_URL
121         value: "kafka-server:9092"
122       - name: KAFKA_DETECTION_TOPIC
123         value: "surveillance"
124       - name: THREADS_COUNT
125         value: "1"
126       - name: STORE_METRIC
127         value: "False"
128       - name: DEBUG
129         value: "True"
130     cloud:
131       cpu: "1.2"
132       memory: "512Mi"
133     env:
134       - name: KAFKA_URL
135         value: "kafka-server:9092"
136       - name: KAFKA_DETECTION_TOPIC
137         value: "surveillance"
138       - name: THREADS_COUNT
139         value: "1"
140       - name: STORE_METRIC
141         value: "False"
142       - name: DEBUG
143         value: "True"
144     traits:
145       - type: affinity
146         properties:
147           podAntiAffinity:
148             required:
149               - labelSelector:
150                   matchExpressions:
151                     - key: "app.oam.dev/component"
152                       operator: "In"
153                       values: [ "video-capture" ]
154                   topologyKey: "test"
155       - type: nodePlacement
156         properties:
157           cloudWorkers:
158             count: 6
159             nodeSelector:
160               - name: node1
161                 value: 2
162               - name: node2
163                 value: 1
164               - name: node3
165                 value: 3
166           edgeWorkers:
167             count: 3
168             nodeSelector:
169               - name: node4
170                 value: 2
171               - name: node5
172                 value: 1
173       - type: geoLocation
174         properties:
175           affinity:
176             required:
177               - labelSelector:
178                   - key: "continent"
179                     operator: "In"
180                     values: ["Europe"]

```

Listing 12: A KubeVela defined component for facial detection with resource requirements in red boxes and component placement instructions in the green box.

4.2 Optimisation model

The optimisation model fundamentally describes the constraints and objectives of the automatic application management problem. The constraints define when a deployment configuration is valid, and a new configuration should be found when the current configuration is infeasible. All the decision variables described above from the parameterised KubeVela file, and the metrics of the metric model can be used in the constraints and utility objective function(s) defining the goals of the application management from the perspective of the owner of the application. The application utility typically has many different *dimensions*, i.e., possibly conflicting goals, and it is assumed that these dimensions are modelled as individual utility functions that may or may not be scalarised into a single utility function to maximise.

The need to specify an optimisation problem in a machine interpretable format has long been recognized by the operational research community, and A Mathematical Programming Language (AMPL⁴⁰) appeared first in 1985 and has since been continuously expanded and improved. AMPL is an algebraic DSL capable of describing large and complex optimisation problems [13]. The AMPL parser, runtime interface and interpreter are today commercial software, but there is a free community version⁴¹ available for research and industrial prototyping. AMPL has been adopted as a starting point for the NebulOuS project, with the understanding that the runtime interface and interpreter require replacement in the integrated NebulOuS. This should happen without the need to change the textual AMPL program description.

The AMPL struct ‘param’ is used to represent constants and metric values, this includes parameters coming from the selection of the node candidates to be used for the deployment of the application. This could be for instance the negotiated price to use a virtual machine (VM) on an Edge server, as this price could fluctuate according to demand for using the Edge server and the actual price to be used ‘now’ will vary with the time of the resource availability request. Other parameters of the models can be given as a static data, for instance, the Cloud providers and Edge providers that can be used for the deployment. The variables of AMPL are used to represent the decision variables exemplified in the previous section.

The information in the AMPL file is used in expressions to calculate the utility value in the unit interval giving a value between zero and unity. The optimisation aims to find assignments to the variables so that the utility value is maximized. These value assignments are subject to constraints over the variables and the parameters of the problem. An example of an AMPL formulation of the parameterized topology model for the facial recognition component of the previous subsection will be presented next. The example also presents the naming convention that will be used in the AMPL models used in NebulOuS, where the type of the variable is indicated by the last part of the name, after the component type, and the deployment type (e.g., `faceDetection.edge.cpu`).

Listing 13 shows the definition of the requirements for the component to run and the allowed options for the deployment. The intervals mean any real number in the range and the integer interval means any integer value in the range; hence, one may want to rather replace the intervals for the memory requirements with a set of possible values instead of allowing all integer values in the given range. Listing 14 shows the definitions for the multiplicity variables for the number of workers. There are two scalar variables representing the total number of workers in the Cloud and in the Edge respectively. Then there are index variables over the set of providers of each type to represent the number of workers on each provider and each type. Since the total number of workers of a type on all providers must add up to the total number of workers for that type, there are constraints to enforce this restriction per provider type.

⁴⁰ <https://ampl.com/>

⁴¹ <https://ampl.com/ce/>


```
# Illustrative model for the 'face-detection' component
# Component resource requirements
# Values in meaningful units
var faceDetection.edge.cpu in interval [1.2, 3.0];
var faceDetection.edge.memory in integer [250, 1000];
var faceDetection.cloud.cpu in interval [3.0, 6.0];
var faceDetection.cloud.memory in integer [1000, 4000];
# Cloud and edge providers
set CloudProviders := AWS Google Azure; set EdgeProviders := TID Orange Vodaphone Swisscom;
```

Listing 13: Representing the application deployment requirements.

```
# Number of workers to deploy and at different locations paint
var faceDetection.cloudWorkers.count in integer [2, 10];
var faceDetection.edgeWorkers.count in integer [0, 5];
var faceDetection.cloudWorkers.location{ p in CloudProviders } in integer [0, faceDetection.cloudWorkers.count];
var faceDetection.edgeWorkers.location{ p in EdgeProviders } in integer [0, faceDetection.edgeWorkers.count];
# Making sure to deploy correct number of workers over all locations
subject to CloudWorkerLimit :
    sum{ p in CloudProviders } faceDetection.cloudWorkers.location[p] == faceDetection.cloudWorkers.count;
subject to EdgeWorkerLimit :
    sum{ p in EdgeProviders } faceDetection.edgeWorkers.location[p] == faceDetection.edgeWorkers.count;
```

Listing 14: The total multiplicity of the various worker types broken down in workers per location

Listing 15 presents the node identifiers on each provider as parameters that must be given as sparse matrices where the first, row index is for the named providers, and the second, column index is for the number of workers at that provider. Hence, not all rows need to have the same number of columns. The number of worker instances allocated to each node at each provider is represented similarly. Finally, there must be constraints to ensure that the allocation of workers across all nodes of a provider matches the number of workers to be allocated to that provider.

```
# Label the nodes at each provider the range is set so that there are as many nodes as
# there are workers at each provider to accommodate the case where there is only one worker per node.
param CloudNodeIDs{ p in CloudProviders, 0..faceDetection.cloudWorkers.location[p] };
param EdgeNodeIDs{ p in EdgeProviders, 0..faceDetection.edgeWorkers.location[p] };
# Specific deployment decision variables with the constraint that the sum of nodes on
# each provider matches the sum of all providers
var faceDetection.cloudWorkers.cloud.node.instances { p in CloudProviders,
1..faceDetection.cloudWorkers.location[p] } in integer [0, faceDetection.cloudWorkers.location[p] ];
var faceDetection.edgeWorkers.cloud.node.instances { p in EdgeProviders, 1..faceDetection.edgeWorkers.location[p]
} in integer[0,faceDetection.edgeWorkers.location[p]];
subject to CloudNodeWorkerLimit:
    sum{ p in CloudProviders, id in integer [1, faceDetection.cloudWorkers.location[p] ] }
    faceDetection.cloudWorkers.cloud.node.instances[p, id] == faceDetection.cloudWorkers.location[p];
subject to EdgeNodeWorkerLimit:
    sum{ p in EdgeProviders, id in integer [1, faceDetection.edgeWorkers.location[p] ] }
    faceDetection.edgeWorkers.edge.node.instances[p, id] == faceDetection.edgeWorkers.location[p];
```

Listing 15: Deployment node identifiers and number of worker instances per node

Listing 16 shows the definition of deployment cost starting from the individual node costs based on the already established node identifiers. This is essentially given as two vectors of values for the possible deployment nodes. Since the cost is changing dynamically depending on the market demand for each type of node, the available nodes and the cost tables will be continuously updated to reflect the current availability for deployment. Then two auxiliary parameters are calculated to compute the cost of the selected nodes from each provider. This is done by summing over all possible node indices, but selecting only those node indices for which the number

of instances is larger than zero. The result of these calculations is stored in two vectors indexed by the Cloud providers and Edge providers respectively. Finally, given the deployment budget parameter, the total deployment cost for the Cloud part and the Edge part are calculated individually and then compared against the budget in a constraint.

```
# Cost parameters to be set for the available node candidates
# Values in some currency
param CloudNodeCost{ id in CloudNodeIDs };
param EdgeNodeCost{ id in EdgeNodeIDs };
# Then calculate the total deployment cost for Cloud and Edge
param TotalProviderCloudCost{ p in CloudProviders }
  = sum{ n in faceDetection.cloudWorkers.location[p] :
    aceDetection.cloudWorkers.cloud.node.instances[ p, n ]>0 }
    ( CloudNodeCost[ CloudNodeIDs[p, n] ] );
param TotalProviderEdgeCost{ p in EdgeProviders }
  = sum{ n in faceDetection.edgeWorkers.location[p] :
    faceDetectionedgeWorkers.edge.node.instances[ p, n ]>0 }
    ( EdgeNodeCost[ EdgeNodeIDs[p, n] ] );
# Cost constraint on the number of workers
param DeploymentBudget;
param TotalCloudCost = sum{ p in CloudProviders } TotalProviderCloudCost[p];
param TotalEdgeCost = sum{ p in EdgeProviders } TotalProviderEdgeCost[p];
subject to DeploymentCostConstraint :
  TotalCloudCost + TotalEdgeCost <= DeploymentBudget;
```

Listing 16: The cost constraints of the optimisation problem

Given all the variable and constraint definitions of the previous listings, one must formulate at least one deployment goal or objective. This could be to deploy at minimal cost. However, the minimal cost is an application that has zero worker instances, and it is therefore not a realistic objective without also providing a goal related to the application performance. Since the role of the facial detection component is to analyse images, one can imagine that the number of images awaiting processing over the next time unit, for instance minutes, will be measured by the application and submitted to the event management system via an appropriate sensor to a metric, which is here called `ImagesToProcess`. Recall that the optimiser will subscribe to all ‘parameters’ that are not composite parameters that can be directly calculated from other definitions, and so this parameter will be replaced with its current metric value before optimising the configuration.

Secondly, there must be a way to measure the processing capabilities of the facial recognition component. This capacity depends on the complexity of each image, but also on the node hosting the facial recognition component since the underlying hardware has different capabilities. The result is that the processing time per image will be a stochastic quantity that can be reported by the facial component after completing the processing of each image. The event management system will then be able to compute the empirical distribution of these measurements and calculate the upper quantile of this distribution. Essentially, this quantile will represent an upper bound on the computing time needed to finish computation, for instance that 80% of the images are processed using less computing time than this upper quantile value. The upper bound on the number of images that is to be served by one single facial recognition component is then the length of the time interval divided by the upper bound on the image computation time. Hence, this can be used to find the number of facial recognition components needed to process the images in the queue over the time interval available for their processing.

The utility of the application is obviously best if exactly the number of queued images can be processed in the next time interval as the application provides the correct amount of facial processing components. The utility is decreased if less than the queued number of images can be served, but also if more than the number of queued

images will be served as it is an indication of overprovisioning, and therefore a costlier deployment than needed to do the job. The utility function calculations are shown in Listing 17.

```
# There will be two utility objectives for this deployment:
# The first objective aims at minimising the total cost of the deployment.
minimize Cost:
    TotalCloudCost + TotalEdgeCost;
# The second objective aims to provide enough facial detection components to be
# able to process the queued number of images.
param ImagesToProcess;
param UpperQuantileImagesProcessingTime;
param TimeIntervallLength = 60s;
param UpperQuantileNoImagesPerComponent = TimeIntervallLength /
    UpperQuantileImagesProcessingTime;
maximize Performance:
    1/exp( (ImagesToProcess - UpperQuantileNoImagesPerComponent
        * (faceDetection.cloudWorkers.count + faceDetection.edgeWorkers.count) )^2 );
```

Listing 17: The utility calculations for the facial detection component workers problem

4.3 NebulOuS Integration

As can be seen from the worked example in the previous sections, there must be a strong consistency between the variability definitions in the parameterised topology model; the metric model providing advanced calculations of composite metrics whose values are functional combinations of other metric values, e.g., the quantile of the empirical computation time distribution used in this example; and the formulation of the constraints and the utility functions. The semantic model will provide a framework for this modelling and the capabilities of the infrastructure available for the deployment, and the user interface will support the application owner's definitions of the involved functional expression linking these to the available metrics and component variability variables.

5. Resource Discovery Mechanism

This section outlines a Resource Discovery mechanism designed to efficiently register, discover, monitor, and manage fog and edge devices within the cloud computing continuum. It emphasizes secure communication, detailed device profiling, health monitoring, administrative control, and data persistence for auditing and resource management purposes.

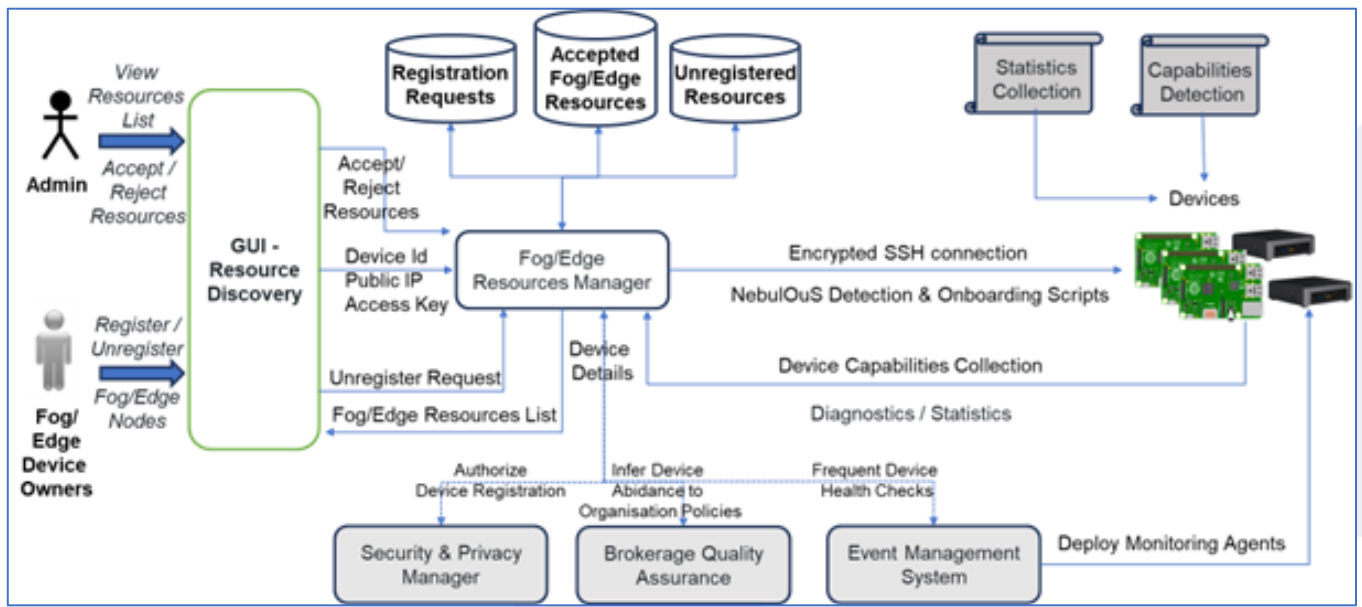


Figure 5: Overview of NebulOuS Resource Discovery mechanism

5.1 Registering Fog/Edge devices to NebulOuS

The Resource Discovery mechanism allows fog and edge device owners to register their devices with NebulOuS. These devices are potential candidates for deploying application component instances (or workloads). As seen in Figure 1, device owners use a graphical user interface (GUI) to provide necessary details of their devices, such as Device Id, Public IP, Credentials, Location etc. Public IP and connection credentials are important for allowing NebulOuS to connect and set up these fog or edge devices, while other additional details collected (e.g., Location) will be used later on by the Cloud/Fog Service Broker to generate the available resource pools that are appropriate for a specific application. These details are then used by the Fog/Edge Resources Manager, a dedicated component of the Resources Discovery mechanism, to connect to the device through an SSH connection and execute appropriate scripts to detect/identify the capabilities of the device, and to install an appropriate monitoring agent for collecting health status data. Once all the information is collected the device details are persisted in a no-SQL database that holds details for all Fog/Edge resources available to the system. They are also ontologically captured as part of the asset model for interoperability purposes.

Incoming registration requests are checked against predefined access control rules and application consumption policies. More specifically:

- A first pre-authorization step takes place by invoking the NebulOuS Security and Privacy Manager to check against available access control rules. At this stage a filtering of allowed devices can be made based on the rights of the owner of the device and/or its location (e.g., providers/users whitelist/blacklist, permitted geographical regions etc.).
- A second authorization step takes place after the Fog/Edge Resources Manager has aggregated all device capabilities. This authorization step comprises a comparison check between the nominal and real device capabilities and/or a check based on the real capabilities and the minimum requirements that have been defined for security or performance/quality assurance purposes. For the latter, the Broker Quality

Assurance mechanism is invoked in order to infer the device’s abidance to higher-level *application consumption policies*. These are policies that operate at a higher level of abstraction and express a broader set of business and security requirements that characterise an application as opposed to an application workload (instance). For example, a policy may impose minimum limits on the compute capacity that must be assigned to an application component; any QoS requirement attached to a workload (running instance) of this component must respect these limits. In other words, we are envisaging a situation whereby QoS requirements potentially vary across different deployed workloads of an application component, whilst abiding by an overarching set of application consumption policies. Consider, for instance, the following scenario. An organisation develops an IoT application and sets an application consumption policy that imposes minimum limits on the CPU cores and RAM size that must be available to an application execution. The organisation then deploys application instances at different locations to serve the needs of its customers (one deployed instance per customer is assumed). Customers are free to set their own QoS requirements on their application instances if these abide by the overarching application consumption policy.

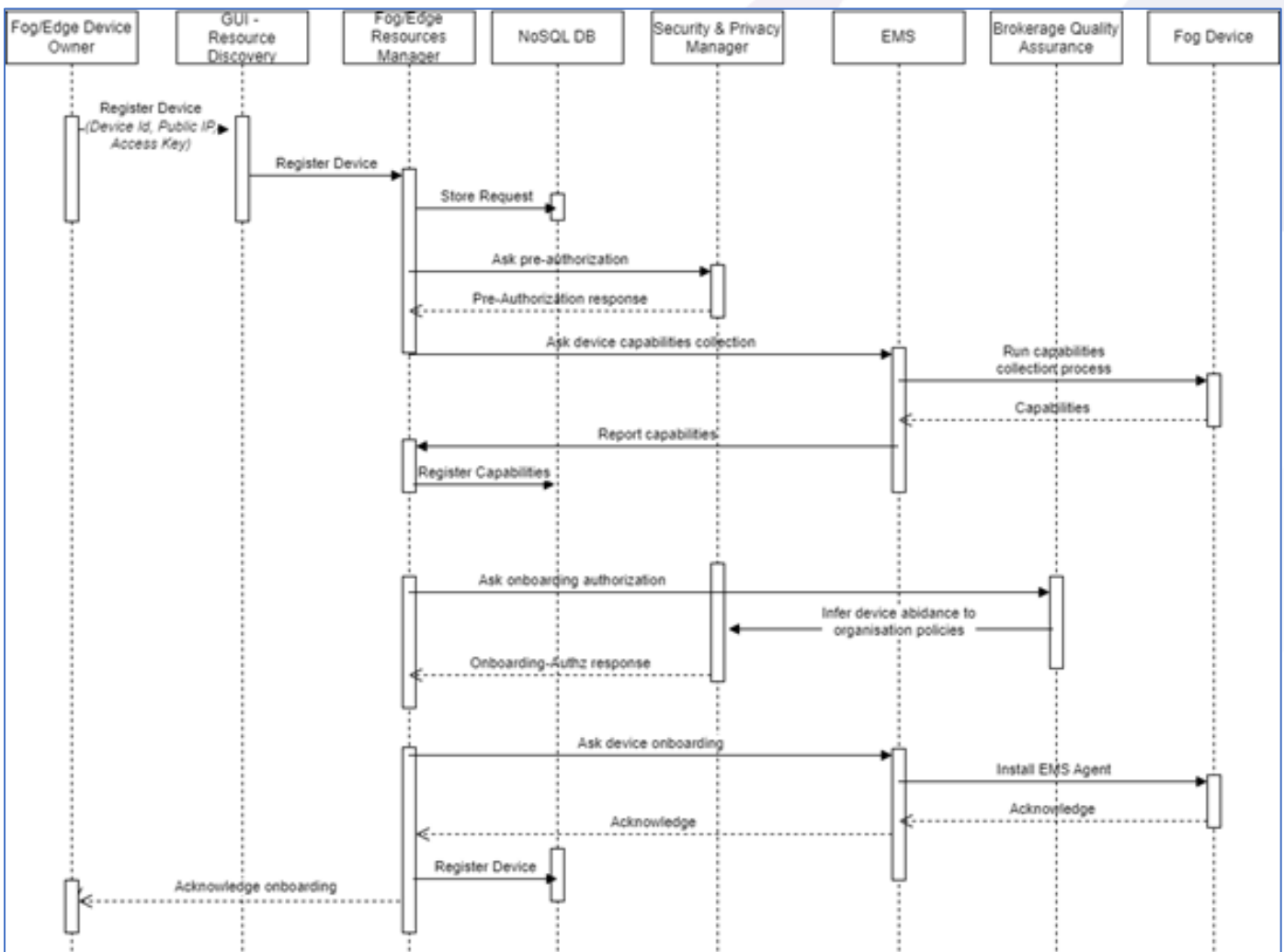


Figure 6: Sequence diagram for registering Fog/Edge devices

Through the GUI a NebulOuS administrator is able to accept or reject the registration of any new device. All incoming devices registration requests are persisted in a no-SQL store for auditing purposes. Figure 6 depicts a sequence diagram that encapsulates the Fog/Edge devices registration process.

The list of available devices, along with their details (location, processing capacity, network quality etc.), will be used by the Optimiser (see Section 4) to determine, based on the set of user preferences, which subsets of

these resources can satisfy application provisioning requirements and preferences, and therefore formulate an adequate resource pool.

5.2 Unregistering Fog/Edge devices to NebulOuS

Fog and edge devices are unregistered in three occasions: i) when a device owner requests the withdrawal of a device; ii) when a NebulOuS administrator requests the withdrawal of a device; iii) when the monitoring system detects that the device is unreachable for more than a given amount of time (and retries). If the unregistered device is already commissioned hosting application instances, a reconfiguration process is triggered that relocates all hosted application instances.

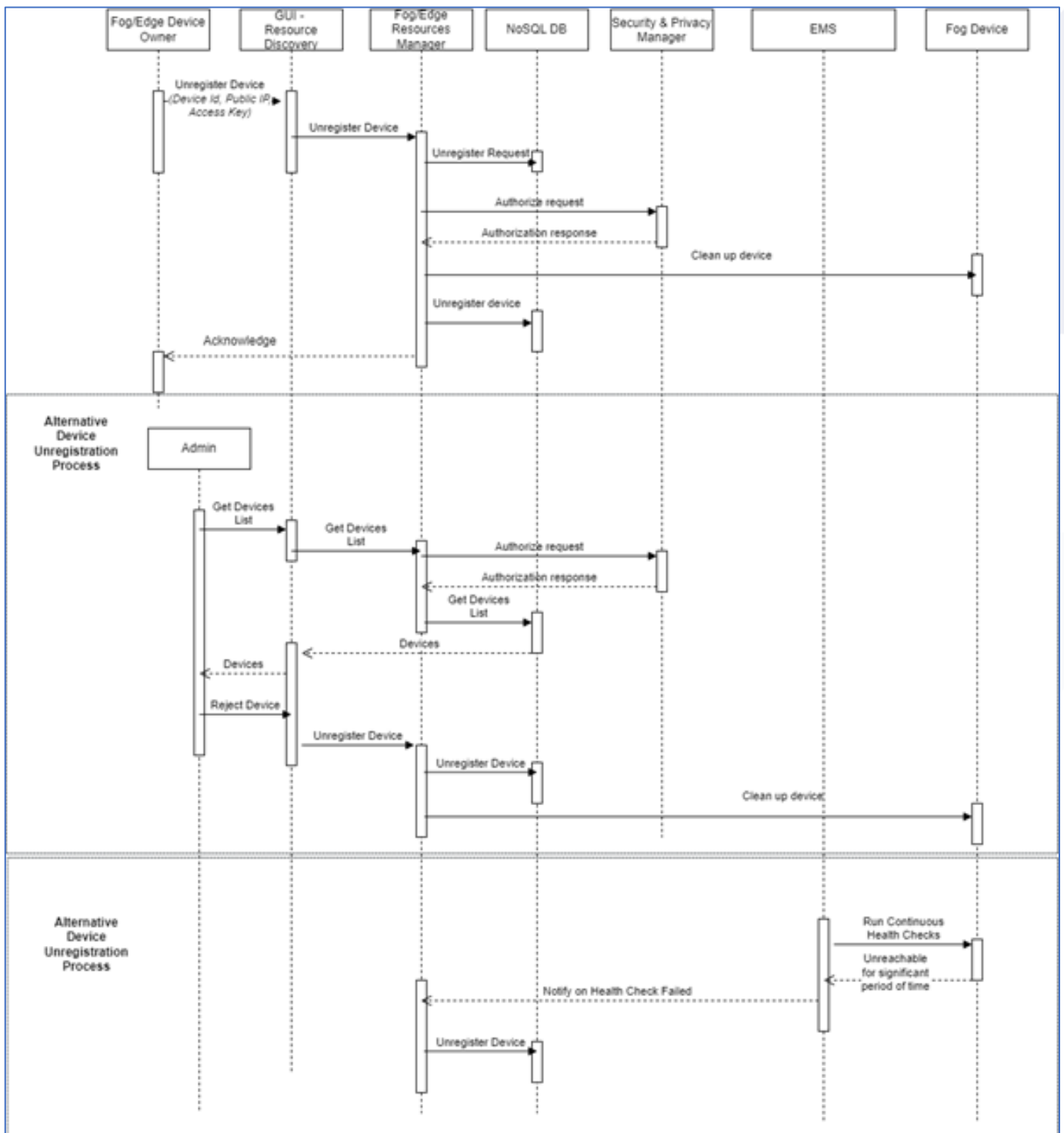


Figure 7: Sequence diagram for unregistering Fog/Edge devices

An authorization step takes place every time an unregistration request arrives at the Fog/Edge Resources Manager, by invoking the NebulOuS Security and Privacy Manager. This check:

- Determines whether the device is already commissioned hosting specific component instance(s).
- Evaluates the request against the available access control rules that determine the entities that are allowed to perform a device unregistration.

The Resource Discovery mechanism maintains a data collection with records of all devices that have been unregistered from the pool of available resources. This database can be used for tracking device lifecycle events, and for reference when assessing resource availability and providers' reputation. Figure 7 depicts a sequence diagram that analyses the “unregistration” process of Fog/Edge devices.

5.3 Resource Discovery

As part of Task 2.4, we designed and developed a first prototype of the NebulOuS Resource Discovery mechanism that is capable of registering/unregistering fog and edge devices. These devices form part of the transient cloud computing continuum, and therefore consolidate resource pools that will be used by the NebulOuS Optimiser as node candidates for deploying application components. This first release of the mechanism implements only basic functionality and can be accessed here: <https://opendev.org/nebulous/resource-manager>. More comprehensive functionality including the authentication steps outlined above is deferred for the 2nd release. The rest of this section provides screenshots of the Resource Discovery mechanism's GUI.

After successful authentication, the user (either admin or device owner) is directed to the Resource Discovery mechanism's dashboard. From there they can navigate to specific service sections, like new device registration (for onboarding), onboarded devices monitoring and management, archived registration requests and devices offboarded, and a settings section.

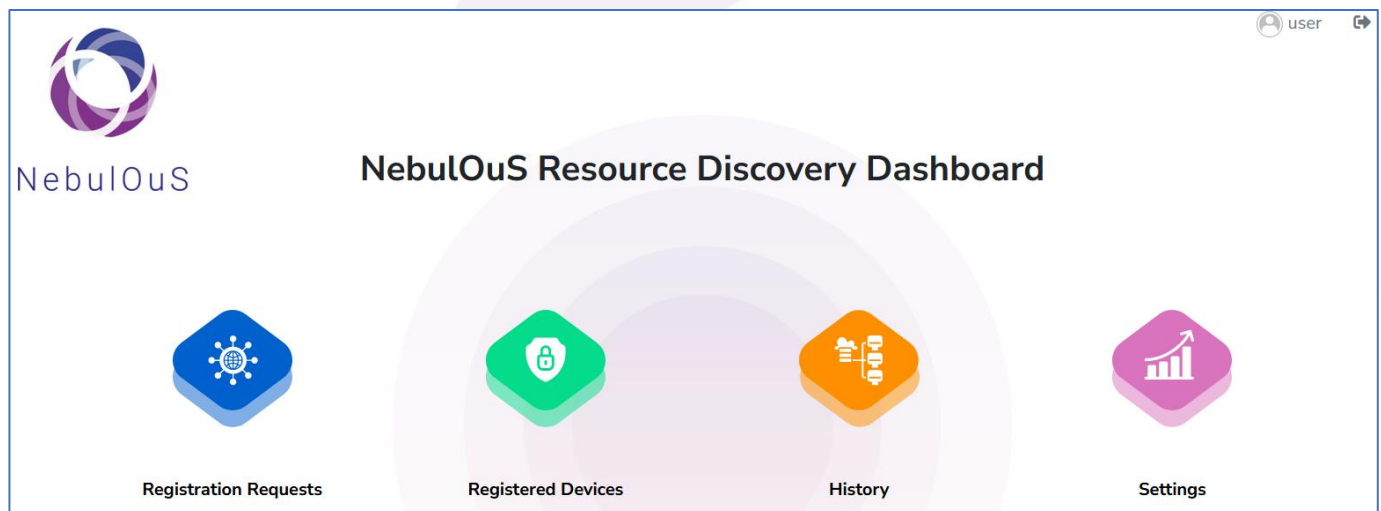


Figure 8: Resource Discovery mechanism dashboard

In the device registration section, a device owner can view a list of all open registration requests he/she has submitted, as well as create new ones. Past requests that completed either successfully or unsuccessfully are listed at the History section.

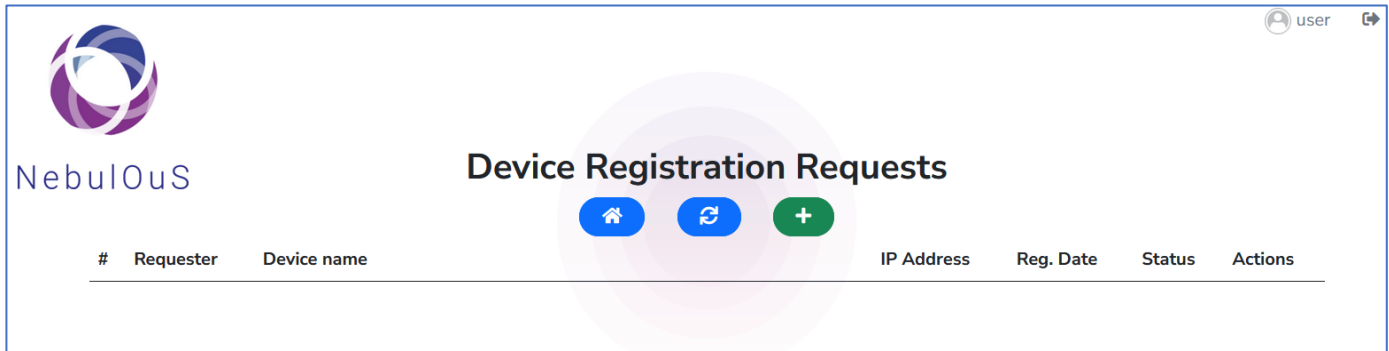


Figure 9: List of open user's registration requests (currently empty)

When creating a new request, the device owner is required to fill important device data like its IP address, a unique device Id, SSH connection credentials, as well as optional information like a human-readable device name, device capabilities etc.

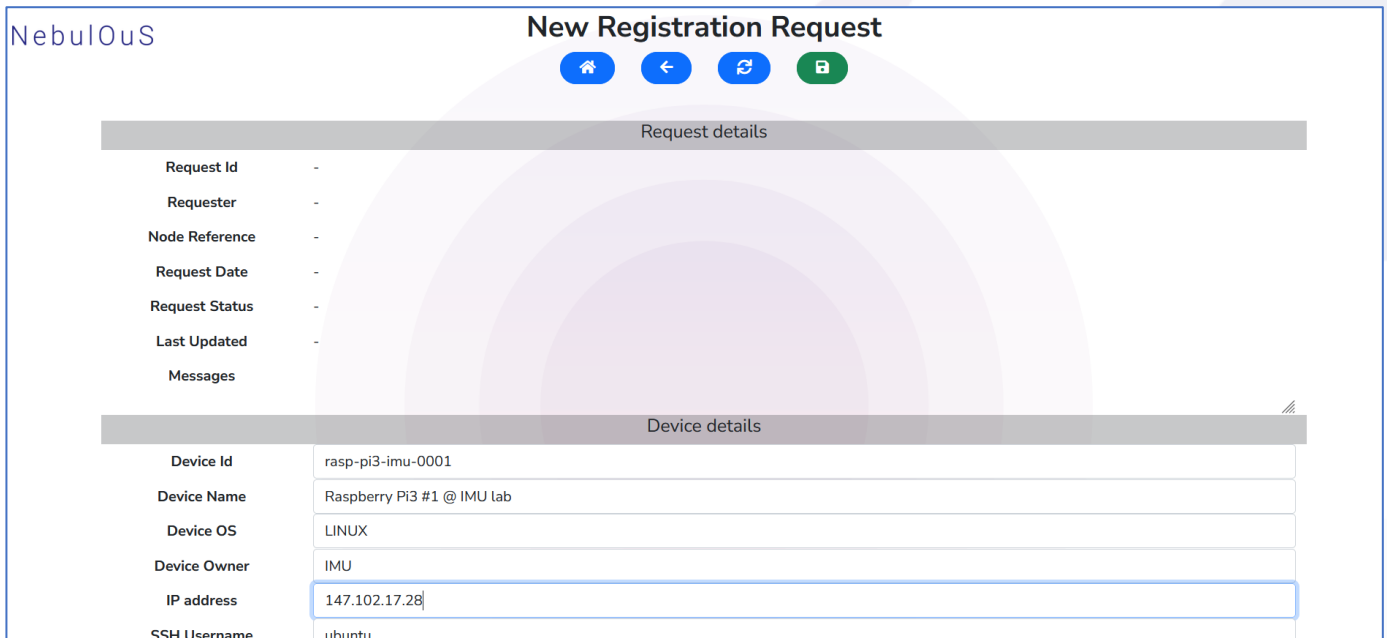


Figure 10: New device registration request form

After submitting the registration request, it will receive a unique Request Id, its status will be set to `NEW_REQUEST` and certain administrative info will be recorded (like creation date and owner). It will also be listed in the list of open requests.

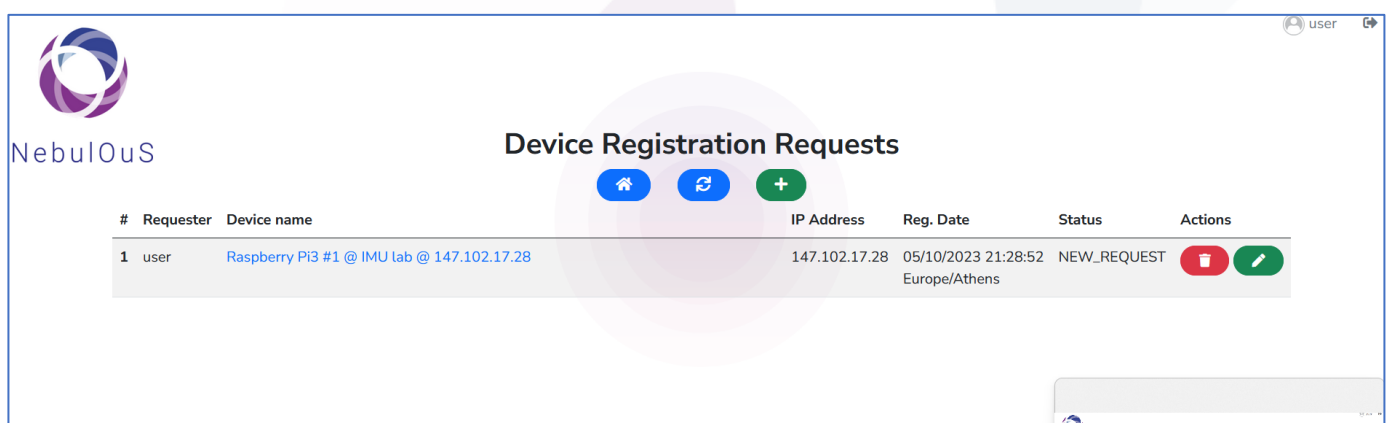


Figure 11: List of open user's registration requests (with a new request)

Periodically the Resource Manager will process new requests, by attempting to connect to the described devices and detect their capabilities. The collected capability data are then stored along with the request. During device capability collection the request status changes to DATA_COLLECTION_REQUESTED.

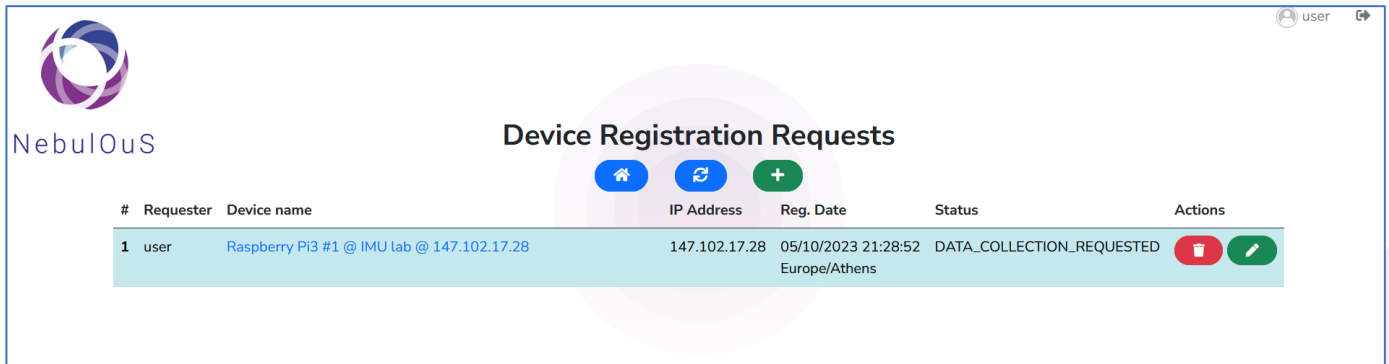


Figure 12: List of open user’s registration requests – Collecting device capabilities

The collected device capability data can be viewed and edited in the request form.

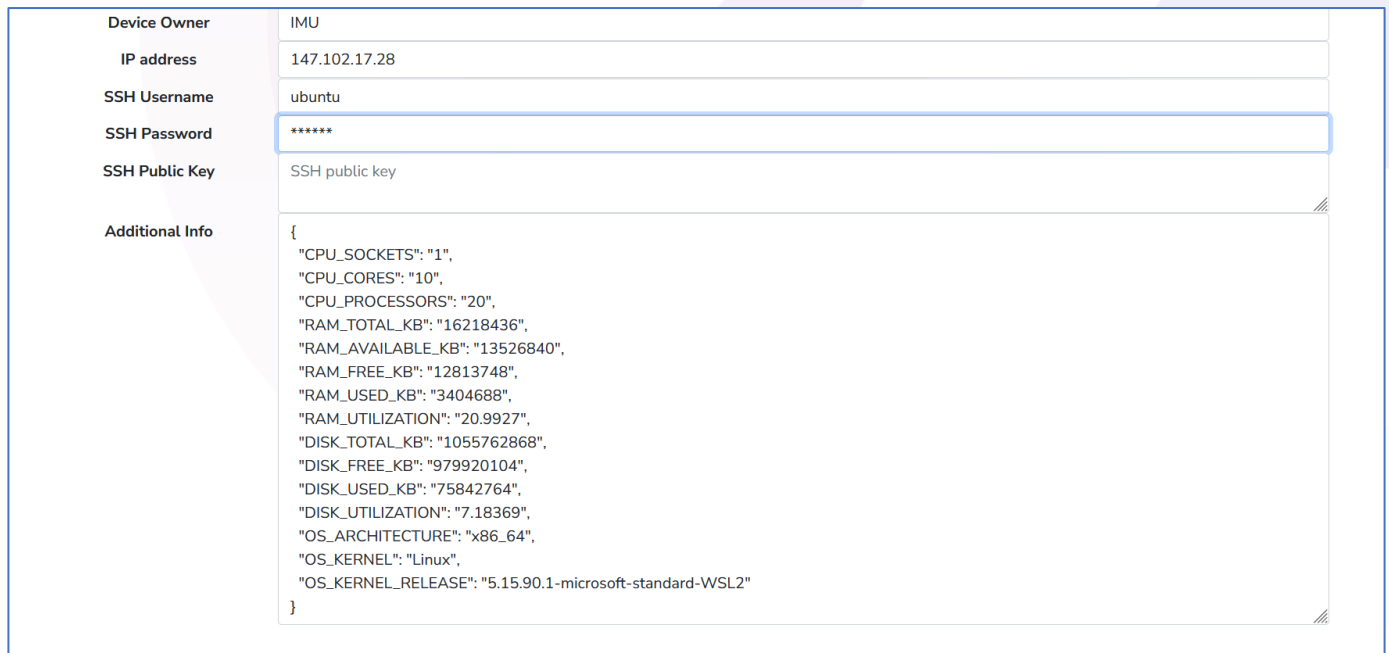


Figure 13: Device capabilities as collected by Resource Manager and stored in database – Request has been updated

After device capability data collection, the Resource Manager will request authorization for onboarding the device. This is indicated by setting request’s status to PENDING_AUTHORIZATION.

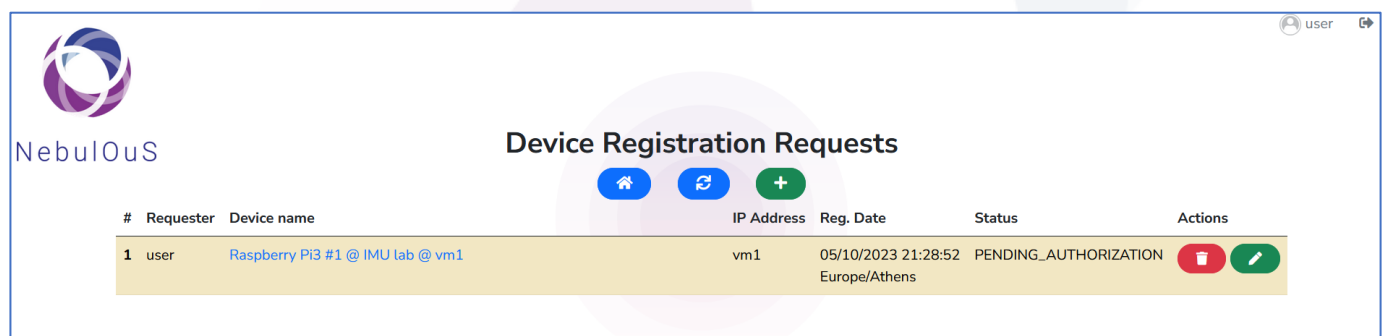


Figure 14: List of open user's registration requests (Request awaits authorization, after capabilities collection)

Administrator can manually authorize a device onboarding. However automated authorization procedures will also be employed.

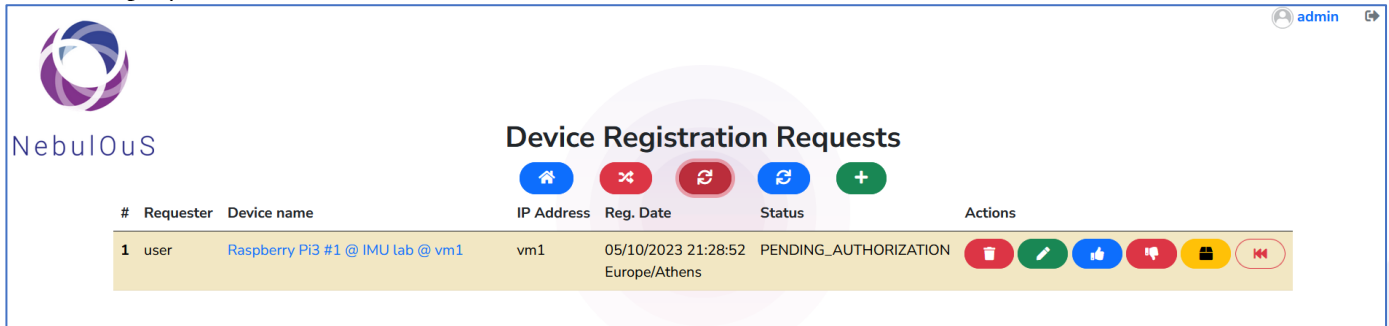


Figure 15: Admin view of open registration requests awaiting authorization

Resource Manager will periodically process the authorized requests and instruct the appropriate component to carry out the onboarding process. During onboarding the request status is ONBOARDING_REQUESTED.

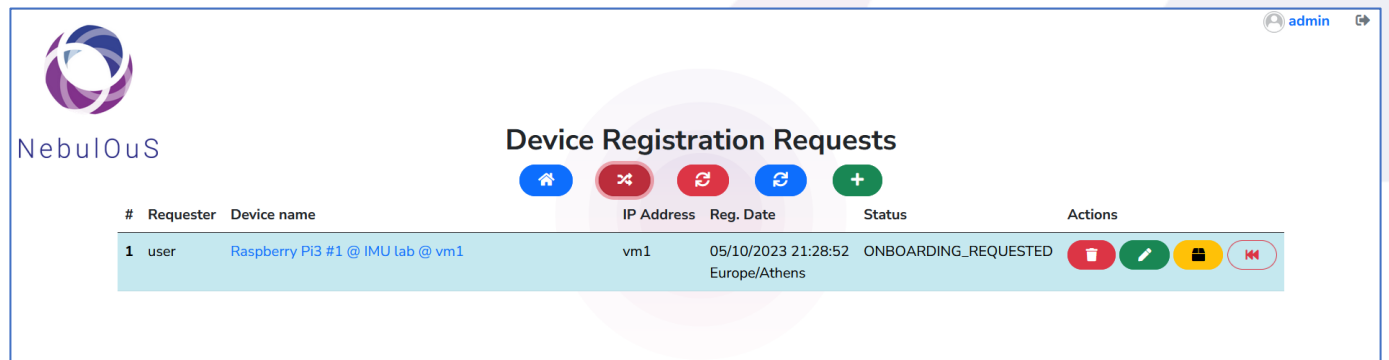


Figure 16: List of open user's registration requests (device is being onboarded)

After successful onboarding completion the request status changes to SUCCESS and the onboarded device will be listed in the list of active devices. Device owner can view only the devices he/she has registered (and had onboarded), but the administrator can view all of them.

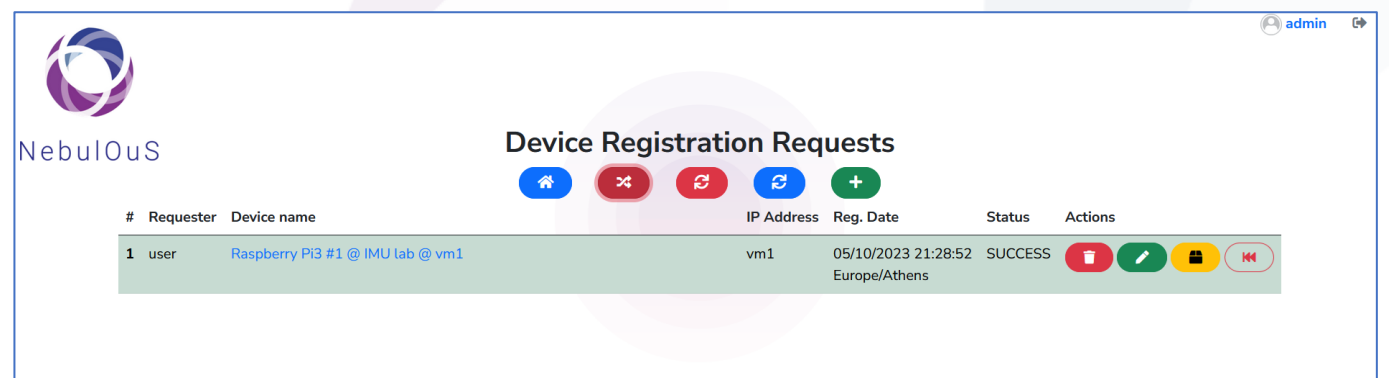


Figure 17: List of open user's registration requests (successful device onboarding)

Eventually, the Resource Manager can periodically archive old requests that have completed either successfully or with an error (data collection error, onboarding error, authorization error or rejection).

6. Semantic Modelling

As explained in the Introduction, NebulOuS provides the following ontological models: 1) the *asset model*, for describing common traits encountered in infrastructural CC services, including compute and storage capacities, network connectivity, geolocation, and price; 2) the *application component QoS requirements model* that is populated with the information described in the metric model outlined above. These models are elaborated in Sections 6.1 and 6.2.

6.1 Asset Modelling

The asset model provides the basis for determining how an application component deployment is to be realised across a pool of available resources given a set of user-expressed preferences, and providing that the application component's QoS are satisfiable. The asset model forms the basis of NebulOuS's *optimisation model* that describes the constraints and the objectives according to which application components are managed throughout their lifecycles (see Section 4). This includes *optimised* application component placement that considers the current capacities and capabilities of a pool of available CC nodes, the component's QoS requirements, as well as any user-expressed preferences regarding the consumption of the component.

Before presenting NebulOuS' asset model, we outline related work on semantic modelling in the IoT, as well as on semantic modelling of IaaS offerings.

6.1.1 IoT Ontologies

Several ontologies targeting interoperability in the IoT have been proposed. The "Semantic Sensor Networks" (SSN) ontology [14], [15] is a comprehensive W3C recommendation for providing a formal representation of sensor properties (sensing modalities, units, and ranges of measurement), actuators, the types of phenomena – or *features of interest*– being observed or affected by actuations, and the procedures involved in realising observations and actuations. SSN incorporates a lightweight, self-contained, core ontology called SOSA (Sensor, Observation, Sample, and Actuator). Through their different, albeit complementary, scopes and degrees of axiomatization, SSN and SOSA are together able to provide interoperability across a wide gamut of applications and use cases ranging from satellite imagery to social sensing and citizen science. Both SSN and SOSA are based on W3C's Resource Description Framework (RDF) and are thus extensible and reusable. In [16] IoT-Lite, a lightweight semantic model that includes the least number of concepts required for classifying IoT data, is defined as an instantiation of SSN⁴². IoT-Lite is not intended to be a fully-fledged ontology but a lightweight core extensible with application-specific semantic models.

The Smart Applications Reference (SAREF) [17] ontology provides a common vocabulary for describing the functionalities, features, and services of smart appliances, as well as the communication and data exchange protocols that these devices use. SAREF is intended to enable interoperability and support the development of smart home and building automation applications; to this end, it provides a modular representation of the service that an appliance provides in terms of its functions and the actual commands that invoke these functions. Notably, SAREF is narrower-in-scope than SSN and based on a more domain-specific nexus of interrelated concepts.

The IoT-A ontology [18] is part of the wider IoT Architectural Reference Model (ARM). It provides a formal representation of the key components and their relationships in an IoT architecture. The ontology is modular comprising different facets or "models": the domain model that includes fundamental concepts such as services and virtual entities, the entity model representing digital twins, the resource model carrying device-specific information, the service description model that describes the services offered by an IoT device, the event model

⁴² An ontology is considered to be an instantiation of another "parent" ontology when it builds upon or specializes the concepts and relationships defined in the parent ontology to create a more specific or domain-specific representation. This relationship is often described as a hierarchy, where the parent ontology is more general, and the child ontology is more specific.

describing the events and changes that a device may engage in, the functional model that encodes lower-level concerns such as protocols and device management, as well as concepts for describing data flows and intercomponent interactions.

The NGSILD ontology [19] is a formal representation of the concepts and relationships of the NGSILD data model that aims at allowing the exchange of data across IoT devices and systems. It includes the usual prose for the representation of IoT entities and the properties thereof, as well as a set of interrelated concepts for representing and sharing contextual awareness. It fails to support actuation environments.

In [20], an ontology is proposed for enabling the development of ‘generic’ IoT applications that are agnostic of the underlying sensing or actuating devices that they interact with. The ontology enables seamless device-to-application communication by abstracting away from vendor-specific device details through the annotation and classification of the data streams that these devices generate/accept. The ontology is based on an old version of SSN (prior to the integration of SOSA) extended with the authors’ own ontological actuation model.

In a similar vein, the SEMIC ontology [21] aspires to introduce an interoperability layer that bridges the world of IoT applications with the realm of sensing and actuating devices. SEMIC extends SSN through a nexus of interrelated concepts that enables the modelling of virtual (software) sensors and their interrelations with underlying physical sensors, and of virtual observations. Virtual observations are aggregations of physical observations that collectively provide the kind of higher-level information sought by IoT applications (e.g., deriving room occupancy information based on data from cameras and Wi-Fi access points).

IoTMA (Internet of Things Model and Analytics) [22] is another SSN-based ontology that provides a common vocabulary for describing the capabilities and properties of IoT devices. IoTMA emphasises formalising context awareness, i.e., including concepts for characterising observations gathered in a particular IoT context, and determining/prioritising any future actuations in that context based on that awareness.

The above ontologies are primarily designed as generic frameworks for capturing the delivery and consumption of heterogeneous sensor data, and the actuation of IoT devices. They focus on formally representing sensors, actuators, observations, and the phenomena being observed. They are, however, unsuitable for describing CC resources for they fail to incorporate concepts for modelling common traits such as compute and storage capacity, data transfer capability, geographical location, and price. Such concepts are the main focus of another class of ontologies, henceforth referred to as IaaS ontologies, that primarily aim at generically describing infrastructural cloud offerings. A brief overview of such ontologies is in order.

6.1.2 IaaS Ontologies

Several ontologies targeting interoperability across IaaS offerings have been defined. In [23], a series of ontologies are proposed, including one for describing compute instances (virtual machine characteristics), one for describing pricing schemes, one for describing regions and availability zones, and one for describing SLAs. A main drawback of these ontologies is that they typically associate domains and ranges of object properties through global scope constraints (`rdfs:domain` and `rdfs:range`): this is overly restrictive and can lead to unintended inferences [24].

In [25], the mOSAIC ontology is proposed for cloud service discovery and composition. It provides a platform and set of APIs for resolving interoperability issues in federated clouds. Its main “drawback” is that it predates the development of major standard domain ontologies such as schema.org, QUDT, SSN, and Wikidata, thus failing to rely on –and link to– them. Moreover, it seems not to be maintained anymore.

In [26], the Cloud Description Ontology is presented for facilitating cloud service brokerage at the IaaS, PaaS and SaaS levels. The ontology features a rather simplistic price model that is insufficient for modelling real world scenarios. It is unavailable online and seems not to be maintained anymore.

In [27], an OWL ontology for generically describing the lifecycle of cloud services is proposed. The ontology provides concepts and relations for modelling generic processes such as (cloud) service discovery, negotiation, composition, and consumption. It does not, however, provide any concepts and relations for modelling lower-level technical service specifications such as compute and storage capacity, network connectivity, geographical

location, and price; for such specifications it relies instead on (now outdated) external ontologies such as DReggie [28].

In [24], the Cloud Computing Ontology (CoCoOn) [24] is proposed for semantically describing cloud service offerings. CoCoOn reuses existing established domain ontologies including SSN [15], schema.org⁴³, and QUDT⁴⁴ (Quantities, Units, Dimensions and dataTypes). This brings about several advantages:

- Interoperability. Reusing terms from different ontologies enhances interoperability between different systems and datasets for it facilitates the unambiguous exchange and integration of information.
- Standardization. Reusing terms from established ontologies that typically represent industry standards or widely accepted vocabularies helps avoiding ambiguities and reducing redundancies (duplicate terms), whilst it increases credibility and understandability, hence acceptance, among domain experts.
- Evolution and maintenance. Reusing concepts and properties from established ontologies facilitates evolution and updates over time (regarding at least the reused concepts and properties).

Moreover, CoCoOn adheres to the principle of *minimal commitment* [29] leading to a more flexible and extensible model [24]. This is achieved by defining object property domains and ranges through guarded restrictions (i.e., through the property `owl:someValuesFrom`) and cardinality constraints (e.g., `owl:qualifiedCardinality` and `owl:maxQualifiedCardinality`), instead of the usual rigid global scope constraints (`rdfs:domain` and `rdfs:range`).

Due to the above advantages, we have decided to base our ontological asset model for describing infrastructural CC service offerings on CoCoOn. More specifically, we have decided to reuse CoCoOn's concepts and properties for providing a *schema* against which queries regarding QoS capabilities of CC resources can be executed. Notably, due to CoCoOn's reliance on SSN, this schema is easily extensible to serve queries that incorporate, in addition to QoS requirements, requirements on sensors/actuators (e.g., discover all CC resources that have certain QoS characteristics and are in proximity to sensors with a certain specification). This clearly leads to a more holistic approach to CC resource discovery that can take into account a multitude of requirements. Section 5.1.3 provides a summary of CoCoOn's main concepts and properties with emphasis on terms adopted and reused in our model. A fuller account of CoCoOn can be found in [30].

6.1.3 CoCoOn

Figure 18 provides an overview of CoCoOn's hierarchically structured classes⁴⁵. `cocoon:CloudService` is the main class hosting vocabularies for describing features and attributes of cloud service offerings at three different levels: IaaS, PaaS, and SaaS. Here we focus on IaaS. IaaS services are classified as compute, storage, and network (modelled, respectively, through the classes `cocoon:ComputeService`, `cocoon:StorageService`, and `cocoon:NetworkService`).

Compute Service

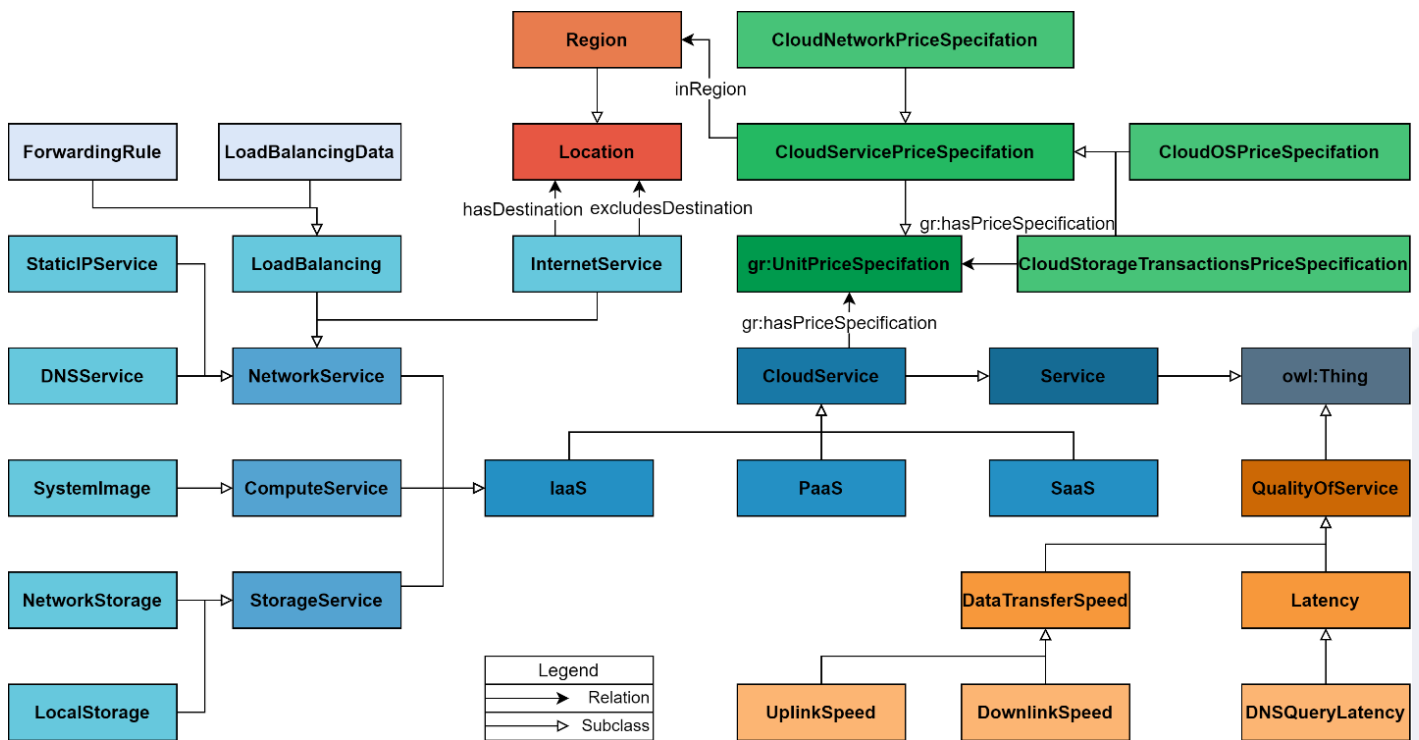
The following Virtual Machine (VM) attributes are modelled through data properties of `cocoon:ComputeService`:

- Number of cores available to a Virtual Machine (VM) (`cocoon:numberOfCores`).
- CPU performance power (`cocoon:hasCPUcapacity`).
- RAM size available to a VM (`cocoon:hasMemory`).

⁴³ <https://schema.org/>

⁴⁴ <https://www.qudt.org/>

⁴⁵ Undecorated lines represent subclassing relations.



The local storage available to a VM is modelled through the object property `cocoon:hasStorage` that maps a compute service instance to an instance of the class `cocoon:LocalStorage` (see below); the maximum number of disks and storage capacity assignable to a VM (if any) are specified through the data properties `cocoon:hasMaxNumberOfDisks` and `cocoon:hasMaxStorageSize` respectively. Turning now to pricing, the class `cocoon:ComputeService` inherits from its parent `cocoon:CloudService` class the object property `gr:hasPriceSpecification` which maps a compute service instance to a pricing specification from the class `gr:UnitPriceSpecification` (of the GoodRelations ontology⁴⁶). More details on CoCoOn’s price modelling are provided later. Listing 18 provides an example of a compute service instance specification⁴⁷. Note the use of the class `schema:TypeAndQuantityNode` to describe integer and decimal values that are associated with units of measurement as part of data properties (e.g., “a compute service instance has 86.4GB of memory”).

⁴⁶ GoodRelations is part of Schema.org ontology.

⁴⁷ The data properties `cocoon:inRegion` and `cocoon:hasProvider` are covered later.

```

@prefix schema: <https://schema.org/> .
@prefix unit: <http://qudt.org/vocab/unit#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix cocoon: <https://w3id.org/cocoon/v1.0.1#> .
@base <https://w3id.org/cocoon/data/v1.0.1/> .
<2019-02-12/ComputeService/Gcloud/CP-COMPUTEENGINE-VMIMAGE-N1-HIGHCPU-96-PREEMPTIBLE>
a cocoon:ComputeService ;
rdfs:label "CP-COMPUTEENGINE-VMIMAGE-N1-HIGHCPU-96-PREEMPTIBLE" ;
gr:hasPriceSpecification [ a cocoon:CloudServicePriceSpecification ;
    gr:hasCurrency "USD" ;
    gr:hasCurrencyValue 0.72 ;
    gr:hasUnitOfMeasurement unit:Hour ;
    cocoon:inRegion <Region/Gcloud/us-east1>
] ;
cocoon:hasMemory [ a schema:TypeAndQuantityNode ;
    schema:amountOfThisGood 86.4 ;
    schema:unitCode cocoon:GB
] ;
cocoon:hasProvider cocoon:Gcloud ;
cocoon:numberOfCores "96"^^xsd:decimal ;
schema:dateModified "2019-02-12"^^xsd:date .

```

Listing 18: Example compute service specification

Storage Service

Two kinds of storage service are defined: `cocoon:LocalStorage` and `cocoon:NetworkStorage`. For either kind, size is defined through the data property `cocoon:hasStorageSize`, whereas the amount of input/output operations per second (IOPS), and the throughput, are specified through the data properties `cocoon:hasStorageIOMax` and `cocoon:hasStorageThroughputMax` respectively. Pricing specifications are associated with storage service instances in the same way as with compute service instances. Listing 19 provides an example of a storage service specification.

```

@base <https://w3id.org/cocoon/data/v1.0.1/> .
<2019-03-07/NetworkStorage/Azure/premiumssd-p30>
a cocoon:NetworkStorage ;
rdfs:label "premiumssd-p30" ;
gr:hasPriceSpecification [ a gr:CloudServicePriceSpecification ;
    gr:hasCurrency "USD" ;
    gr:hasCurrencyValue 0.13200195133686066 ;
    gr:hasUnitOfMeasurement cocoon:GBPerMonth ;
    cocoon:inRegion <Region/Azure/australia-east>
] ;
cocoon:hasProvider cocoon:Azure ;
cocoon:hasStorageIOMax [ a schema:TypeAndQuantityNode ;
    schema:amountOfThisGood "5000"^^xsd:nonNegativeInteger ;
    schema:unitCode cocoon:IOPS
] ;
cocoon:hasStorageSize [ a schema:TypeAndQuantityNode ;
    schema:amountOfThisGood "1024"^^xsd:nonNegativeInteger ;
    schema:unitCode cocoon:GB
] ;
cocoon:hasStorageThroughputMax [ a schema:TypeAndQuantityNode ;
    schema:amountOfThisGood "200"^^xsd:nonNegativeInteger ;
    schema:unitCode unit:MegabitsPerSecond
] .

```

Listing 19: Example storage service specification

Network Service

Several kinds of network service are defined: `cocoon:InternetService`, `cocoon:LoadBalancing`, `cocoon:StaticIPService`, and `cocoon:DNSService`.

- `cocoon:InternetService`. Uses the object property `cocoon:hasDirection` to indicate traffic direction by mapping to the class `cocoon:TrafficDirection` which is partitioned by the singletons `cocoon:Egress` and `cocoon:Ingress`. The object properties `cocoon:hasDestination` and `cocoon:excludesDestination` are used to specify destination ranges by mapping to the class `cocoon:Location` (see later).
- `cocoon:LoadBalancing`. A load balancing service is modelled in terms of load balancing data, modelled as instances of the subclass `cocoon:LoadBalancingData`, and forwarding rules, modelled as instances of the subclass `cocoon:ForwardingRule`. Load balancing data are associated with a direction, modelled through the object property `cocoon:hasDirection`, and with a pricing specification; the latter association is achieved in the same way as with compute service instances.

Static IP and DNS services shall not further concern here for they do not directly relate to our work. Listing 21 provides an example of a load balancing service specification.

```
@base <https://w3id.org/cocoon/data/v1.0.1/2019-02-12/> .
<LoadBalancingData/Gcloud>
a cocoon:LoadBalancingData ;
gr:hasPriceSpecification [ a gr:CloudServicePriceSpecification ;
  gr:hasCurrency "USD" ;
  gr:hasCurrencyValue 0.008 ;
  gr:hasUnitOfMeasurement cocoon:GB ;
  cocoon:inRegion <Region/Gcloud/us>
] ;
cocoon:hasDirection cocoon:Ingress ;
cocoon:hasProvider cocoon:Gcloud ;
schema:dateModified "2019-02-12"^^xsd:date .
```

Price Modelling

The class `cocoon:CloudServicePriceSpecification` is defined as a subclass extension of the class `gr:UnitPriceSpecification`. The regional dimension of service pricing is addressed through the object property `cocoon:inRegion` which maps a price specification instance to an instance of the class `cocoon:Region` (see later). Disjoint specialisation subclasses are defined to handle different kinds of pricing specification: VM pricing (`cocoon:CloudOSPriceSpecification`), storage transactions pricing (`cocoon:CloudStorageTransactionsPriceSpecification`), and network services pricing (`cocoon:CloudNetworkPriceSpecification`). The reason for introducing these specialisation subclasses is to accommodate, through appropriate object and data properties, the different requirements of each kind of pricing specification.

- VM pricing. `cocoon:CloudOSPriceSpecification` features three main properties: `cocoon:chargedPerCore` that specifies the price charged per CPU core; `cocoon:forCoresMoreThan` that specifies the price charged for machines with more than the specified number of cores; `cocoon:forCoresLessEqual` that specifies the price charged for machines with less than the specified number of cores. Listing 21 provides an example of a VM pricing specification.
- Storage transactions pricing. No additional object or data properties are defined for this class. Listing 22 provides an example of a storage pricing specification.
- Network service pricing. The data properties `cocoon:forUsageLessEqual` and `cocoon:forUsageMoreThan` are used to specify upper and lower usage limits for network pricing schemes (e.g. the price for 0-1TB of egress Internet traffic, the price for 1-10TB of egress Internet traffic, and the price for 10+TB of egress Internet traffic). Special rates that apply e.g. to egress traffic between zones in the same region may be modelled through the data property `cocoon:specialRateType`.


```
@base <https://w3id.org/cocoon/data/v1.0.1/2019-02-12/> .
<SystemImage/Gcloud/suse-sap>
a cocoon:SystemImage ;
rdfs:label "suse-sap" ;
gr:hasPriceSpecification [ a cocoon:CloudOSPriceSpecification ;
  gr:hasCurrency "USD" ;
  gr:hasCurrencyValue 0.41 ;
  cocoon:chargedPerCore false ;
  cocoon:forCoresMoreThan "4"^^xsd:decimal
] ;
gr:hasPriceSpecification [ a cocoon:CloudOSPriceSpecification ;
  gr:hasCurrency "USD" ;
  gr:hasCurrencyValue 0.34 ;
  cocoon:chargedPerCore false ;
  cocoon:forCoresLessEqual "4"^^xsd:decimal ;
  cocoon:forCoresMoreThan "2"^^xsd:decimal
] ;
gr:hasPriceSpecification [ a cocoon:CloudOSPriceSpecification ;
  gr:hasCurrency "USD" ;
  gr:hasCurrencyValue 0.17 ;
  cocoon:chargedPerCore false ;
  cocoon:forCoresLessEqual "2"^^xsd:decimal
] .
```

Listing 21: Example pricing specification

QoS modelling

The quality with which a service is being delivered depends on the capabilities of the infrastructure allocated to it (i.e., compute and storage capacity, and network bandwidth), as well as on the actual data transfer speed and latency⁴⁸. As already seen, infrastructural capabilities are modelled through the classes `cocoon:ComputeService`, `cocoon:StorageService`, and `cocoon:NetworkService`. Data transfer speed and latency are defined through the class `cocoon:QualityOfService`, specifically through the subclasses `cocoon:DataTransferSpeed` and `cocoon:DNSQueryLatency` respectively. `cocoon:DataTransferSpeed` is further partitioned by `cocoon:DownlinkSpeed` and `cocoon:UplinkSpeed`. Listing 23; **Error! No se encuentra el origen de la referencia.** provides an example of a downlink speed specification for a specific data size, which is defined as equivalent to the class `ssn-system:SystemProperty`.

```
@base <https://w3id.org/cocoon/data/v1.0.1/> .
<2019-03-07/CloudStorageTransactionsPriceSpecification/Azure/managed_disk/transactions-ssd>
a cocoon:CloudStorageTransactionsPriceSpecification ;
rdfs:label "transactions-ssd" ;
gr:hasPriceSpecification [ a gr:CloudServicePriceSpecification ;
  gr:hasCurrency "USD" ;
  gr:hasCurrencyValue 0.0000002 ;
  cocoon:inRegion <Region/Azure/brazil-south>
] .
```

Listing 22: Example storage pricing specification

QoS measurement

QoS measurements are grouped under the class `cocoon:Measurement` which is defined as equivalent to the class `sosa:Observation`, enabling the use of `sosa:hasFeatureOfInterest` to specify the particular feature being measured

⁴⁸ Latency refers here to round trip time. It is affected by several uncontrollable factors including network congestion, routing efficiency, network infrastructure quality. The same as the ones affecting data transfer rate.

each time. The devices that are used for measuring QoS are described through the class `cocoon:Device` which is a subclass of `sosa:Sensor`.

```
@base <https://w3id.org/cocoon/data/v1.0.1/> .
<256-KB> a schema:TypeAndQuantityNode;
  schema:amountOfThisGood "256"^^xsd:integer;
  schema:unitText "KB";
  schema:unitCode "2P".

<QualityOfService/DownlinkSpeed-256-10240-KB> a cocoon:DownlinkSpeed;
  cocoon:transferredFileSizeMin <256-KB>;
  cocoon:transferredFileSizeMax <10240-KB>.
```

Listing 23: Downlink speed specification

Location and region

The class `cocoon:Location` represents any kind of geographical location. In contrast, its subclass `cocoon:Region` is a specialisation class used to represent cloud regions. Regions are mapped to their geographical locations through the `cocoon:inPhysicalLocation` and `cocoon:inJurisdiction` object properties depending on whether locations are known exactly or approximately respectively. A region is typically mapped to a single physical location. The continent to which a region belongs is specified through the `cocoon:continent` data property, whereas a region's provider is specified through the data property `cocoon:hasProvider` (maps to an `rdfs:label` describing a provider).

6.1.4 The NebulOuS Approach

Infrastructural CC service offerings are characterised by the same traits as IaaS cloud offerings: compute and storage capacity, data transmission capability, geographical location, and price. They may thus be modelled in terms of the same concepts and properties as IaaS cloud offerings. In NebulOuS, such modelling is based on CoCoOn. More specifically, in our ontological model a CC service takes the form of a contextualised cloud service i.e., one that is offered from outside the context of a cloud data centre. A CC service is thus represented as an instance of the class `cocoon:CloudService` that is associated with a price specification whose location – specified through the `cocoon:inRegion` property – lies outside the subclass `cocoon:Region`. Formally, we define the class `nebulous:FogService` in the Description Logic *SRQJC*⁴⁹ as⁵⁰:

$$\begin{aligned} \text{nebulous:FogService} &\equiv \\ & (= 1 \text{ gr. hasPriceSpecification. } ((= 1 \text{ cocoon:inRegion. cocoon:Location}) \\ & \quad \sqcap (< 1 \text{ cocoon:inRegion. cocoon:Region}))) \end{aligned}$$

where $((= 1 \text{ cocoon:inRegion. cocoon:Location}) \sqcap (< 1 \text{ cocoon:inRegion. cocoon:Region}))$ is the abstract class encompassing all those instances of the class `cocoon:CloudServicePriceSpecification` that are associated with locations that are not cloud data centre locations.

⁴⁹ OWL 2 is known to provide the expressiveness of the *SRQJC* Description Logic [31] which may thus be used to describe OWL 2 abstract classes such as `nebulous:FogService`.

⁵⁰ Note that, for any object property P and concept C , $(= 1 P.C)$ represents the abstract class that comprises all those individuals that feature exactly one association through P with an instance of C ; it is an abbreviation for the Description Logic notation $(\leq 1 P.C) \sqcap (\geq 1 P.C)$. $(\leq 1 P.C)$ represents the class of all individuals that have at most one association through P with an instance of C , and $(\geq 1 P.C)$ represents the class of all individuals that have at least one association through P with an instance of C .

6.2 QoS Requirements

We have hitherto focused on modelling capabilities and features of infrastructural CC services. We shall now focus on QoS requirements.

6.2.1 Service Quality Meta-Models

Several ontology-based meta-models have been proposed for describing QoS requirements. These include: WSAF-QoS [32], DAML-QoS [33], QoSOnt [34], WSMO-QoS [35], OWL-Q [36], [37], onQoS-QL [38], and PCM [39]. Nevertheless, only OWL-Q can be claimed to provide a rich metric model for describing QoS requirements in NebulOUS. We assess richness according to the following criteria based on modelling capabilities [40]:

- **Metric value types.** A metric value type defines a range of possible values, which are applicable in constraints related to that metric. When dealing with continuous numeric domains, which have an inherent order, it is sufficient to model only the highest and lowest values along with their numeric type (e.g., real, integer, etc.). If the domain is numeric but not continuous, it can be represented as a combination of multiple continuous domains. In practical applications, numeric domains are commonly employed for the majority of quality metrics. However, set and enumeration domains lack a predefined order, so the user must specify the ordering of elements within the domain.
- **Metric unit.** Metric values are typically measured using specific units, such as seconds for measuring execution time. However, it's insufficient to model just the unit name; we must also capture information on how to convert a value from one unit to another. To achieve this, units can be categorized into basic and derived units. Basic units are defined with a name and a concise abbreviation. Derived units, on the other hand, are created by multiplying a base unit by a specific float value, representing multiples of those base units. For instance, the unit for minutes can be derived by multiplying the unit for seconds by 1/60, while the unit for throughput quality is expressed by dividing the unit for "bytes" by the unit for "seconds". It is important to model these multiplying coefficients for derived units to ensure accurate conversions.
- **Metric measurement directive or function.** Quality metrics are categorized into resource and composite metrics. Resource or raw metrics are directly obtained from the service's instrumentation system by following measurement directives. These directives should include a URI that specifies how to retrieve the value of a managed resource, as well as information about the data type of the returned value. Additionally, the access model (either push or pull) must be defined to determine whether the party responsible for measurement will actively request the value or passively receive it when it becomes available. Furthermore, specific measurement directives may require a timeout attribute to specify the maximum waiting time for obtaining the measurement value. Composite metrics, on the other hand, are calculated by applying mathematical (often statistical) functions to other metrics. Therefore, the description of both the function used and the other metrics involved in the computation is essential. Additionally, a function model should be provided to allow users to select the appropriate mathematical function for each specific composite metric.
- **Metric schedule.** At least one of the following types of time windows should be defined for periodic or instantaneous calculations of new values for metrics: (a) calendar time window like week, month, and/or year; (b) sliding windows e.g., the last ten days; (c) expanding window or running total e.g., from this year's start until now.
- **Metric weight relative to its implicit domain and user preferences.** This weight can be used to calculate the rank of a service quality offer and indicates the impact that this metric has to the overall quality offered by a service.
- **Aggregation of the values of a composite service's metric.** It is imperative to provide a formal description of how the value of a metric for a complex service can be derived from the values of the corresponding

metrics of the individual services it comprises. This description is critical for automating the estimation of metric values for composite services.

We therefore opt for OWL-Q as basis for the NebulOuS ontological model for modelling QoS requirements.

6.2.2 OWL-Q

OWL-Q [36], [37] is an ontology designed to capture QoS requirements. It offers a comprehensive set of concepts (represented as OWL 2 classes) and properties for semantically describing and constraining virtually any QoS attribute. Through its Q-SLA facet, OWL-Q provides adequate support for semantic SLA specification. To the best of knowledge, no other existing ontology provides such support. We have therefore decided to base our ontological model for describing QoS requirements on OWL-Q.

OWL-Q comprises a collection of *facets* serving as logical boundaries between conceptually different parts of the ontology (see Figure 19). Each facet is designed to address a particular aspect of QoS modelling. The *Specification* facet focuses on modelling QoS characteristics as *constraints* on service attributes and metrics; such characteristics may be defined by a service requester as part of a service request, or by a service provider as part of service offerings. The *Attribute* facet captures knowledge about the attributes constrained in quality specifications; service response time, availability, and network bandwidth are all examples of such attributes. The *Metric* facet encapsulates knowledge about the (statistical) formulae –if any– applied on the constrained attributes in quality specifications; average response time, minimum availability, and minimum bandwidth over a period are all examples of such metrics. Lastly, the *Unit* facet focuses on modelling units of measurement, and the *Value Type* facet specifies allowable types and value ranges for the attributes constrained in quality specifications. More details on facets can be found in the following paragraphs.

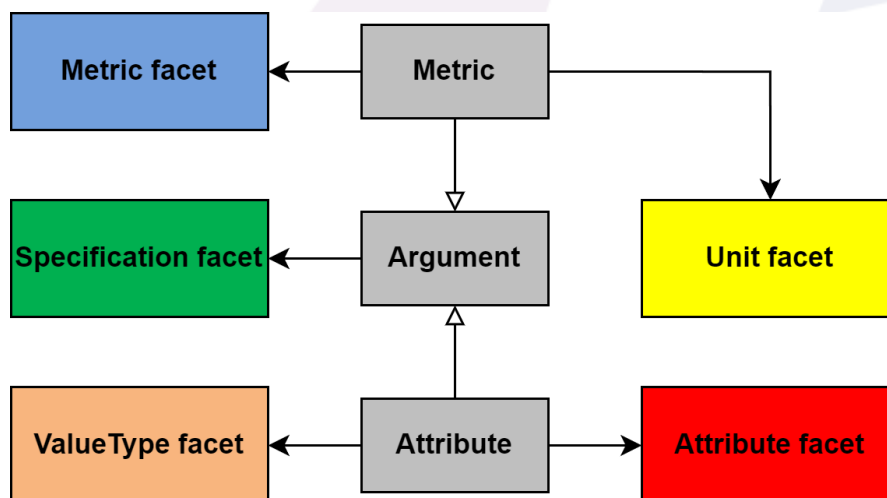


Figure 19: OWL-Q facets

In addition, OWL-Q includes top-level concepts that span several facets (depicted in grey colour in Figure 19). `owlq:Argument` represents constrained entities in QoS specifications. A constrained entity takes the form of either a service attribute (i.e., an instance of `owlq:Attribute`), in case a constraint directly concerns a service attribute (e.g., response time less than 1ms), or of a metric (i.e., an instance of `owlq:Metric`), in case a constraint is expressed in terms of a function on a service attribute (e.g., average response time over a period). An attribute is further specified by the *Attribute* and *Value Type* facets (see below). A metric is further specified by the *Metric* facet.

Specification facet

Represents aggregations of constraints on attributes and metrics. It revolves around the concept `owlq:Specification` which is partitioned by the classes `owlq:QoSProfile` and `owlq:QoSRequest` (see Figure 20). The

former represents quality characteristics set by a service provider for its offered services⁵¹. The latter represents quality characteristics that a service consumer requires. Quality characteristics take the form of constraints on service attributes or metrics. More specifically, the object properties `owlq:service` and `owlq:property` are used to collectively attach QoS attributes and metrics to a service specification, whereas the object property `owlq:containsConstraint` is used to attach a constraint. A constraint can be either simple or complex. A simple constraint directly compares its first argument (a QoS attribute or metric) with its second one (a threshold value); any unary, binary, or n-ary comparison operator may be used. A complex constraint is a logical combination of (other complex or simple) constraints to which it relates through the object property `owlq:constraint` (see Figure 20). A constraint may also be associated with a context that determines:

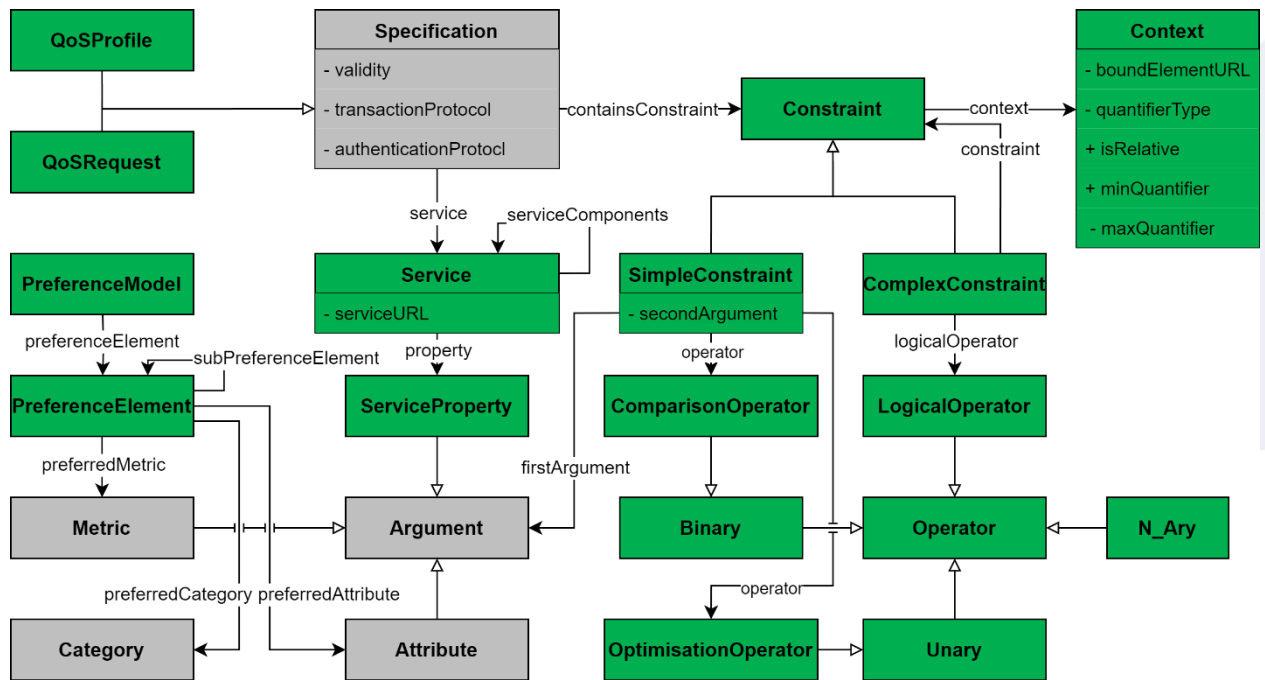


Figure 20: OWL-Q specification facet

- The specific service *part* (if any) to which the constraint applies.
- The number of service instances that must be accounted for determining whether the constraint is violated.

A specification may also be associated with a preference model (represented here as an instance of the class `owlq:PreferenceModel`). A preference model enables a requester to express their preferences on certain attributes and metrics over others through preference elements. A preference element is associated with a quality term (a service attribute or a metric – see Figure 20) through the object properties `owlq:preferredAttribute` and `owlq:preferredMetric`, and with a weight through the data property `owlq:weight`. A requester may also define *preference categories*, i.e., aggregations of QoS attributes or metrics. For example, a requester may define the ‘performance’ category as comprising the attributes *response time* and *throughput* with relative preference weights of 0.4 and 0.6 respectively. This assigns a normalised preference value of 0.42 to the overall ‘performance’ category. Note that the sum of preferences attached to the attributes of a particular category must equal to 1.

⁵¹ For instance, a service may be offered in three different qualities: ‘high’ with a response time of at most 10ms and an availability of at least 0.99999; ‘medium’ with a response time of at most 15ms and an availability of at least 0.9999; ‘low’ with a response time of at most 20ms and an availability of at least 0.999.

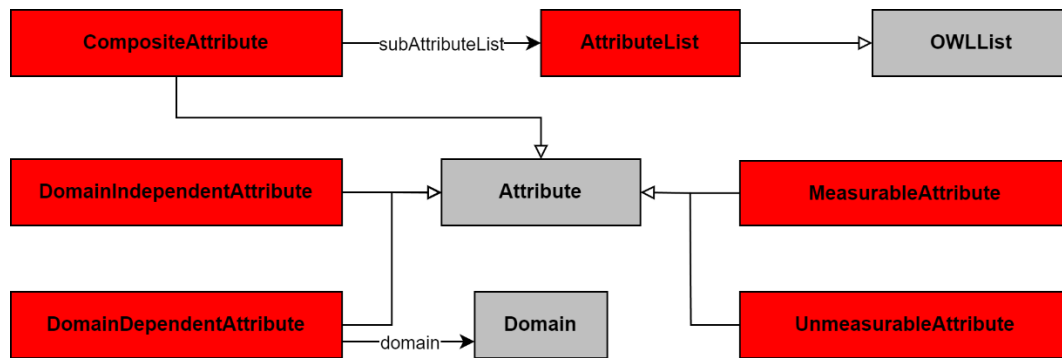


Figure 21: OWL-Q Attribute facet

Attribute facet

Any attribute constrained through a QoS specification takes the form of an instance of the class `owlq:Attribute` which is partitioned into (see Figure 21):

- `owlq:MeasurableAttribute`: attributes that can be measured through `owlq:Metric` (see below).
- `owlq:UnmeasurableAttribute`: an attribute that cannot be quantitatively measured e.g., a UX attribute.
- `owlq:DomainDependentAttribute`: an attribute that is only meaningful in certain domains; round trip time is an example of such an attribute since it is only meaningful in network performance measurements.
- `owlq:DomainIndependentAttribute`: an attribute that is meaningful across domains e.g., time.

In NebulOuS we focus on measurable attributes. An attribute may be composite comprising other attributes in which case it belongs to the class `owlq:CompositeAttribute` and is associated with an attribute list and a function that determines how elements in the list are combined (see Figure 21).

Metric facet

Any metric constrained through a QoS specification takes the form of an instance of the class `owlq:Metric` which is partitioned into the classes `owlq:RawMetric` and `owlq:CompositeMetric` (see Figure 22). Raw metrics (e.g., response time) are directly recorded through observation from the measurement system's instrumentation or from sensors. Composite metrics (e.g., average response time) are derived by applying a (statistical) formula on a list of arguments. Raw metrics may be related to a sensor (instance of `owlq:Sensor`) and a measurement directive (instance of `owlq:MeasurementDirective`). Sensors model the instrument—whether physical such as a thermometer or virtual such as a piece of software measuring latency—that makes the measurement; sensors may be associated with a configuration (instance of `owlq:Configuration`) that describes their installation, invocation, and stopping. Measurement directives define various parameters such as whether the measured value will get pulled or pushed by the sensor, or the type of measurement being made (count, downtime, execution time, status, etc).

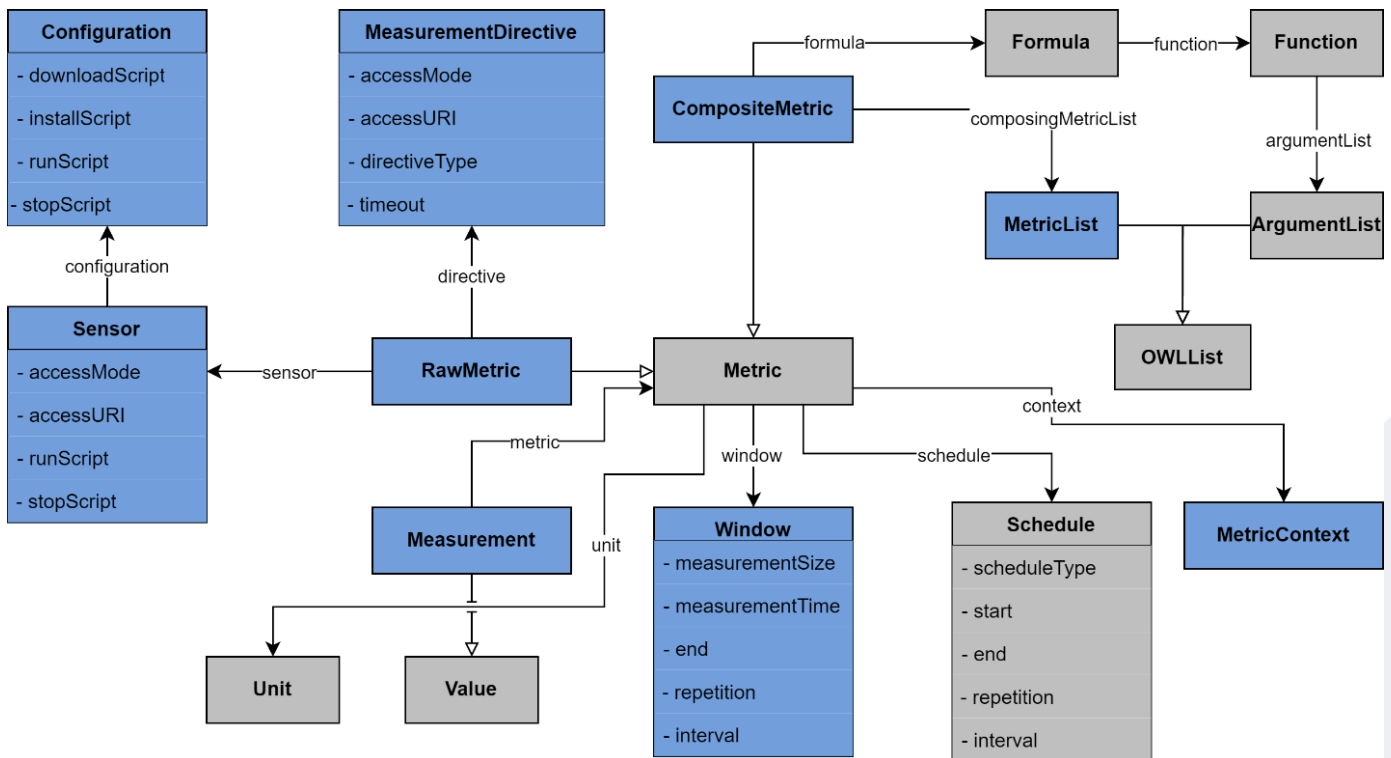


Figure 22: OWL-Q Metric facet

Composite metrics are related to a formula (instance of `owlq:Formula`) which represents the computation needed to produce a measurement. Each formula consists of a statistical function (e.g., mean, median, min, max, etc.) and a list of arguments that can be constants, other formulae, or other metrics. A composite metric may also be related to a `owlq:MetricList` that contains all other metrics that it comprises. Both raw and composite metrics may be related to a schedule (instance of `owlq:Schedule`) that defines their temporal dimensions (e.g., when the metric starts/stops being active), how many measurements should be taken, how often they should be taken, and the type of schedule used (fixed delay, fixed rate, single event). Raw and composite metrics may also be related to a window (instance of `owlq:Window`) specifying other properties such as measurement type and size, and measurement window type (sliding vs fixed).

Unit facet

Models units of measurement. Units can be classified into single, derived, and dimensionless. Derived units are computed from single ones through the application of mathematical operations, typically division and multiplication (e.g., bits per second). Both single and derived units may be associated with a dimension, represented as an instance of `owlq:QuantityKind`, and with a quantity, represented as an instance of `owlq:Quantity` (see Figure 10) For example, the unit “bits per second” is related to the speed “dimension” (an instance of `owlq:QuantityKind`), and to the network speed “quantity” (an instance of `owlq:Quantity`) respectively. Dimensionless units (e.g. number of times a threshold value has been exceeded) are represented as instances of the namesake class. Finally, the object property `owlq:multipleOf` is used to denote compatibility between units that are multiples of each other (e.g. bits, bytes, kilobytes, etc.).

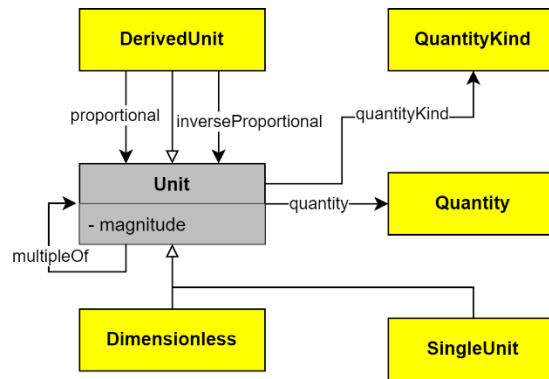


Figure 23: OWLQ Unit facet

Value Type facet

The value type facet focuses on representing allowable value types and ranges, enabling the *validation* of metric measurements. It comprises two main classes: `owlq:Value` and `owlq:ValueType` (see Figure 24). The former represents any kind of value that can be a component of a value type. It is further subdivided into specialised sub-classes mapping to widely used XSD data types, such as integers and doubles. Value types are distinguished into scalar value types and value lists. A scalar value type can be bounded or unbounded. Bounded value types are separated into ranges and unions of ranges. Ranges are characterised by two equivalently typed limits that may or may not be included in the range and directly map to a certain Value. Unions of ranges comprise non-overlapping ranges that contain the same kind of values (e.g., integers). Unbounded value types map to numerical types (Integers, Floats and Doubles) and Strings.

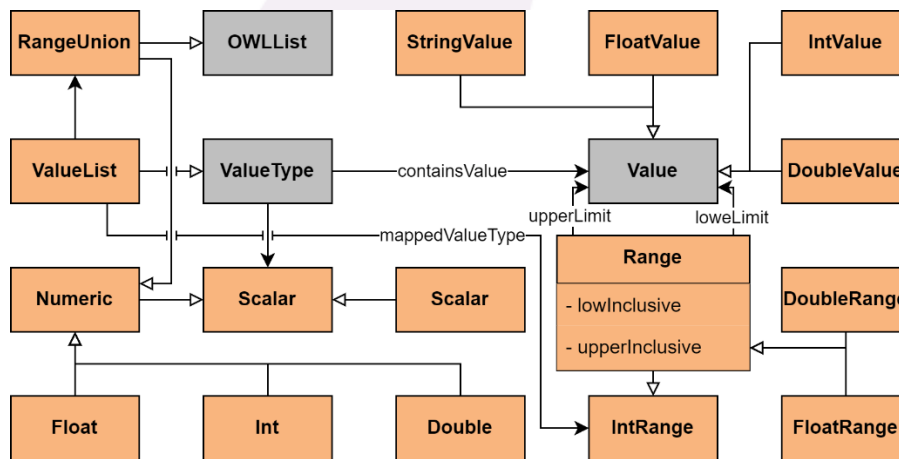


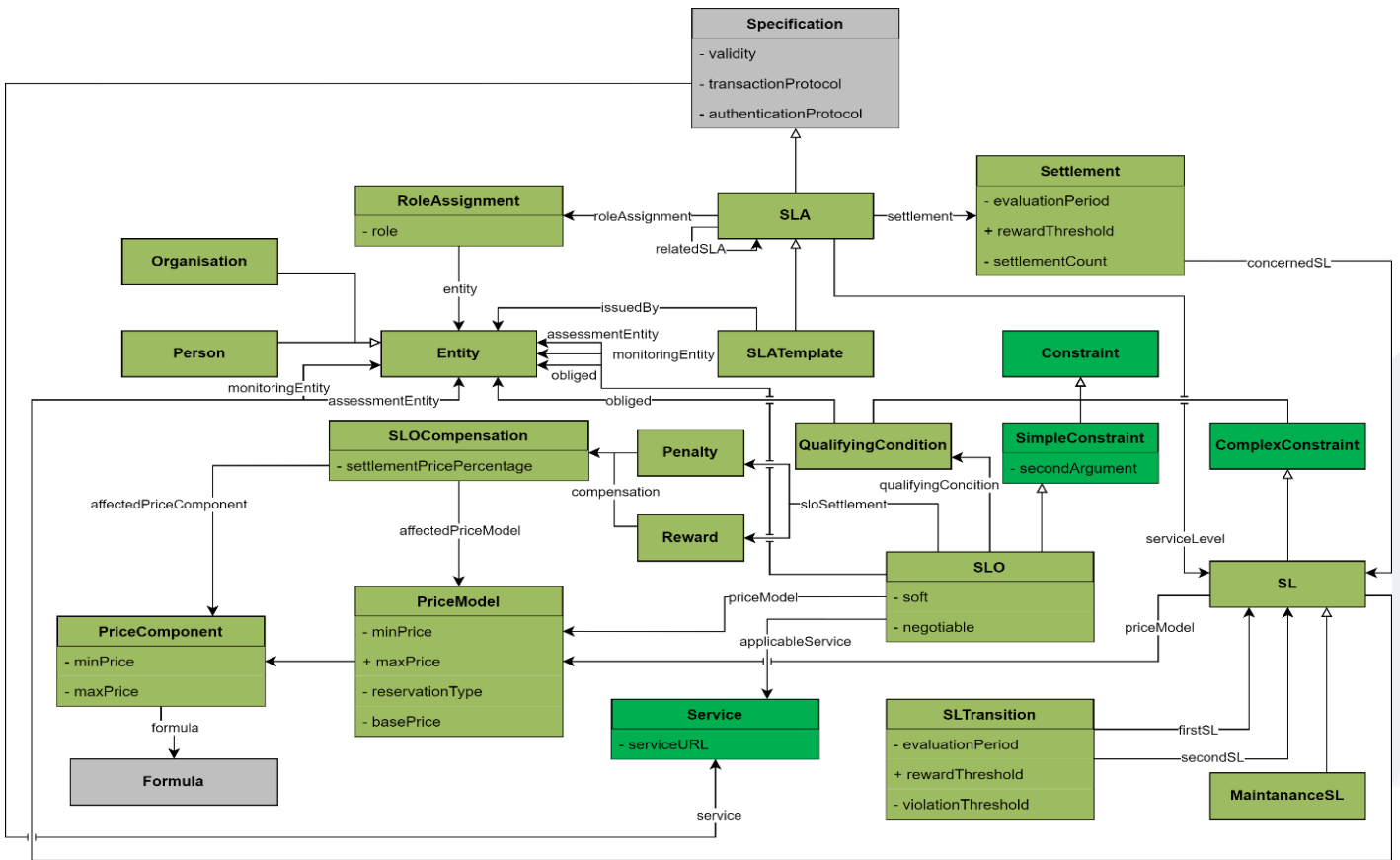
Figure 24: OWLQ Value Type facet

6.2.3 Q-SLA

An extended version of OWLQ, namely Q-SLA, has been proposed for semantically describing SLAs [37]. Q-SLA introduces an SLA facet as an extension of the Specification facet, reflecting the fact that SLAs are a kind of quality specification including, alongside constraints and user preferences, contractual information such as the entities bound by an SLA, validity periods, the various service levels offered, pricing, as well as compensation schemes in case of SLA violations.

More specifically, SLAs are represented as instances of the class `owlq:SLA` which is a subclass of `owlq:Specification` (see Figure 25). The entities⁵² bound by an SLA assume one of the roles “requester”,

⁵² Either legal entities or physical persons.



“provider”, or “third party”. Entity-role assignments take the form of instances of the class `owlq:RoleAssignment`. SLAs are associated with their entity-role assignments through the object property `owlq:roleAssignment`, and entity-role assignments are associated with entities through the data property `owlq:role` (Figure 25). SLA validity periods are determined through the data property `owlq:validity`.

The service levels (SLs) specified in an SLA determine the different performance behaviours that a service can exhibit; they are thus a kind of complex constraints [37]. SLs are represented as instances of the class `owlq:SL` – a subclass of `owlq:ComplexConstraint`. Any SL comprises at least one simple constraint termed service level objective (SLO). SLOs are modelled as instances of the class `owlq:SLO` which is a subclass of `owlq:SimpleConstraint`. An SL is associated with its constituent SLOs through the object property `owlq:constraint` (see Figure 20). An SLO is bound to a service or service component through the `owlq:applicableService` property (see Figure 20). An SLO is also associated with:

- The *entities* responsible for monitoring and assessing it, as well as with the entity obliged to guarantee it, through the data properties `owlq:monitoringEntity`, `owlq:assessmentEntity`, and `owlq:obliged` respectively.
- *Compensations*, through the object property `owlq:sloSettlement` (see Figure 25). Compensations are penalties or rewards that apply, respectively, when the SLO is violated or when service performance exceeds pre-set thresholds; they are represented as instances of the class `owlq:SLOCompensation`.
- A *qualifying condition* that must hold for the SLO to be assessable (and possibly compensable). Such a condition can refer to contextual restrictions at the requester side such as the number of concurrent incoming requests that can be served over a period. Qualifying conditions are a kind of constraint. They are modelled as instances of the class `owlq:QualifyingCondition` – a subclass of `owlq:Constraint`.

Lastly, an SLO may be soft and/or negotiable as determined by the namesake data properties (see Figure 25). SLOs are soft if their violation is deemed unimportant when matching provider-defined SLs with requester-defined SLs. In other words, SLOs are soft when their violation does not affect the match making process. SLOs

are negotiable when the value ranges that they enable may be shaped after negotiation between the requester and provider.

SLs are further associated with a price model that specifies the cost of consuming a service or service component (at the particular service level). Price models are represented as instances of the class `owlq:PriceModel`. Pricing details such as the minimum and maximum chargeable prices for a service, and its base price, are captured through the data properties `owlq:minPrice`, `owlq:maxPrice`, and `owlq:basePrice` respectively (see Figure 25). Base prices are the prices charged under normal service operation; minimum chargeable prices define the least cost for a service in case of SLO violation penalties (in other words, the maximum penalty that an SLO violation can incur); maximum chargeable prices define an upper cost boundary applicable when more resources are assigned to a service. Price models comprise *price components* modelled as instances of the class `owlq:PriceComponent`. Each price component focuses on a particular cost aspect. For instance, a price component may focus on the cost of consuming compute resources, while other components may focus on network resources and data exchange costs. The cost charged by a price component is calculated through a formula over relevant quality terms and attributes. A price model is also related to a reservation type stating if charging can be performed periodically, via advanced reservations, or on demand (spot pricing). Price model instances are attached to SLO compensations.

If the number of SLO violations is kept below a threshold, only SLO compensations may be paid. However, if the number of SLO violations over a period is high, it must be determined if the SLA is viable and whether it should be cancelled, renegotiated, or re-enforced (i.e., the service has to be re-executed). This is achieved through *settlements*. Settlements are activities that are associated with SLs and assess what has happened during service execution (i.e., with respect to the SL's SLOs), and which are each signatory party's responsibilities in case of SLO violations. Formally, settlements are instances of the namesake class (Figure 25). The data properties `owlq:settlementAction`, `owlq:evaluationPeriod`, and `owlq:settlementCount` are used, respectively, to determine a settlement's actions in case of SLO violations (cancellation, renegotiation, re-enforcement), the number of SLO violations that activates the settlement, and the service execution length over which this number must be observed. Settlements are associated with SLAs through the `owlq:settlement` object property (Figure 25) and are attached to SLs through the `owlq:concernedSL` property.

Q-SLA also includes the concept of SL transitions (formally represented by the namesake class in Figure 25) to enable movement between two SLs and hence between different performance levels. Transitions may be event-driven: triggered when too many SLO violations over a period occur, or when SLO expectations are exceeded beyond a certain threshold for a period. This is captured through the data properties `owlq:evaluationPeriod`, `owlq:rewardThreshold` and `owlq:violationThreshold` (see Figure 25). Transitions may also be chronologically driven, occurring at pre-set time points. Transitions are associated with their corresponding SLs - the transitioned from and the transitioned to SLs - via the object properties `owlq:firstSL` and `owlq:secondSL` respectively.

As an example, let us consider a face detection service with the following QoS attributes: response time, availability, and throughput. Each attribute is evaluated by measuring a pertinent raw metric, or by calculating a composite metric. Listing 24 demonstrates how an attribute is evaluated either by directly measuring (through a relevant sensor) a raw response time metric, or by calculating an average response time over a period.

```

### Measurable attribute denoting response time.
:RESPONSE_TIME rdf:type owl:NamedIndividual ,
    owl:MeasurableAttribute ;
    owl:measuredBy :AVERAGE_RESPONSE_TIME_METRIC , :RAW_RESPONSE_TIME_METRIC .

### Metric measuring the raw response time.
:RAW_RESPONSE_TIME_METRIC rdf:type owl:NamedIndividual ,
    owl:RawMetric ;
    owl:context :RAW_RESPONSE_TIME_METRIC_CONTEXT ;
    owl:directive :RAW_RESPONSE_TIME_METRIC_DIRECTIVE ;
    owl:schedule :RAW_RESPONSE_TIME_SCHEDULE ;
    owl:sensor :RAW_RESPONSE_TIME_SENSOR .

### Metric calculating the average response time.
:AVERAGE_RESPONSE_TIME_METRIC rdf:type owl:NamedIndividual ,
    owl:CompositeMetric ;
    owl:context :AVERAGE_RESPONSE_TIME_CONTEXT ;
    owl:formula :AVERAGE_RESPONSE_TIME_FORMULA ;
    owl:schedule :AVERAGE_RESPONSE_TIME_SCHEDULE .

```

Listing 24: Example of an attribute and its metrics

An SLA for this service specifies: the involved parties (`owlq:entity`) and their roles (`owlq:roleAssignment`), the concerned service (`owlq:service`), the available SLs (`owlq:serviceLevel`) alongside their SLOs (`owlq:constraint`), settlements in case of failure to meet SLOs (`owlq:so1Settlement`), validity periods (`owlq:validity`) (see Listing 25). SLs are specified in terms of the constraints that they impose in the form of SLOs (`owlq:constraint`), a pricing model (`owlq:priceModel`), and a transition specification to enable movement to different performance levels under certain conditions (`owlq:SLTransition`)(see Listing 26).

```

### Example of an SLA.
:SLA_AC rdf:type owl:NamedIndividual, owlq:SLA ;
    owlq:roleAssignment :PROVIDER, :REQUESTER ;
    owlq:service :FACE_DETECTION_SERVICE ;
    owlq:serviceLevel :LOW_SL, :NORMAL_SL ;
    owlq:settlement :SET_LOW ;
    owlq:validity "2024-05-30T09:00:00"^^xsd:dateTime .

```

Listing 25: Example of a simple SLA

```

### Example of a service level.
:LOW_SL rdf:type owl:NamedIndividual, owlq:SL ;
    owlq:constraint :LOW_AV, :LOW_THR, :LOW_RT ;
    owlq:priceModel :PM_LOW .

### Example of pricing model.
:PM_LOW rdf:type owl:NamedIndividual, owlq:PriceModel ;
    owlq:minPrice 600 ;
    owlq:reservationType "PER_MONTH" .

### Example of SL transition.
:NOR_TO_LOW rdf:type owl:NamedIndividual, owlq:SLTransition ;
    owlq:firstSL :NORMAL_SL ;
    owlq:secondSL :LOW_SL ;
    owlq:evaluationPeriod 0.5 ;
    owlq:violationThreshold 4 .

```

Listing 26: Example of a SL, its pricing model, and SL transition

Listing 27 provides an example of an SLO specification that includes a constraining operator (less or equal than), a metric as the SLO's first argument, and a threshold value as the SLO's second argument again which this metric is compared through the constraining operator. A settlement that defines what happens in case of

SLO violation (e.g., the price is reduced to only 5% of the originally agreed-upon price – see Listing 27) is also specified.

```

### Example of SLO.
:LOW_RT rdf:type owl:NamedIndividual, owl:SLO ;
        owl:firstArgument :AVERAGE_RESPONSE_TIME_METRIC ;
        owl:operator owl:LESS_EQUAL_THAN ;
        owl:sloSettlement :P1 ;
        owl:secondArgument 2 .

### Example of a penalty.
:P1 rdf:type owl:NamedIndividual, owl:Penalty ;
    owl:compensation :C1 .

### Example of an SLO compensation.
:C1 rdf:type owl:NamedIndividual, owl:SLOCompensation ;
    owl:settlementPricePercentage "0.05"^^xsd:double .

```

Listing 27: Example of an SLO, its penalty, and compensation

6.2.4 Metadata Schema

OWL-Q provides a comprehensive set of data properties for describing various aspects of the quality provided by a computing service. Albeit, OWL-Q was not designed to cater for the particular needs of resources in the cloud continuum. It is therefore likely that, during the course of NebulOuS, it will be extended with additional data properties that enable capturing a richer set of aspects related to application deployment across heterogeneous CC resources and multi-clouds. To this end, the Metadata Schema (MDS) will be utilised.

MDS provides a rich schema covering essential aspects related to application deployment and big data management in the cloud continuum. MDS aggregates several classes and properties that correspond to concepts used for describing DevOps requirements and constraints, and infrastructural service offerings in multi-cloud placement decisions. Its objective is to create the background modelling layer for any Domain Specific Language (DSL) that aspires to describe application deployments in multi-cloud and fog environments. It was used as the formal means for extending the CAMEL language [1] with appropriate concepts related to big data management, the placement optimisation of processing jobs and access control in multi-cloud environments. In a similar vein, it may be used for extending NebulOuS' CoCoOn– and OWL-Q –based ontologies if the project's use cases require it.

MDS was introduced as part of the Melodic⁵³ project for addressing the multi-clouds requirements and offerings description and it was extensively updated during the Morphemic⁵⁴ project for coping with additional kinds of resources like HPC and hardware accelerated ones, while supporting polymorphic adaptations (i.e., allowing the descriptions of different technical implementation forms for the relevant application components).

⁵³ <https://www.melodic.cloud/>

⁵⁴ <https://www.morphemic.cloud/>

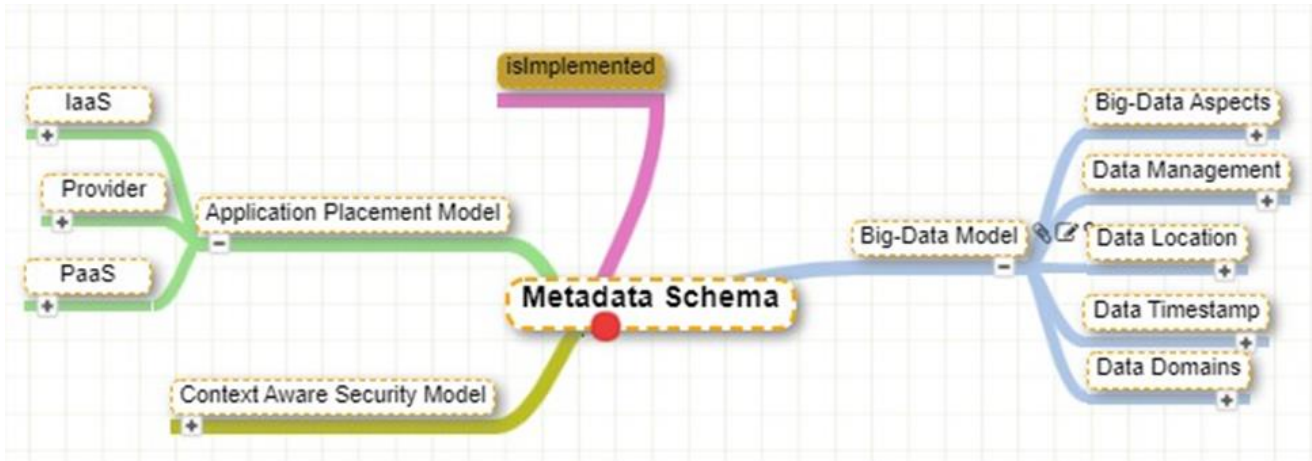


Figure 26: Metadata Schema overview

MDS comprises the Application Placement, Big Data and Context Aware Security models that group a number of classes and properties to be used for defining where a certain big data application should be placed; what are the unique characteristics of the data artefacts that need to be processed; and what are the contextual aspects that may be used for restricting the access to the sensitive data.

For example, one of the classes of the Application Placement sub-model of MDS is the Processing. This class involves any infrastructural feature bound to the processing capability of virtualised resources. One of its subclasses is the Accelerator class which refers to application-specific hardware designed or programmed to compute operations faster than a general-purpose computer processor. It involves the subclasses such as GPU, ASIC, FPGA and VPU for defining different accelerator types offered or required in a DSL description. The complete class diagram for the Processing domain can be seen in Fig. 10. MDS⁵⁵ was serialized in XMI⁵⁶ as an Ecore-based language encoding form to enable the re-use of its elements for annotating CAMEL models. A bird's eye view of the complete MDS taxonomy can be found here⁵⁷.

⁵⁵ <https://gitlab.ow2.org/melodic/camel/-/tree/rc3.1/metadata-schema/current>

⁵⁶ <http://www.omg.org/spec/XMI/>

⁵⁷ <https://melodic.cloud/UuTf-KRW.png>

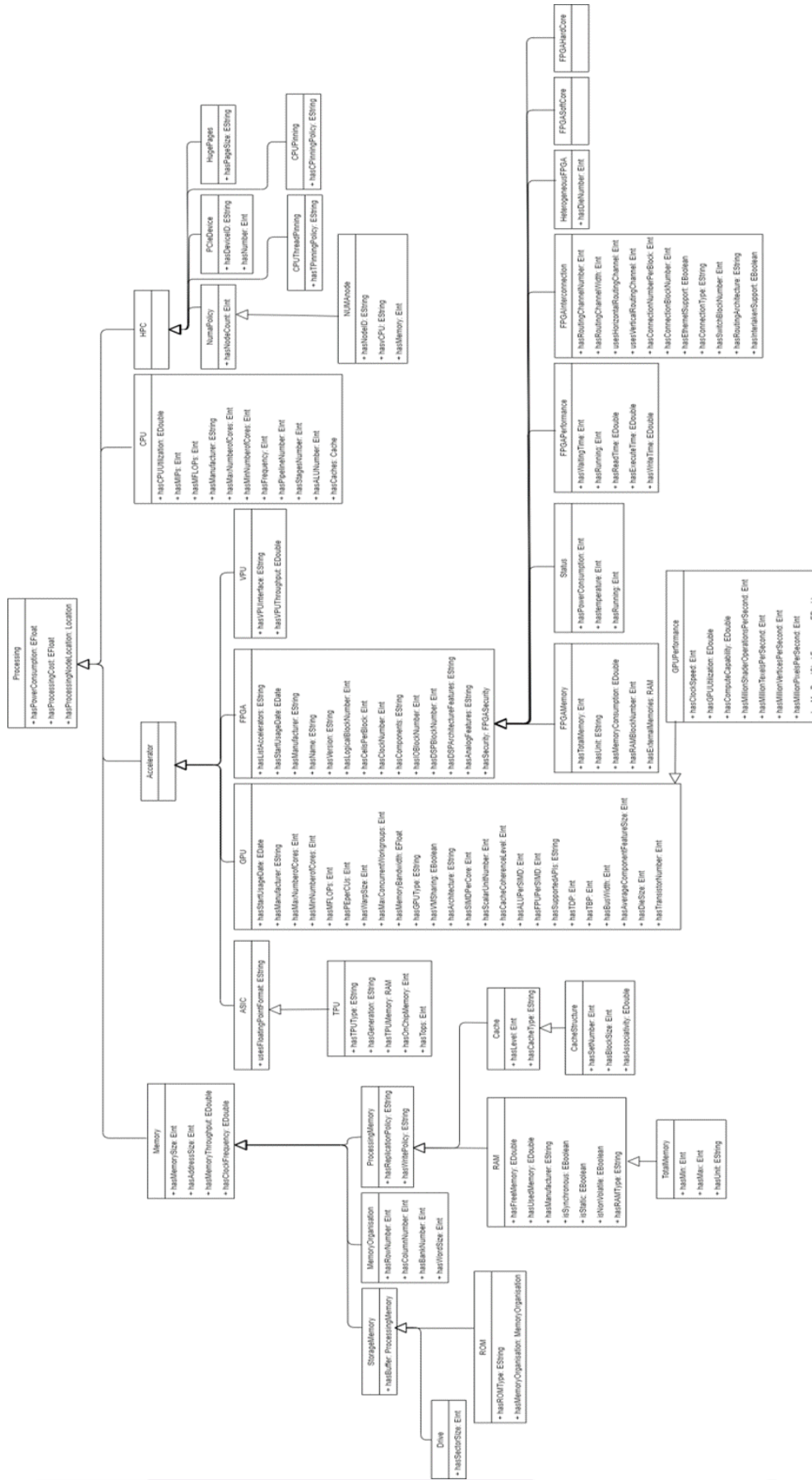


Figure 27: The UML class diagram for the Processing domain

7. Conclusions

This document provides a first account of NebulOuS' approach to resource discovery. It presents the three distinct, albeit interrelated, declarative models that underpin NebulOuS' resource discovery mechanism, namely:

- Application component compositions and deployments. A KubeVela-based model for describing application composition and deployment.
- QoS requirements attached to application components. A custom model (based on the metric model from CAMEL) for enabling the definition of custom metrics over arbitrary user-defined QoS attributes.
- Optimisation. An AMPL-based model for describing the constraints and the objectives according to which application components are managed throughout their lifecycles. This model is underpinned by an ontologically-described asset model that captures the capabilities and characteristics of a pool of CC resources across which an application component is to be hosted.

Moreover, this document presents an ontological model for capturing application workload QoS requirements that forms the basis for NebulOuS' quality mechanism. More specifically, by ontologically describing QoS requirements, we pave the way for a quality assurance mechanism that relies on *semantic reasoning* for assessing the correctness of these requirements by comparing them against a set of semantically captured *application consumption policies*.

Finally, this document provides an initial account of a prototype of the NebulOuS resource discovery mechanism. The fully-fledged NebulOuS resource discovery mechanism, alongside the final version of the models, will be reported in D2.3 (due in by M35).

References

- [1] 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.
- [2] C.-H. Hong and B. Varghese, ‘Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms’, *ACM Comput. Surv.*, vol. 52, no. 5, pp. 1–37, Sep. 2020, doi: 10.1145/3326066.
- [3] B. Varghese and R. Buyya, ‘Next Generation Cloud Computing: New Trends and Research Directions’, 2017, doi: 10.48550/ARXIV.1707.07452.
- [4] C.-H. Hong, K. Lee, M. Kang, and C. Yoo, ‘qCon: QoS-Aware Network Resource Management for Fog Computing’, *Sensors*, vol. 18, no. 10, p. 3444, Oct. 2018, doi: 10.3390/s18103444.
- [5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, ‘Edge Computing: Vision and Challenges’, *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.
- [6] B. Costa, J. Bachiega, L. R. De Carvalho, and A. P. F. Araujo, ‘Orchestration in Fog Computing: A Comprehensive Survey’, *ACM Comput. Surv.*, vol. 55, no. 2, pp. 1–34, Feb. 2023, doi: 10.1145/3486221.
- [7] B. Varghese, N. Wang, J. Li, and D. S. Nikolopoulos, ‘Edge-as-a-Service: Towards Distributed Cloud Architectures’, 2017, doi: 10.48550/ARXIV.1710.10090.
- [8] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwalder, ‘Incremental deployment and migration of geo-distributed situation awareness applications in the fog’, in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, Irvine California: ACM, Jun. 2016, pp. 258–269. doi: 10.1145/2933267.2933317.
- [9] A. Tsagkaropoulos, Y. Verginadis, M. Compastie, D. Apostolou, and G. Mentzas, ‘Extending TOSCA for Edge and Fog Deployment Support’, *Electronics*, vol. 10, no. 6, p. 737, Mar. 2021, doi: 10.3390/electronics10060737.
- [10] Y. Verginadis, I. Alshabani, G. Mentzas, and N. Stojanovic, ‘PrEstoCloud: Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing’, in *Proceedings of the 7th International Conference on Cloud Computing and Services Science*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 611–617. doi: 10.5220/0006359106110617.
- [11] M. Wurster, U. Breitenbucher, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov, ‘TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies’, in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*, Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2020, pp. 216–226. doi: 10.5220/0009794302160226.
- [12] G. Blair, N. Bencomo, and R. B. France, ‘Models@ run.time’, *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009, doi: 10.1109/MC.2009.326.
- [13] Robert Fourer, David M. Gay, and Brian W. Kernighan, *AMPL - A Modeling Language for Mathematical Programming*, Second edition. Duxbury Press, 2003. [Online]. Available: <https://ampl.com/wp-content/uploads/BOOK.pdf>
- [14] M. Compton *et al.*, ‘The SSN ontology of the W3C semantic sensor network incubator group’, *J. Web Semant.*, vol. 17, pp. 25–32, Dec. 2012, doi: 10.1016/j.websem.2012.05.003.
- [15] A. Haller *et al.*, ‘The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation’, *Semantic Web*, vol. 10, no. 1, pp. 9–32, Dec. 2018, doi: 10.3233/SW-180320.
- [16] M. Bermudez-Edo, T. Elsaleh, P. Barnaghi, and K. Taylor, ‘IoT-Lite: A Lightweight Semantic Model for the Internet of Things’, in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCCom/IoP/SmartWorld)*, Toulouse: IEEE, Jul. 2016, pp. 90–97. doi: 10.1109/UIC-ATC-ScalCom-CBDCCom-IoP-SmartWorld.2016.0035.
- [17] L. Daniele, R. Garcia-Castro, M. Lefrancois, and M. Poveda-Villalon, ‘ETSI TS 103 264’. ETSI, Feb. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/103200_103299/103264/03.01.01_60/ts_103264v030101p.pdf
- [18] M. Bauer, M. Boussard, N. Bui, and F. Carrez, ‘Internet of Things – Architecture IoT-A Deliverable D1.5 – Final architectural reference model for the IoT v3.0’, Technical report D1.5, Jul. 2013.
- [19] P. Gilles, *Guidelines for Modelling with NGSILD*. 2021.
- [20] K. Kotis and A. Katasonov, ‘An ontology for the automated deployment of applications in heterogeneous IoT environments’, [Online]. Available: https://www.semantic-web-journal.net/sites/default/files/swj247_0.pdf

- [21] R. Yus, G. Bouloukakis, S. Mehrotra, and N. Venkatasubramanian, 'The SEMIOTIC Ecosystem: A Semantic Bridge between IoT Devices and Smart Spaces', *ACM Trans. Internet Technol.*, vol. 22, no. 3, pp. 1–33, Aug. 2022, doi: 10.1145/3527241.
- [22] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy, 'Spotcheck: Designing a derivative iaas cloud on the spot market', presented at the Proceedings of the Tenth European Conference on Computer Systems, 2015, pp. 1–15.
- [23] M. Parra-Royon and J. Benitez, 'Data Mining Service definition in Cloud Computing'. [Online]. Available: <http://cookingbigdata.com/linkeddata/dmservices/>
- [24] Q. Zhang, A. Haller, and Q. Wang, 'CoCoOn: Cloud Computing Ontology for IaaS Price and Performance Comparison', in *The Semantic Web – ISWC 2019*, vol. 11779, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, Eds., in Lecture Notes in Computer Science, vol. 11779. , Cham: Springer International Publishing, 2019, pp. 325–341. doi: 10.1007/978-3-030-30796-7_21.
- [25] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiș, and V. Munteanu, 'An analysis of mosaic ontology for cloud resources annotation', presented at the 2011 federated conference on computer science and information systems (FedCSIS), IEEE, 2011, pp. 973–980.
- [26] M. Rekik, K. Boukadi, and H. Ben-abdallah, 'Cloud Description Ontology for Service Discovery and Selection', in *Proceedings of the 10th International Conference on Software Engineering and Applications*, Colmar, Alsace, France: SCITEPRESS - Science and Technology Publications, 2015, pp. 26–36. doi: 10.5220/0005556400260036.
- [27] K. P. Joshi, Y. Yesha, and T. Finin, 'Automating Cloud Services Life Cycle through Semantic Technologies', *IEEE Trans. Serv. Comput.*, vol. 7, no. 1, pp. 109–122, Jan. 2014, doi: 10.1109/TSC.2012.41.
- [28] D. Chakraborty, F. Perich, S. Avancha, and A. Joshi, 'Dreggie: Semantic service discovery for m-commerce applications', in *Workshop on reliable and secure applications in mobile environment, in conjunction with 20th symposium on reliable distributed systems (SRDS)*, 2001.
- [29] T. R. Gruber, 'Toward principles for the design of ontologies used for knowledge sharing?', *Int. J. Hum.-Comput. Stud.*, vol. 43, no. 5, pp. 907–928, Nov. 1995, doi: 10.1006/ijhc.1995.1081.
- [30] M. Zhang, 'CoCoOn git repository'. Jul. 21, 2022. Accessed: Sep. 13, 2023. [Online]. Available: <https://github.com/miranda-zhang/cloud-computing-schema>
- [31] I. Horrocks, O. Kutz, and U. Sattler, 'The even more irresistible SROIQ', in *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning*, in KR'06. Lake District, UK: AAAI Press, Jun. 2006, pp. 57–67.
- [32] E. M. Maximilien and M. P. Singh, 'A framework and ontology for dynamic Web services selection', *IEEE Internet Comput.*, vol. 8, no. 5, pp. 84–93, Sep. 2004, doi: 10.1109/MIC.2004.27.
- [33] Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, 'DAML-QoS ontology for Web services', in *Proceedings. IEEE International Conference on Web Services, 2004.*, San Diego, CA, USA: IEEE, 2004, pp. 472–479. doi: 10.1109/ICWS.2004.1314772.
- [34] G. Dobson, R. Lock, I. Sommerville, and Ian Sommerville, 'QoSOnt: a QoS Ontology for Service-Centric Systems', in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, Porto, Portugal: IEEE, 2005, pp. 80–87. doi: 10.1109/EUROMICRO.2005.49.
- [35] X. Wang, T. Vitvar, M. Kerrigan, and I. Toma, 'A QoS-Aware Selection Model for Semantic Web Services', in *Service-Oriented Computing – ICSOC 2007*, vol. 4749, B. J. Krämer, K.-J. Lin, and P. Narasimhan, Eds., in Lecture Notes in Computer Science, vol. 4749. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 390–401. doi: 10.1007/11948148_32.
- [36] K. Kritikos and D. Plexousakis, 'OWL-Q for Semantic QoS-based Web Service Description and Discovery', *Found. Res. Technol. Heraklion Greece*, [Online]. Available: https://publications.ics.forth.gr/_publications/10.1.1.93.9067.pdf
- [37] K. Kritikos, D. Plexousakis, and P. Plebani, 'Semantic SLAs for Services with Q-SLA', *Procedia Comput. Sci.*, vol. 97, pp. 24–33, 2016, doi: 10.1016/j.procs.2016.08.277.
- [38] G. Damiano, E. Giallonardo, and E. Zimeo, 'onQoS-QL: A Query Language for QoS-Based Service Selection and Ranking', in *Service-Oriented Computing – ICSOC 2007*, vol. 4749, B. J. Krämer, K.-J. Lin, and P. Narasimhan, Eds., in Lecture Notes in Computer Science, vol. 4749. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 115–127. doi: 10.1007/978-3-540-93851-4_12.

- [39] F. D. Paoli, M. Palmonari, M. Comerio, and A. Maurino, 'A Meta-model for Non-functional Property Descriptions of Web Services', in *2008 IEEE International Conference on Web Services*, Beijing: IEEE, Sep. 2008, pp. 393–400. doi: 10.1109/ICWS.2008.97.
- [40] K. Kritikos *et al.*, 'A survey on service quality description', *ACM Comput. Surv.*, vol. 46, no. 1, pp. 1–58, Oct. 2013, doi: 10.1145/2522968.2522969.