




Article

A Semantic Model for Interchangeable Microservices in Cloud Continuum Computing

Salman Taherizadeh ¹, Dimitris Apostolou ^{2,3,*} , Yiannis Verginadis ^{2,4}, Marko Grobelnik ¹ 
and Gregoris Mentzas ² 

¹ Artificial Intelligence Laboratory, Jozef Stefan Institute, Jamova cesta 39, 1000 Ljubljana, Slovenia; salman.taherizadeh@gmail.com (S.T.); Marko.Grobelnik@ijs.si (M.G.)

² Institute of Communications and Computer Systems, Iroon Polytechniou 9, 15780 Zografou, Greece; jverg@aub.gr (Y.V.); gmentzas@mail.ntua.gr (G.M.)

³ Department of Informatics, University of Piraeus, Karaoli & Dimitriou 80, 18534 Piraeus, Greece

⁴ School of Business, Department of Business Administration, Athens University of Economics and Business, Patission 76, 10434 Athens, Greece

* Correspondence: dapost@mail.ntua.gr; Tel.: +30-210-7723895

Abstract: The rapid growth of new computing models that exploit the cloud continuum has a big impact on the adoption of microservices, especially in dynamic environments where the amount of workload varies over time or when Internet of Things (IoT) devices dynamically change their geographic location. In order to exploit the true potential of cloud continuum computing applications, it is essential to use a comprehensive set of various intricate technologies together. This complex blend of technologies currently raises data interoperability problems in such modern computing frameworks. Therefore, a semantic model is required to unambiguously specify notions of various concepts employed in cloud applications. The goal of the present paper is therefore twofold: (i) offering a new model, which allows an easier understanding of microservices within adaptive fog computing frameworks, and (ii) presenting the latest open standards and tools which are now widely used to implement each class defined in our proposed model.

Keywords: microservices; cloud continuum computing; semantic model



Citation: Taherizadeh, S.; Apostolou, D.; Verginadis, Y.; Grobelnik, M.; Mentzas, G. A Semantic Model for Interchangeable Microservices in Cloud Continuum Computing. *Information* **2021**, *12*, 40.
<https://doi.org/10.3390/info12010040>

Received: 30 November 2020

Accepted: 13 January 2021

Published: 18 January 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Internet of Things (IoT) [1] solutions have emerged as highly distributed environments in which large amounts of data are generated by devices and transferred toward centralised datacentres. This fact causes not only inefficient utilisation of network bandwidth and computing resources, but also a high-latency response time for IoT applications. Nowadays, in order to decrease both service response time and network traffic load over the Internet, computing platforms are extended from centralised cloud-based infrastructures [2] across the cloud continuum and down to fog resources in close proximity to IoT devices.

A fog node may be referred to communication service providers (e.g., ISP providers, cable or mobile operators), switches, routers or industrial controllers. It could be even a small computing resource at the extreme edge of the network that has a single or multi-processor on-board system (e.g., Raspberry Pi [3], BeagleBoard [4] and PCDuino [5]). In this case, various lightweight operating systems such as CoreOS [6] and RancherOS [7] have been offered so far to exploit such fog computing infrastructures.

In a computing environment that exploits resources available in the cloud continuum, the prominent trend in software engineering is the microservices architecture aimed at an efficient execution of IoT applications [8,9]. Advanced software engineering IoT workbenches such as PrEStoCloud [10] may exploit small, discrete, reusable, elastic microservices generally packaged into lightweight containers, shown in Figure 1. The depicted three-layer cloud continuum computing framework comprises:

- **IoT Devices:** They are simple networked devices such as sensors, actuators and objects which are connected to fog nodes via a wide range of interfaces, for example, 3G, 4G, 5G, Wi-Fi, PCIe, USB or Ethernet.
- **Fog nodes:** These resources provide limited processing, storage and networking functions. Fog nodes may be installed in the end-users' premises. At the edge of the network, fog nodes are typically aimed at sensor data acquisition, collection, aggregation, filtering, compression, as well as the control of sensors and actuators. In comparison to the cloud, which always benefits from an enormous computation and storage capacity, fog nodes still may suffer from the resource limitation. The location of IoT devices may change over time, and hence migrating microservices from one fog node in a specific geographic region to another one needs to be offered to deliver always-on services. Various situations may also arise in which case computation provided on a fog node has to be offloaded to the cloud. A reason for this may include a sudden computational workload, particularly when the number of IoT devices in one geographic location increases at runtime.
- **Cloud resources:** Cloud-based infrastructure provides a powerful central storage and processing capability for IoT use cases. It should be noted that the centralised cloud computing continues to be a significant part of the fog computing model. Cloud resources and fog nodes complement each other to provide a mutually beneficial and interdependent service continuum. Since not all of the data could be processed on the fog nodes itself, further demanding processing tasks such as offline data analytics or long-term storage of data may be performed on the cloud side.

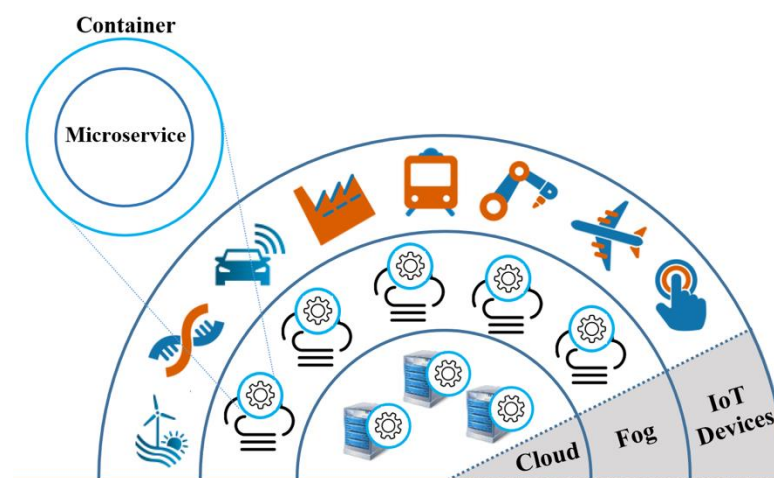


Figure 1. Container-based microservices within a fog computing framework.

Microservices [11] are small, highly decoupled processes which communicate with each other through language independent application programming interfaces (APIs) to form complex applications. Decomposing a single application into smaller microservices allows application providers to distribute the computational load of services among different resources, e.g., fog nodes located in various geographic locations.

In comparison with the service-oriented architecture (SOA) [12], microservices are usually organised around business priorities and capabilities, and they have the capability of independent deployability [13] and often use simplified interfaces, such as representational state transfer (REST). Resilience to failure could be the next characteristic of microservices since every request will be separated and translated to various service calls in this modern architecture. For instance, a bottleneck in one service brings down only that service and not the entire system. In such a situation, other services will continue handling requests in a normal manner. Therefore, the microservices architecture is able to support requirements, namely, modularity, distribution, fault-tolerance and reliability [14]. The microservices architecture also supports the efficiency of re-usable functionality. In other words, a mi-

microservice is a re-usable and well-tested unit that means a single service can be re-used or shared in various applications or even in separate areas of the same application.

Figure 2 shows an example of the microservices architecture in which different services have a varying amount of demands to process their own specific tasks at runtime, and hence it is possible to dynamically scale each service at different level. Due to various demands for different services, for example, cluster service 1 includes four microservice instances, whereas cluster service 2 includes only one microservice instance. This fact indicates a heavy workload sent to cluster service 1 and the lowest level of cluster service 2. The REST API gateway is basically a proxy to microservices, and it acts as a single-entry point into the system. It should be noted that the REST API gateway can be seen as another service like other microservices in the system, and hence it is an elastic component which can be scaled up or down according to the amount of workloads. The REST API gateway is also aimed at addressing some other functionalities such as dealing with security, caching and monitoring at one place. Moreover, it may be capable of satisfying the needs of heterogeneous clients.

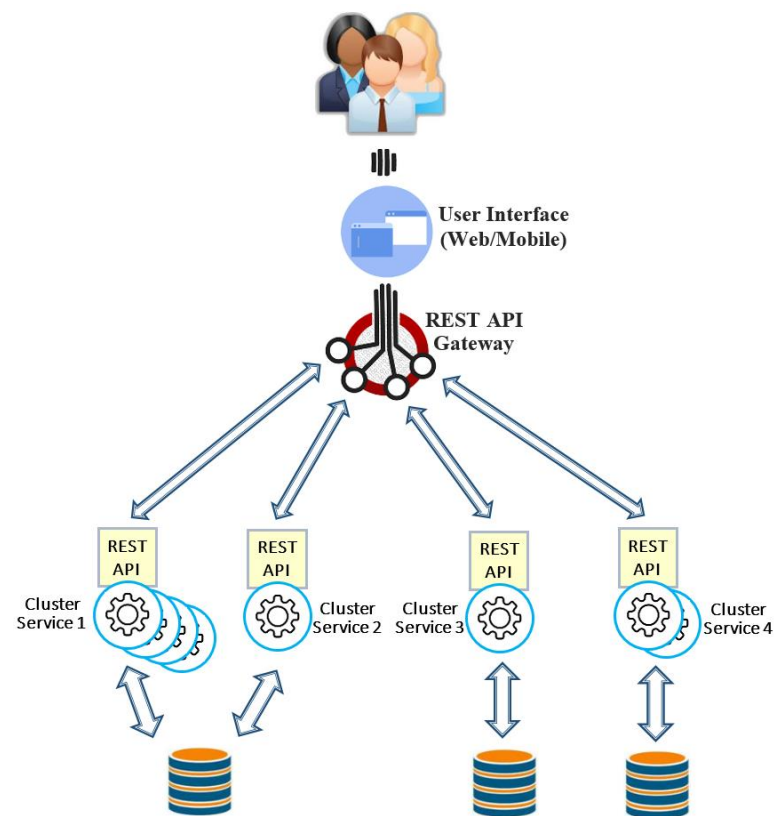


Figure 2. Microservices architecture.

The current focus of cloud continuum platforms is on the dynamic orchestration of microservices to support quality of service (QoS) under varied amounts of workload, while IoT devices move dynamically from one geographic area to another. In order to implement such systems, a wide variety of different technologies should be exploited to develop, deploy, execute, monitor and adapt services. However, interrelations among such various technologies have always been ambiguous, and hence the feasibility of their interoperability is currently uncertain. Besides that, the state-of-the-art in this research domain lacks a systematic classification of technologies used in modern cloud continuum computing environments, and also their interdependency. Each provider of such isolated computing technologies has different interface characteristics and employs a distinct access features. This situation makes a user dependent on a specific microservice provider, unable to exploit another provider without substantial switching costs. Along this line, the EU

general data protection regulation (GDPR) encourages more portability of own data, with the aim of making it simpler to transmit this data from one application or service vendor to another who defines purposes and means of data processing.

To overcome the aforementioned interoperability problems, the primary goal of this paper is to present a new semantic model which clarifies all components necessary to exploit microservices within cloud continuum applications. This model allows both the professional and scientific communities to gain a comprehensive understanding of all technologies which are required to be employed in IoT environments. Beyond only a model, we also highlight widely used tools which can be currently involved in building each component defined in our proposed model. Moreover, we discuss several solutions proposed by research and industry which enable data portability from one microservice to another.

The rest of the paper is organised as follows. Section 2 discusses related work on existing models for fog computing environments. Section 3 describes the research methodology employed to generate the proposed semantic model. Section 4 presents our proposed model in detail. Section 5 outlines available data portability solutions and how they address GDPR requirements. Finally, conclusions appear in Section 6.

2. Related Work

Despite all benefits of cloud continuum computing frameworks, interoperability problem is a key unsolved challenge which needs to be addressed. This is because in light of emerging IoT solutions addressing the needs of domains such as intelligent energy management [15], smart cities [16], lighting [17,18] and healthcare [19], many different technologies and techniques offered by various providers have to be integrated with each other and even customised in most cases. In this regards, very limited research works [20–25], which concentrate on heterogeneous modern computing environments and the related interoperability issues have been conducted. Key members of fog computing community have recognised that they need to work on open standards, specifications and open-source platforms in order to overcome this challenge. Figure 3 shows a number of large foundations and consortiums aiming at such an objective.

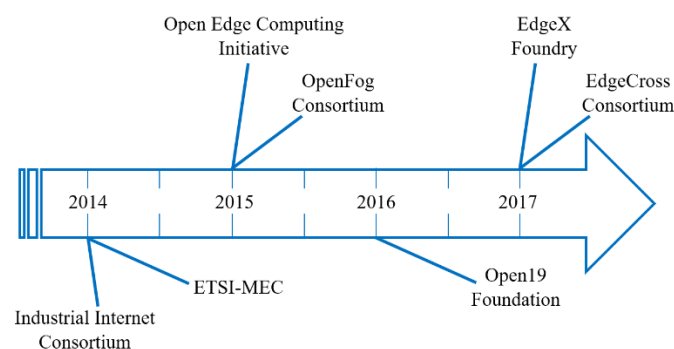


Figure 3. Interoperable open standard projects for fog computing.

The Industrial Internet Consortium [26], founded by Cisco, IBM, AT&T, Intel and GE, was established in 2014. This consortium is aimed at pushing the standardisation for the industrial IoT. The Industrial Internet Consortium published the latest version of their industrial IoT reference architecture in 2017. The European Telecommunications Standards Institute (ETSI) is an independent standardisation organization in the telecommunications industry. Multi-Access edge computing (MEC) [27], which is an industry specification group within ETSI, is aimed at creating a standardised, open environment for the efficient and seamless integration of IoT technologies from service providers, vendors and third parties. The key objective of the Open Edge Computing Initiative [28] is to make all nearby components such as access points, WiFi, DSL-boxes and base stations able to act as computing resources at the edge of the network via standardised, open mechanisms

for IoT applications. Members of this initiative are Intel, Nokia, Telekom, Vodafone, NTT, Crown Castle along with Carnegie Mellon University.

The OpenFog consortium [29] was founded by high-tech industry companies, namely, Cisco, Dell, Intel, ARM and Microsoft together with Princeton University in order to drive standardisation of fog computing. This consortium aims to promote fog computing to improve IoT scenarios. The OpenFog consortium released its reference architecture for fog computing in 2017. The Open19 Foundation [30] was launched in 2016 by LinkedIn, VaporIO and Hewlett Packard Enterprise. This fast-growing project offers an open standard technology able to create a flexible and economic edge solution for operators and datacentres of all sizes with wide levels of adoption. The open-source EdgeX Foundry [31], which is supported by the Linux Foundation, started in 2017. This project is aimed at developing a vendor-neutral framework for IoT edge computing. The EdgeX Foundry allows developers to build, deploy, run and scale IoT solutions for enterprise, industrial and consumer applications. The EdgeCross Consortium [32], established by Oracle, IBM, NEC, Omron and Advantec in Japan, provides an open edge computing software platform. This project stands for addressing the need for edge computing standardisation across industries on a global scale.

One of the significant efforts to improve the cloud interoperability has been made by the mOSAIC project [33] within the European framework FP7. The mOSAIC project has developed an ontology which offers a comprehensive collection of cloud computing concepts. In other words, the mOSAIC ontology provides a unified formal description of different cloud computing components, resources, requirements, APIs, interfaces, service-level agreements (SLAs), etc. Han et al. [34] provided an ontology which includes a taxonomy of cloud-related concepts in order to support cloud service discovery. The proposed cloud ontology can be used to assess the similarity between cloud services and return a list of results matching the user's query. In essence, they presented a discovery system using ontology for VM services according to different search parameters, for example, CPU architecture, CPU frequency, storage capacity, memory size, operating system, and so on.

Bassiliades et al. [35] presented an ontology named PaaSPort which is an extension of the DOLCE+DnS Ultralite ontology [36]. The PaaSPort ontology is exploited to represent Platform-as-a-Service (PaaS) offering capabilities as well as requirements of cloud-based applications to be deployed. In other words, this ontology is able to support a semantic ranking algorithm which primarily recommends the best-matching PaaS offering to the application developers. However, this ontology could be extended to consider the perspective of modern distributed models such as the microservices architecture which is currently one of the state-of-the-art concept in the cloud. Sahlmann and Schwotzer [37] proposed the usage of the oneM2M ontology descriptions [38] enabling automatic resource management as well as service discovery and aggregation. They employed these semantic descriptions for the automatic deployment of virtual IoT edge devices settled at the edge of network. Androcec and Vrcek [39] developed a cloud ontology as a key solution to cope with interoperability issues among different parts of an IoT system, such as service instances, resources, agents, etc. The authors claimed that their ontology, which can be considered as sharable information among different partners, is a tool able to explicitly specify various semantics. The main aim in this work is to clearly describe and categorise the existing features, functionalities and specificities of commercial PaaS offers.

Although existing standards and proposed ontologies address various aspects of the cloud continuum (from centralized clouds, to fog and the extreme edge), they do not provide a comprehensive account of the various aspects of microservices and their emerging role in the cloud continuum. Our work aims to capture the essential concepts and entities of microservices and their relations, focusing mainly on microservices used in self-adaptive IoT applications. Therefore, the focus of our proposed semantic model lies in directions, such as what components need to be defined in a microservice-based IoT fog computing architecture, what parameters have to be taken into consideration for

each component, how to best serve relationships among various entities of a microservice-based IoT environment, how each entity can be characterised, what aspects should be monitored, etc.

3. Research Methodology

Among the methods available for formally describing software engineering entities, we opt for a semantics-based approach which captures the essential concepts, entities and their relations together for microservice-based cloud continuum computing. Therefore, the focus of the proposed extensible semantic model or ontology lies in directions, for example, what components need to be defined in a cloud continuum computing architecture, what parameters have to be taken into consideration for each microservice, how to best serve relationships among various entities of an IoT environment, how each entity can be characterised, etc.

Given a wide variety of computing methods and the increasing complexity of IoT applications, we have followed four steps towards proposing a semantic model clarifying all necessary components and their relations together in cloud continuum computing applications. Figure 4 depicts the research methodology which was adopted in this work.

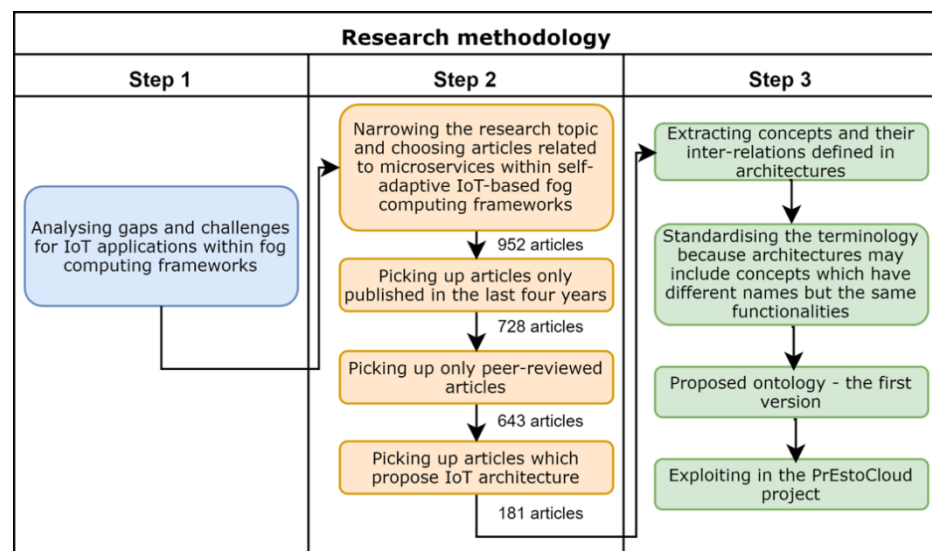


Figure 4. Methodology for developing the semantic model for microservices.

The four steps of the research methodology are explained as follows:

- Step (1): The first step is not only analysing main gaps and challenges for IoT applications, but also expanding our knowledge of this field. It means that the goal is providing an overview of current research areas and challenges in IoT applications orchestrated by cloud continuum computing models.
- Step (2): The second step is the article selection process used to narrow the research topic related to microservices within IoT-based cloud computing frameworks. At the first level, 952 articles were selected. At the second level, 728 articles, which were published only in the last four years, were qualified. We excluded earlier works because they tackled microservices in premature stages of their current development as emerged in 2015 [11]. Afterwards, 643 articles peer-reviewed in qualified journals and highly ranked conferences were found at the third level. At the fourth level, 181 articles proposing an IoT architecture were listed.
- Step (3): At this step, different IoT architectures including various components are presented from all research works in the chosen literature. All required components and their inter-relations defined in architectures were extracted. It should be noted that various architectures may have components which have different names but the

same functionalities and identical concepts. We have standardised the terminology accordingly and generated our proposed semantic model.

4. Microservice Semantic Model

A key characteristic of the design of a semantic model is the notion of interchangeability. In order to ensure interchangeability of microservices within the cloud continuum, it is necessary that the interaction between information objects should be sufficiently well-defined. Therefore, a set of formal specifications for information objects within IoT-based cloud computing environments is necessary. In other words, an abstract formal description of concepts within such IoT environments is required with which these formal specifications can be associated. To this end, a semantic model shown in Figure 5 is defined to clearly specify the relevant notions.

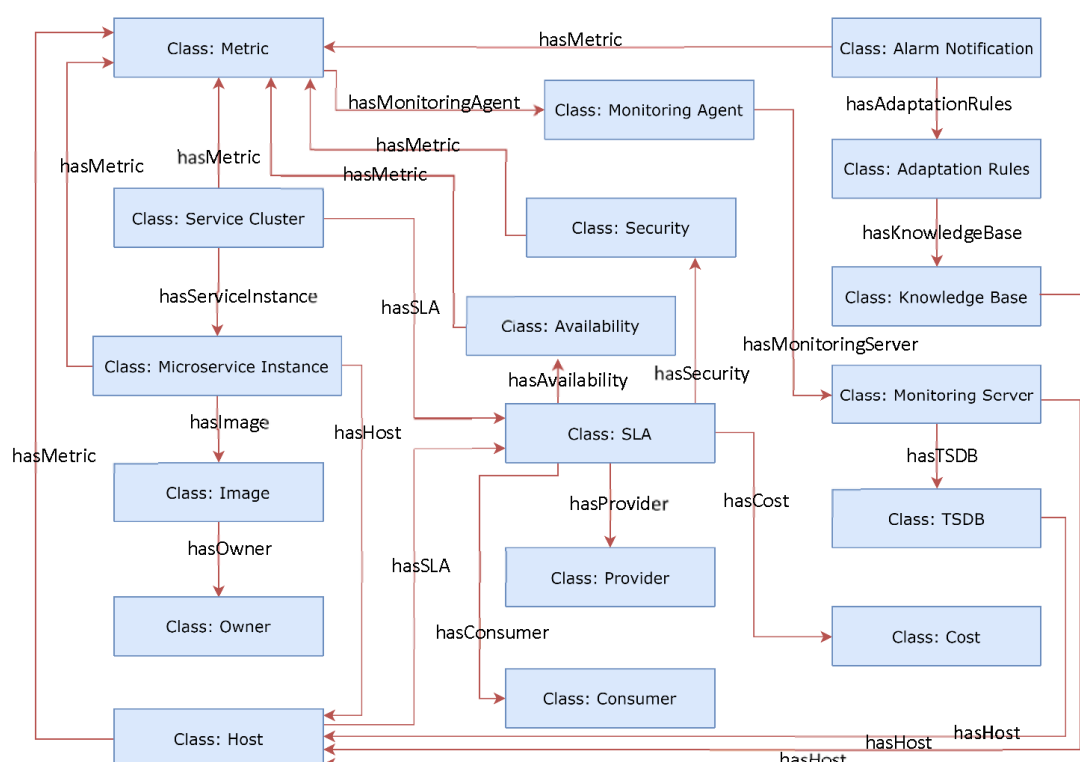


Figure 5. Microservice semantic model depicting classes and interrelations between classes.

All classes, their own properties and relations to each other are explained in detail in the next subsections. Furthermore, different modern open standards and tools which can be employed to implement some of classes defined in our proposed semantic model are discussed and compared to each other.

4.1. Service Cluster

A service cluster is a set of the same microservice instances providing a specific functionality to reply to the requests. Each service cluster, which can consist of one or more microservice instances, offers an individual functionality. When the workload is too high, the service cluster includes more microservice instances; for example, an e-commerce website in which daytime comprises more traffic than at night. At the lowest level of workload intensity, the service cluster includes only one microservice instance. The service cluster class, along with its own properties defined in the semantic model, is presented in Table 1.

Table 1. Service cluster class.

Property Name	Range (Type)	Rationale
Service Cluster ID	unsignedInt	This property allows a unique number to be generated when a new individual service cluster is inserted into the system.
Service Cluster Name	string	This property implies the service cluster's name.
Service Cluster Description	string	This property is an explanation which describes the service cluster.
hasMetric	Metric ID	This relation includes one or more service-level metrics which represent the information about the status of the service cluster as a whole and its performance, such as the aggregated service response time or the aggregated application throughput.
hasMicroserviceInstance	Microservice Instance ID	This relation includes one or more microservice Instances which belong to the service cluster in order to process the tasks.
hasSLA	SLA ID	This relation includes the SLA which applies to the service cluster.

4.2. Microservice Instance Class

Each Service Cluster includes one or more Microservice Instances with regard to the amount of requests to be processed. The Microservice Instance class, along with its own properties defined in the semantic model, is presented in Table 2.

Table 2. Microservice instance class.

Property Name	Range (Type)	Rationale
Microservice Instance ID	unsignedInt	This property allows a unique number to be generated when a new individual microservice instance is inserted into the system.
Microservice Instance Port Number	unsignedInt	The task provided by the microservice instance uses a set of ports which need to be exposed.
Microservice Instance CPU Portion	decimal	Each microservice instance located on a host achieves a proportion of CPU cycles through an assigned relative weight.
Microservice Instance Memory Limit	decimal	Each microservice instance located on a host has its own limit at the use of memory.
Microservice Instance Type	string	A microservice instance is usually packaged into containers. Besides that, it could be also based on VM.
hasMetric	Metric ID	This relation includes one or more service-level metrics which represent the information about the status of an individual microservice instance, such as the its response time or application throughput.
hasHost	Host ID	This relation includes the host where the microservice Instance is located.
hasImage	Image ID	This relation includes the container or VM Image based on which the microservice instance is instantiated.

4.3. Image Class

An image consists of pre-configured files and software. The purpose of an image is to simplify the delivery of a packaged microservice instance. In order to run a microservice instance on a host, the image file needs to be pulled down from a local repository or a public registry (e.g., Docker hub) where this file is stored, and then the microservice instance is instantiated and starts working. The image class, along with its own properties defined in the semantic model, is presented in Table 3.

Table 3. Image class.

Property Name	Range (Type)	Rationale
Image ID	unsignedInt	This property allows a unique number to be generated when a new individual image is inserted into the system.
Image Name	string	Each image stored in a local repository or a public registry such as the Docker hub has its own name.
Image Description	string	This property is used to display the description about the image.
Image Title	string	Title is a short and keywords-relevant description of the image.
Image Functionality	string	This property implies the functionality which is provided by the service included in the image.
Image Version	string	During the application lifecycle, Images may be upgraded several times, and each one has its own version.
Image Instruction Set	string	Different types of image instruction set can be, for example, ARM or X86.
Image Creation Time	dateTime	This property specifies the time when the image was generated or upgraded.
Image File Format	string	This property is used to define the file format of the Image.
Image IRI	anyURI	Images need to be properly indexed through the URI (Uniform resource identifier), the geographic location, and other details for the search facility.
Image License	string	This property specifies the license based on which the image is published.
hasOwner	Owner ID	This relation includes the owner of the image that can be for example the application developer, the service provider, or anyone else.

4.4. Host Class

A host is a resource that not only exhibits the behaviour of a separate infrastructure but is also capable of hosting microservices. A host can be a fog node or a VM on the cloud. The host class, along with its own properties defined in the semantic model, is presented in Table 4.

Table 4. Host class.

Property Name	Range (Type)	Rationale
Host ID	unsignedInt	This property allows a unique number to be generated when a new individual host is inserted into the system.
Host Type	string	Different types of host can be, for example, a fog node or a cloud resource.
Host IP	string	Each host has its own IP address to use the Internet Protocol for communication.
Host Location	string	Each host has its own particular geographic location.
Host Network Interface	string	Each host has its own network adapter to transmit and receive data such as eth0.
Host Network Speed	unsignedInt	This property shows how much bandwidth is assigned to the host.
Host Subnet Mask	string	Each host has its own subnet mask address in the network area.
Host Default Gateway	string	Default gateway is the node that is assumed to know how to forward packets to other networks.
Host OS	string	Each host has an operating system (OS).
Host Storage Size	decimal	Each host has its own particular storage size.
Host Memory Size	decimal	This is a parameter to define the size of an individual host's memory.
Host CPU Count	unsignedInt	This property indicates the number of cores called processors that belong to the host.
Host CPU Clock Rate	decimal	This property indicates the hertz which is the measure of frequency in cycles per second.
Host GPU	string	This property indicates the graphics processing unit (GPU) which is a specialised electronic circuit developed to accomplish rapid mathematical calculations, principally for the purpose of rendering images.
Host Root Username	string	This property indicates the username assigned for the root user employed to connect to the host.
Host Root Password	string	This property indicates the password assigned for the root user employed to connect to the host.
hasMetric	Metric ID	This relation includes one or more host-related metrics which represent the information about the status of an individual infrastructure, such as its current CPU or memory utilisation.
hasSLA	SLA ID	This relation includes the ID of the SLA which applies to the host.

There are various types of fog nodes to be used; for example, communication service providers (e.g., ISP providers, cable or mobile operators), routers, switches, industrial controllers, video surveillance cameras, or even small single-board computers such as Raspberry Pi, BeagleBoard and PCDuino at the extreme edge of the network. Compared to Raspberry Pi, an important characteristic of BeagleBoard and PCDuino is support for Android in addition to Linux operating system. Table 5 compares the latest characteristics of these three single-board computers at the time of writing this paper.

Table 5. Comparison of popular single-board computers.

Specification	Raspberry Pi Model B	BeagleBoard-X15	pcDuino3 Nano
Chip	BCM2837	Sitara AM5728	Allwinner A20
Core	4	1	2
CPU	1.4 GHz	1.5 GHz	1 GHz
GPU	VideoCore IV	PowerVR SGX544	Mali-400MP2
Memory	1 GB RAM	2 GB RAM	1 GB RAM
SD Card	Minimum of 4 GB	Minimum of 4 GB	Minimum of 4 GB
Operating System	Linux	Linux + Android	Linux + Android
Dimensions	85.60 mm × 53.98 mm	107 mm × 102 mm	96 mm × 64 mm

4.5. Metric Class

A metric is a stream of monitoring data. Metrics can be measured at different layers including (i) underlying host (e.g., CPU, memory, disk or network utilisation, etc.), (ii) possible service parameters from deployed microservices (e.g., service response time or application throughput, etc.), as well as (iii) SLA related parameters (e.g., availability, cost, security, etc.) The metric class along with its own properties defined in the semantic model is presented in Table 6.

Table 6. Metric class.

Property Name	Range (Type)	Rationale
Metric ID	unsignedInt	This property allows a unique number to be generated when a new individual monitoring metric is inserted into the system.
Metric Name	string	This property implies the name of the monitoring metric.
Metric Description	string	This is a freestyle textual description of the monitoring metric.
Metric Group	string	For example, all metrics such as memTotal, memFree, memUsed and memUsedPercent may belong to a group named 'memory'.
Metric Level	unsignedInt	Metrics can be monitored as (i) host-level parameters related to the infrastructure utilisation such as CPU, memory, etc., (ii) service-level parameters related to the application performance such as response time, etc., or (iii) SLA-level parameters such as availability, cost, etc.
Metric Unit	string	Metric units can be, for example, (i) percentage, (ii) KBps, (iii) MBps, (iv) Bps, (vi) Yes/No, etc.
Metric Data Type	string	Metric data types can be (i) integer, (ii) long, (iii) double, etc.
Metric Collecting Interval	unsignedInt	The collecting period is an important parameter specially for the time-critical environment. It indicates the monitoring frequency which represents the interval between each measurement for the metric.
Metric History	unsignedInt	This property implies how many days the monitoring system keeps measured values of the metric.
Metric Upper Limit	decimal	This property implies the maximum value of the metric that can be observed.
Metric Lower Limit	decimal	This property implies the minimum value of the metric that can be observed.
Metric Threshold	decimal	This property implies the threshold value of the metric that should be continuously checked.
hasMonitoringAgent	Monitoring Agent ID	This relation includes the monitoring agent which measures the metric instance.

4.6. Monitoring Agent Class

A monitoring agent is a collector able to continuously gather the values of metrics and generate time-stamped monitoring data. It periodically transfers the monitoring data to another component which is called the monitoring server. The monitoring agent class, along with its own properties defined in the semantic model, is presented in Table 7.

Table 7. Monitoring agent class.

Property Name	Range (Type)	Rationale
Monitoring Agent ID	unsignedInt	This property allows a unique number to be generated when a new individual monitoring agent is inserted into the system.
Monitoring Agent Logging	boolean	When this property is set to true, abnormal situations associated to the monitoring agent will be logged.
Monitoring Agent Port	unsignedInt	This is a port which the monitoring agent uses to distribute metrics to monitoring server. It should be identical to the one defined for the monitoring server.
hasMonitoringServer	Monitoring Server ID	This relation includes the ID of the monitoring server to which the monitoring agent distributes measured values of metrics.

4.7. Monitoring Server Class

A monitoring server receives measured values of monitoring metrics sent from monitoring agents, as well as storing such values in the time series database (TSDB) [40]. The monitoring server class, along with its own properties defined in the semantic model, is presented in Table 8.

Table 8. Monitoring server class.

Property Name	Range (Type)	Rationale
Monitoring Server ID	unsignedInt	This property allows a unique number to be generated when a new individual monitoring server is inserted into the system.
Monitoring Server Logging	boolean	When this property is set to true, abnormal situations associated to the monitoring server will be logged.
Monitoring Server Bind Interface	string	The network interface such as eth0 to which the monitoring server's listener will bind.
Monitoring Server Port	unsignedInt	Monitoring server will bind to this port and listen for metric messages distributed by monitoring agents. This property should be identical to the one defined for monitoring agents.
Monitoring Server Heart Beat	unsignedInt	This property implies the intensity based on which the monitoring server's heartbeat should check monitoring agents' availability.
Monitoring Server Heart Retry	unsignedInt	The number of iterations for which the monitoring server heartbeat will allow a monitoring agent to be down until it is declared as dead.
hasTSDB	TSDB ID	This relation includes the ID of TSDB which will be used to store all measured monitoring metrics.
hasHost	Host ID	This relation includes the host where the monitoring server is located.

Table 9 shows a comparison among monitoring systems which can be employed to monitor resource usage (such as CPU, memory, disk, etc.), service-related parameters (such as response time, etc.) and even customised specific metrics. It should be noted that the comparison presented here is based on the reviewed literature and upon our conducting experiments with these tools.

Table 9. Comparison of monitoring tools widely used within fog environments.

Monitoring Server	Open Source	License	Scalability	Alerting
InfluxDB	Yes	MIT	Yes	No
Prometheus	Yes	Apache 2	No	Yes
Scout	Yes	Commercial	No	Yes
SWITCH	Yes	Apache 2	Yes	Yes
NetData	Yes	GPL	Yes	Yes
Zabbix	Yes	GPL	Yes	Yes

cAdvisor is a widely used system specifically designed for measuring, aggregating and showing resource utilisation of running containers such as CPU, memory, etc. However, cAdvisor has restrictions with alert management. Moreover, cAdvisor itself displays monitoring information measured during only the last 60 s. It means that it is not possible to view any features further back with using only a standard installation of cAdvisor. Nevertheless, cAdvisor is able to send the measured data to an external monitoring server such as InfluxDB which also includes a TSDB for long-term storage of monitoring data. In this way, one cAdvisor monitoring agent is responsible for data collection on each node. Then, it sends the collected monitoring data to the InfluxDB monitoring server.

The Prometheus monitoring server [41] which includes a TSDB can be used along with the cAdvisor monitoring agent as an open-source monitoring solution. This monitoring system is able to gather monitoring metrics at various time intervals, and show the monitoring measurements. Scout [42], which is a commercial container monitoring solution, can store measured monitoring values taken from containers during a maximum of 30 days. This monitoring solution also supports notification and alerting of events specified by predetermined thresholds. However, Scout similar to Prometheus may not be able to appropriately offer turnkey scalability to handle large number of containers.

The SWITCH Monitoring System [43], which has an agent/server architecture, can retrieve both host-level monitoring data (e.g., CPU and memory) and service-level parameters (e.g., response time and throughput). Since there is no built-in monitoring agent already prepared for this monitoring platform, the monitoring agent able to measure application-specific metrics should be developed by the service provider. The communication between the SWITCH monitoring agent and the SWITCH monitoring server is implemented through the lightweight StatsD protocol [44] which can be exploited for many programming languages such as Java, Python and C/C++. Netdata [45] is another popular open-source monitoring system which provides real-time performance and health monitoring for infrastructure-related metrics (such as CPU, memory usage) and only specific applications such as Nginx, MongoDB, fail2ban and MySQL. Zabbix [46] is an open-source agent-based monitoring system. It runs on a standalone host which collects monitoring data sent by the Zabbix Agents. The Zabbix monitoring system supports the alerting feature to trigger notification if any predefined situation occurs. The auto-discovery feature of Zabbix may be inefficient [47]. This is because sometimes it may take around five minutes to discover that a monitored node is no longer running in the environment.

4.8. TSDB Class

The monitoring data streams coming from monitoring agents are received by the monitoring server and then stored in the TSDB database for long storage. The TSDB which is a database optimised for time-stamped data can be employed to store all measured values of monitoring metrics. The TSDB class along with its own properties defined in the semantic model is presented in Table 10.

Table 10. Time series database (TSDB) class.

Property Name	Range (Type)	Rationale
TSDB ID	unsignedInt	This property allows a unique number to be generated when a new individual TSDB is inserted into the system.
TSDB DB Username	string	The DB username of the TSDB backend.
TSDB DB Password	string	The DB password of the TSDB backend.
TSDB DB Name	string	The DB name of the TSDB backend.
hasHost	Host ID	This relation includes the Host where the TSDB is located.

In Table 11, a list of widely used TSDB databases and their features in a general sense is provided. Use of Prometheus instead of InfluxDB might be preferred because it stores a name and additional labels for each monitoring metric effectively only once, whereas InfluxDB redundantly stores them for every timestamp. On the other hand, the high availability of Prometheus may be compromised in terms of a durable long-term storage [48]. Besides that, Prometheus is not truly horizontally scalable. However, it supports partitioning via sharding and replication via federation.

Table 11. Comparison of widely used TSDB databases.

TSDB	Open Source	License	Scalability	High Availability
InfluxDB	Yes	MIT	Yes	Yes
Prometheus	Yes	Apache2	No	No
OpenTSDB	Yes	LGPL	Yes	Yes
Druid	Yes	Apache2	Yes	Yes
Elasticsearch	Yes	Apache2	Yes	Yes
Cassandra	Yes	Apache2	Yes	Yes

OpenTSDB stores metrics in sorted order by the name of metric, timestamp, as well as tags. Therefore, in order to perform filtering and aggregation, first of all it needs to scan all row keys in the time interval, and then parse the tags to see if they match the filter criteria, and finally perform the aggregation on the column data.

Druid stores metrics in column format, and it is capable of applying Lempel-Ziv Finite (LZF) [49] compression. LZF, which is a fast compression algorithm, takes very little code space and working memory. Druid's configuration can be such that it ideally keeps all the data in memory. Since Druid keeps bitmaps indexes for dimensions, it is able to efficiently pick out only the rows which it needs to read.

Elasticsearch stores entries as JSON documents. All fields can be indexed and used in a single query. Elasticsearch has a rich client library support which makes it operative by many programming languages such as Java, Python, C/C++, etc.

Cassandra has its specific query language which is named cassandra query language (CQL). CQL adds an abstraction layer in order to hide implementation details and provides native syntaxes for collections and encodings. Cassandra drivers are available for different programming languages, for example Python (DBAPI2), Java (JDBC), Node.JS (Helenus) and C/C++.

4.9. Alarm Notification Class

Alarm notification is an alert system which will be triggered if any metric's threshold is violated. Violations of predefined thresholds have to be reported since even a small amount of violation should not be overlooked. If any alarm notification is initiated, a set of adaptation actions will need to happen in order to adapt the execution environment. The alarm notification class, along with its own properties defined in the semantic model, is presented in Table 12.

Table 12. Alarm notification class.

Property Name	Range (Type)	Rationale
Alarm Notification ID	unsignedInt	This property allows a unique number to be generated when a new individual alarm notification is inserted into the system.
Alarm Notification Date	dateTime	This property specifies the date when a particular alarm notification is triggered.
Alarm Notification Time	dateTime	This property specifies the time when a particular alarm notification is triggered.
Alarm Notification Value	decimal	This property implies the current value of the metric which violates the threshold.
hasMetric	Metric ID	Each alarm notification is associated to the violation of a specific monitoring metric.
hasAdaptationRules	Adaptation Rule ID	For each predefined condition when a particular alarm notification is triggered, specific adaptation rules need to be performed.

4.10. Adaptation Rules Class

If the system is going to experience an abnormal situation, for example it has run out of free CPU cycles, adaptation actions should be taken at that condition, or ideally prior to that situation in anticipation. To this end, different adaptation rules have to be specified since the system should react to various changing conditions, and thus it can improve the application performance and reach a higher level of resource utilisation. Adaptation rules can be determined to react specifically to the increasing workload, whereas there are other adaptation rules to react specifically to the decreasing workload. There are various possible adaptation rules; for instance, (i) horizontal scaling by adding more microservices into the pool of resources, (ii) vertical scaling by resizing the resources, e.g., to offer more bandwidth or computational capacity allocated to microservices, (iii) live migration by moving running microservices from the current infrastructure to another one, or (iv) microservices may be elastically offloaded from a fog node to a specific cloud resource, e.g., if the CPU runs out of free cycles or the memory on the fog node is exhausted. In some cases, such adaptation rules are defined and upgraded by artificial intelligence methods used for prediction in the near future based on collected historical monitoring data stored in the TSDB. To generate adaptation rules, systems use learning algorithms; for example, reinforcement learning, neural network, queuing theory, data mining, regression models, etc. The adaptation rules class, along with its own properties defined in the semantic model, is presented in Table 13.

Table 13. Adaptation rules class.

Property Name	Range (Type)	Rationale
Adaptation Rule ID	unsignedInt	This property allows a unique number to be generated when a new individual adaptation rule is inserted into the system.
Adaptation Rule Action	string	This property indicates a specific action defined by the adaptation rule to be accomplished when a particular condition happens at runtime.
Adaptation Rule Condition	string	This property indicates a specific condition when the adaptation rule has to be performed.
hasKnowledgeBase	KB ID	This relation includes the knowledge base (KB) which will be used to store all information about the environment when the adaptation rule is performed.

4.11. Knowledge Base Class

The knowledge base (KB) may be used for optimisation, interoperability, analysing and integration purposes. Maintaining a KB allows analysis of long-term trends and

supports capacity planning. All information about the environment can be stored in the KB especially when a specific adaptation rule is performed to avoid application performance reduction. Such information can be the response time, application throughput, resource utilisation, characteristics of virtualisation platforms and workload before and after that point. Afterwards, the information stored in the KB may be used as feedback to see how effective adaptation rules help the system handle abnormal situations during execution. The KB class along with its own properties defined in the semantic model is presented in Table 14.

Table 14. Knowledge base class.

Property Name	Range (Type)	Rationale
KB ID	unsignedInt	This property allows a unique number to be generated when a new individual KB is inserted into the system.
KB Username	string	
KB Password	string	
KB Name	string	
hasHost	Host ID	This relation includes the host where the KB is located.

In order to implement the KB, resource description framework (RDF) triple-stores currently offer some advantages over both RDBMS (relational database management system) and NoSQL databases. For example, they do not require a complex schema development in comparison to the way that relational databases do. RDF has features that facilitate data merging even if the underlying schemas differ. While NoSQL databases are all usually different from one another, triple-stores share the same underlying standards, allowing organisations to swap one triple-store for another with minimal disruption. Additionally, triple-stores can infer new knowledge from existing facts, and they can also track where data has come from. There are several open-source RDF repositories systems that can be chosen for implementing the KB. An overview of the most well-known frameworks is provided in Table 15.

Table 15. Overview of widely used RDF triple stores.

Name	Source	License	Supported Query Language
AllegroGraph	No	Commercial	SPARQL
Virtuoso	Yes	Commercial	SPARQL
Sesame	Yes	BSD	SPARQL + SeRQL
Open Anzo	Yes	Eclipse Public	SPARQL
BigData	Yes	GPL	SPARQL
OWLIM-Lite	No	GPL	SPARQL + SeRQL
Jena Fuseki	Yes	Apache 2	SPARQL + SeRQL

It should be noted that the type of data, queries and reasoning required for the KB used in a specific use case will determine which RDF store would be the best candidate to choose. A mechanism for evaluating the performance of different triple-stores is to run an RDF benchmark. The most well-known of the RDF benchmarks is the Berlin SPARQL benchmark (BSBM) [50].

4.12. Service-Level Agreement Class

A service-level agreement (SLA) is an agreement between a provider and a consumer. SLA tends to ensure that a minimum level of service quality is maintained. The SLA class along with its own properties defined in the semantic model is presented in Table 16.

Table 16. Service-level agreement class.

Property Name	Range (Type)	Rationale
SLA ID	unsignedInt	This property allows a unique number to be generated when a new individual SLA is inserted into the system.
SLA Description	string	This is a freestyle textual description of the SLA.
SLA Start Time	dateTime	This property specifies the time when the SLA operation starts.
SLA Support Time Period	unsignedInt	This property specifies the time period when the SLA operation should be supported.
SLA Recovery Time	unsignedInt	This property specifies the time period by which the system must be recovered after an unplanned disruption in the SLA.
hasAvailability	Availability ID	This relation includes the level of availability agreed between provider and consumer in the SLA.
hasSecurity	Security ID	This relation includes the level of security agreed between provider and consumer in the SLA.
hasCost	Cost ID	This relation includes the cost plan agreed between provider and consumer in the SLA.
hasProvider	Provider ID	This relation includes the ID of provider who is primarily responsible for delivering the SLA.
hasConsumer	Consumer ID	This relation includes the ID of consumers who mainly select the SLA definition that best fits their requirement.

4.13. Availability Class

Availability is a key characteristic of infrastructure or application that may be defined in SLA. It is determined as the percentage of the time window in which the system is capable of operating. The availability class, along with its own properties defined in the semantic model, is presented in Table 17.

Table 17. Availability class.

Property Name	Range (Type)	Rationale
Availability ID	unsignedInt	This property allows a unique number to be generated when a new individual availability is inserted into the system.
Availability Description	string	This is a freestyle textual description of the availability defined in SLA.
Availability Logging	boolean	When this property is set to true, abnormal situations associated to the availability defined in SLA will be logged.
Availability Mechanism	string	This property specifies the mechanism exploited to provide the availability defined in SLA.
Availability Standard	string	This property specifies the standard of mechanism exploited to provide the availability defined in SLA.
Availability Requirement	string	This property specifies the requirement for the availability defined in SLA.
hasMetric	Metric ID	This relation includes the monitoring metric based on which the availability is defined in SLA.

4.14. Security Class

Security is also a key demand used to protect data during transmission or storage that may be defined in SLA. The security class along with its own properties defined in the semantic model is presented in Table 18.

Table 18. Security class.

Property Name	Range (Type)	Rationale
Security ID	unsignedInt	This property allows a unique number to be generated when a new individual security is inserted into the system.
Security Description	string	This is a freestyle textual description of the security defined in SLA.
Security Logging	boolean	When this property is set to true, abnormal situations associated to the security defined in SLA will be logged.
Security Mechanism	string	This property specifies the mechanism exploited to provide the security defined in SLA.
Security Standard	string	This property specifies the standard of mechanism exploited to provide the security defined in SLA.
Security Requirement	string	This property specifies the requirement for the security defined in SLA.
hasMetric	Metric ID	This relation includes the monitoring metric based on which the security is defined in SLA.

4.15. Cost Class

SLA is a legal document which can define the cost of infrastructure or service including any possible penalty terms. The cost class along with its own properties defined in the semantic model is presented in Table 19.

Table 19. Cost class.

Property Name	Range (Type)	Rationale
Cost ID	unsignedInt	This property allows a unique number to be generated when a new individual cost is inserted into the system.
Cost Description	string	This is a freestyle textual description of the cost defined in SLA.
Cost Currency	currency	This property specifies the currency used in SLA.
Cost Pricing Model	string	This property specifies the pricing model exploited in SLA. It can be (i) free, (ii) pay-per-use, (iii) flat fee, or any other option.
Cost Penalty	string	This property specifies the penalty which can be taken into account if the provider violates any terms of SLA.

5. Microservice Data Portability

The right to data portability is considered as one of the main fundamental data subjects' rights in the GDPR. According to Article 20 of the GDPR, the right to data portability enables data subjects in Europe to acquire their own data from the data controller gathering their data or transfer it to another data controller. A data controller is an organisation or any other body which determines the purpose and means of own data processing operations. From the user viewpoint, data subjects have three data portability rights according to the GDPR:

- (i). Right to receive their own data from data controllers. Upon request by data subjects, they are granted the possibility to store their own data on their private storage device. The right to transmit own data prevents the vendor lock-in situation and gives data subjects ownership to their own data.
- (ii). Right to transmit own data indirectly to another data controller. Data subjects may request transmission and reuse of their own data among various data controllers, either within the same business sector or in any other sector in which they are interested.
- (iii). Right to have their own data transmitted directly to another data controller. Upon request by data subjects, their own data can be transmitted from a data controller

directly to another one, without any hindrance from data controller from whom the own data is withdrawn. However, Article 20 of GDPR mentions a limitation for the exercise of this right that this direct transmission has to be technically feasible.

In cloud continuum applications, microservices act as data controllers. They often collect and transmit own data through various sensing probes used in IoT applications, such as location trackers. Hence, microservices should adhere to the aforementioned data portability rights. In the next sections, we outline prominent data portability methods, discuss their similarities and differences, as well as their applicability in microservice-based cloud continuum applications.

5.1. Common Open Application Programming Interface (API)

An API provides a suitable abstraction, which is considered as a widely used interface to share, transmit and reuse data as well as content between services. It is a powerful standard tool for expressing complex application specifications. Implementing a common open API can facilitate data portability in an easy manner, which allows users to access their data stored on whatever resource or transmit it to another one. The most prominent point is that data can be accessed through the same API exposed by various services in every programming language. However, designing a common open API is not an optimal solution as different microservices are developed independently from others and they may adhere to different programming approaches. Coupled with the fact that upgrading an open API may have cumbersome consequences, different service developers, who expose the same API, prefer to keep their code without any change. Therefore, once a common open API is published, a long-term commitment has to be maintained even if the underlying functionality would be grown or shifted. Besides this, the movement of data from one microservice to another via a common open API may require storing the data temporarily on a medium such as a local disk in advance.

5.2. Centralised Framework

A centralised method can be considered as a collaboration committed to building a common portability framework which can connect all services together. Such a framework enables seamless, direct, user-initiated data portability among service vendors participating in the initiative. This service-to-service portability method is centred on the idea that data subjects should be able to freely transfer their data from a data controller directly to another one. This centralised method may lower the infrastructure costs with the common framework and reduce friction among data controllers. The centralised framework is in charge of converting a range of proprietary formats into a small number of canonical formats (called data models) useful for transferring data. It means that the centralised framework acts as intermediary among various proprietary formats, converting their data to canonical formats that can be used for data transmission. It should be noted that the centralised framework should not have access to the actual data either in transit or at rest since data exchange will be in encrypted form between export and import. One of the main examples of such a centralized method is the open-source data transfer project (DTP), which has been formed by big data-driven companies including Google, Facebook, Microsoft and Twitter.

5.3. Standard Formats

This method indicates storing data in commonly used standard formats such as JSON (JavaScript object notation), XML (extensible markup language), TOSCA (Topology and orchestration specification for cloud applications) or CSV (Comma separated values) file archives. However, such standards format specific only the structure of the data and not the intended meaning and use.

5.4. Core Vocabularies and Ontologies

Core vocabularies, which provide a catalogue of ontologies, are considered as simplified, reusable and extensible data models that capture the fundamental characteristics of a particular entity in a context-neutral fashion [51]. Core vocabularies are currently expressed in different formats such as conceptual model and spreadsheet as well as the RDF schema. Since the existing core vocabularies and ontologies may not fully cover data controllers' needs, data controllers can use vocabularies and ontologies from the core, and add their own extensions to them. Approval of a formal consensual extension is a complex process which can take some time, but it results in official approbation that will reassure data controllers about the validity of the extension. As the main drawback, this approach currently exhibits narrow scopes, and hence such framework needs to be taken into account as complementary to existing ontologies.

5.5. Discussion

Table 20 presents an overview of the main challenges the GDPR poses to data portability methods in the context of microservice-based cloud continuum applications. As can be seen, the comparison implies that most of the mentioned challenges pertaining to the existing data portability methods may be addressed by the centralised framework. However, the success of this method highly depends on a high level of open consensual agreement about how much data all participating data controllers, i.e., microservice vendors, are willing to share as well as how much trust they would like to place in this framework. Moreover, a single point of failure as an important challenge unfortunately is a part of this centralised framework. It means that if this framework fails, a bottleneck in the centralised framework will bring down the entire cloud continuum application. Therefore, none of existing widely used data portability methods yet offer an integrated solution capable of addressing the whole fundamental challenges. Consequently, our analysis of the technical challenges in relation to existing data portability methods indicates that user rights activists and development professionals need to collaborate more closely if they wish to put the right to data portability into operation in cloud continuum applications.

Table 20. Overview of challenges in relation to data portability methods.

	Common Open API	Centralised Framework	Standard Formats	Core Vocabularies and Ontologies
Data controllers incur data conversion overhead	Yes	No	No	No
Redundant temporary data storage is required on data controllers	Yes	No	No	No
Data controllers get involved in any upgrade in data portability method	Yes	No	No	No
Storage size is drastically increased on data controllers	No	No	Yes	Yes
Poor hardware prefetching performance may impact data controllers	No	No	No	Yes
Significant investment is required by data controllers	No	No	No	Yes
Long conversion time is required within data controllers	Yes	No	No	No
Difficult to be consensual by all data controllers	Yes	Yes	Yes	Yes
Limiting the freedom and right of data controllers	No	No	No	No
Large effort in data structure design is required by all data controllers	No	No	No	Yes
Single point of failure intrinsic to data portability framework	No	Yes	No	No

6. Conclusions

The rapid growth of the cloud continuum has a great impact on the adoption of new, microservice-based software engineering models. Such modern computing models are aimed at increasing capabilities of cloud continuum infrastructures compared to traditional architectures. This happens through the capability of placing services in the close proximity of users or IoT devices, offering a low-latency response time for IoT applications. In such a distributed computing environment, the latest trend in software engineering is the microservices architecture intended to offer an efficient execution of container-based IoT solutions adaptive in dynamic environments.

A systematic classification of notions has been presented in order to enable easier understanding and facilitate interchangeability of cloud continuum applications using microservices. This is because a broad variety of different isolated technologies which have various interface characteristics need to be employed to implement such systems. This research work presents a new semantic model which determines all components and interrelations among them that are necessary in these cloud continuum computing applications. The semantic model allows both the scientific community and professional associations to approach a comprehensive understanding of all concepts required to be employed in cloud continuum environments. As a consequence, the current interoperability issue in cloud continuum computing models can be addressed, and hence concepts can be translated from one information context to another without forcing rigid adherence to a specific technology stack.

We also compared the existing data portability methods widely used by online data-driven service providers. The aim of the comparison is to identify and establish a trade-off of the strengths and drawbacks which have been encountered in the context of data portability mechanisms. Our analysis shows that an integrated solution that fully meets all technical challenges is currently unavailable and that further research is needed to ensure that next-generation, microservice-based cloud continuum applications are GDPR-compliant.

Author Contributions: Data curation, S.T., D.A., Y.V., M.G. and G.M.; Microservices overview S.T., D.A., Y.V.; Related work analysis, S.T., D.A., G.M.; Research methodology S.T., D.A., M.G.; Microservice semantic model S.T., D.A., Y.V., M.G. and G.M.; Microservice data portability S.T., D.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Matheu-Garcia, S.N.; Hernandez-Ramos, J.L.; Skarmeta, A.F.; Baldini, G. Risk-based automated assessment and testing for the cybersecurity certification and labelling of IoT devices. *Comput. Stand. Interfaces* **2019**, *62*, 64–83. [\[CrossRef\]](#)
2. Bounagui, Y.; Mezrioui, A.; Hafiddi, H. Toward a unified framework for cloud computing governance: An approach for evaluating and integrating IT management and governance models. *Comput. Stand. Interfaces* **2019**, *62*, 98–118. [\[CrossRef\]](#)
3. Gupta, V.; Kaur, K.; Kaur, S. Developing small size low-cost software-defined networking switch using Raspberry Pi. *Next Gener. Netw.* **2018**, 147–152. [\[CrossRef\]](#)
4. Adam, G.; Kontaxis, P.; Doulos, L.; Madias, E.-N.; Bouroussis, C.; Topalis, F. Embedded microcontroller with a CCD camera as a digital lighting control system. *Electronics* **2019**, *8*, 33. [\[CrossRef\]](#)
5. Madumal, P.; Atukorale, A.S.; Usoof, H.A. Adaptive event tree-based hybrid CEP computational model for fog computing architecture. In Proceedings of the 2016 Sixteenth International Conference on Advances in ICT for Emerging Regions (ICTer), IEEE, Negombo, Sri Lanka, 1–3 September 2016; pp. 5–12.
6. Casalicchio, E. Container orchestration: A survey. In *Systems Modeling: Methodologies and Tools*; Springer: Cham, Switzerland, 2019; pp. 221–235.
7. Kakakheh, S.R.U.; Mukkala, L.; Westerlund, T.; Plosila, J. Virtualization at the network edge: A technology perspective. In Proceedings of the 2018 Third International Conference on Fog and Mobile Edge Computing (FMEC), Barcelona, Spain, 23–26 April 2018; pp. 87–92.
8. Fernandez-Garcia, L.I.A.C.J.C.; Jesus, A.; Wang, J.Z. A flexible data acquisition system for storing the interactions on mashup user interfaces. *Comput. Stand. Interfaces* **2018**, *59*, 10–34. [\[CrossRef\]](#)

9. Donassolo, B.; Fajjari, I.; Legrand, A.; Mertikopoulos, P. Fog based framework for iot service orchestration. In Proceedings of the 2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 11–14 January 2019; pp. 1–2.
10. Papageorgiou, N.; Apostolou, D.; Verginadis, Y.; Tsagkaropoulos, A.; Mentzas, G. Situation detection on the edge. In Proceedings of the Workshops of the International Conference on Advanced Information Networking and Applications, Matsue, Japan, 27–29 March 2019; Springer: Cham, Switzerland, 2019; pp. 985–995.
11. Thönes, J. Microservices. *IEEE Softw.* **2015**, *32*, 116. [[CrossRef](#)]
12. Tsakos, K.; Petrakis, E.G. Service oriented architecture for interconnecting lora devices with the cloud. In Proceedings of the International Conference on Advanced Information Networking and Applications, Matsue, Japan, 27–29 March 2019; Springer: Cham, Switzerland, 2019; pp. 1082–1093.
13. Todoli-Ferrandis, D.; Silvestre-Blanes, J.; Santonja-Climent, S.; Sempere-Paya, V.; Vera-Perez, J. Deploy & forget wireless sensor networks for itinerant applications. *Comput. Stand. Interfaces* **2018**, *56*, 27–40.
14. Stubbs, J.; Moreira, W.; Dooley, R. Distributed systems of microservices using docker and serfnode. In Proceedings of the 2015 7th International Workshop on Science Gateways, Budapest, Hungary, 3–5 June 2015; pp. 34–39.
15. Marinakis, V.; Doukas, H. An advanced IoT-based system for intelligent energy management in buildings. *Sensors* **2018**, *18*, 610. [[CrossRef](#)]
16. Lavallo, A.; Teruel, M.A.; Maté, A.; Trujillo, J. Improving Sustainability of Smart Cities through Visualization Techniques for Big Data from IoT Devices. *Sustainability* **2020**, *12*, 5595. [[CrossRef](#)]
17. Durmus, D. Real-Time Sensing and Control of Integrative Horticultural Lighting Systems. *J. Multidiscip. Sci. J.* **2020**, *3*, 266–274. [[CrossRef](#)]
18. Gagliardi, G.; Lupia, M.; Cario, G.; Tedesco, F.; Cicchello Gaccio, F.; Lo Scudo, F.; Casavola, A. Advanced Adaptive Street Lighting Systems for Smart Cities. *Smart Cities* **2020**, *3*, 1495–1512. [[CrossRef](#)]
19. He, Y.; Fu, B.; Yu, J.; Li, R.; Jiang, R. Efficient Learning of Healthcare Data from IoT Devices by Edge Convolution Neural Networks. *Appl. Sci.* **2020**, *10*, 8934. [[CrossRef](#)]
20. Noura, M.; Atiquzzaman, M.; Gaedke, M. Interoperability in internet of things: Taxonomies and open challenges. *Mob. Netw. Appl.* **2018**, 1–14. [[CrossRef](#)]
21. Koo, J.; Oh, S.-R.; Kim, Y.-G. Device identification interoperability in heterogeneous iot platforms. *Sensors* **2019**, *19*, 1433. [[CrossRef](#)] [[PubMed](#)]
22. Kalatzis, N.; Routis, G.; Marinellis, Y.; Avgeris, M.; Roussaki, I.; Papavassiliou, S.; Anagnostou, M. Semantic interoperability for iot plat-forms in support of decision making: An experiment on early wild fire detection. *Sensors* **2019**, *19*, 528. [[CrossRef](#)]
23. Ahmad, A.; Cuomo, S.; Wu, W.; Jeon, G. Intelligent algorithms and standards for interoperability in internet of things. *Future Gener. Comput. Syst.* **2018**. [[CrossRef](#)]
24. Fortino, G.; Savaglio, C.; Palau, C.E.; de Puga, J.S.; Ganzha, M.; Paprzycki, M.; Montesinos, M.; Liotta, A.; Llop, M. Towards multi-layer interoperability of heterogeneous iot platforms: The interiot approach. In *Integration, Interconnection, and Interoperability of IoT Systems*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 199–232.
25. Garcia, A.L.; del Castillo, E.F.; Fernandez, P.O. Standards for enabling heterogeneous iaas cloud federations. *Comput. Stand. Interfaces* **2016**, *47*, 19–23. [[CrossRef](#)]
26. Kemppainen, P. Pharma industrial internet: A reference model based on 5g public private partnership infrastructure, industrial internet consortium reference architecture and pharma industry standards. *Nord. Balt. J. Inf. Commun. Technol.* **2016**, *2016*, 141–162.
27. Yang, S.-R.; Tseng, Y.-J.; Huang, C.-C.; Lin, W.-C. Multi-access edge computing enhanced video streaming: Proof-of-concept implementation and prediction/qoe models. *IEEE Trans. Veh. Technol.* **2019**, *68*, 1888–1902. [[CrossRef](#)]
28. Open Edge Computing Initiative 2019. Available online: <http://openedgecomputing.org/> (accessed on 15 April 2019).
29. Yannuzzi, M.; Irons-Mclean, R.; Van-Lingen, F.; Raghav, S.; Somaraju, A.; Byers, C.; Zhang, T.; Jain, A.; Curado, J.; Carrera, D.; et al. Toward a converged openfog and etsi mano architecture. In Proceedings of the 2017 IEEE Fog World Congress (FWC), Santa Clara, CA, USA, 30 October–1 November 2017; pp. 1–6.
30. Open19 Foundation 2019. Available online: <https://www.open19.org/> (accessed on 15 April 2019).
31. EdgeX Foundry 2019. Available online: <https://www.edgexfoundry.org/> (accessed on 15 April 2019).
32. EdgeCross Consortium 2019. Available online: <https://www.edgexcross.org/en/> (accessed on 15 April 2019).
33. Cretella, G.; Martino, B.D. A semantic engine for porting applications to the cloud and among clouds. *Softw. Pract. Exp.* **2015**, *45*, 1619–1637. [[CrossRef](#)]
34. Han, T.; Sim, K.M. An ontology-enhanced cloud service discovery system. In Proceedings of the International Multi Conference of Engineers and Computer Scientists, Hong Kong, China, 17–19 March 2010; Volume 1, pp. 17–19.
35. Bassiliades, N.; Symeonidis, M.; Gouvas, P.; Kontopoulos, E.; Meditskos, G.; Vlahavas, I. Paasport semantic model: An ontology for a platform-as-a-service semantically interoperable marketplace. *Data Knowl. Eng.* **2018**, *113*, 81–115. [[CrossRef](#)]
36. Agarwal, R.; Fernandez, D.G.; Elsaleh, T.; Gyrard, A.; Lanza, J.; Sanchez, L.; Georgantas, N.; Issarny, V. Unified iot ontology to enable interoperability and federation of testbeds. In Proceedings of the 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, USA, 12–14 December 2016; pp. 70–75.
37. Sahlmann, K.; Schwotzer, T. Ontology-based virtual iot devices for edge computing. In Proceedings of the 8th International Conference on the Internet of Things, Santa Barbara, CA, USA, 15–18 October 2018; p. 15.

38. Sahlmann, K.; Scheffler, T.; Schnor, B. Ontology-driven device descriptions for iot network management. In Proceedings of the 2018 Global Internet of Things Summit (GloTS), Bilbao, Spain, 4–7 June 2018; pp. 1–6.
39. Androcec, D.; Vrcek, N. Ontologies for platform as service apis inter-operability. *Cybern. Inf. Technol.* **2016**, *16*, 29–44.
40. Naqvi, S.N.Z.; Yfantidou, S.; Zimanyi, E. Time series databases and influxdb. In *Studienarbeit*; Université Libre de Bruxelles: Brussels, Belgium, 17 December 2017.
41. Kumari, M.; Vishwanathan, A.; Dash, D. Real-time cloud monitoring solution using prometheus tool and predictive analysis using arimamodel. *Int. J. Eng. Sci.* **2018**, *8*, 18338.
42. Scout 2019. Available online: <https://scoutapp.com/> (accessed on 15 April 2019).
43. Taherizadeh, S.; Stankovski, V. Dynamic multi-level auto-scaling rules for containerized applications. *Comput. J.* **2018**, *62*, 174–197. [[CrossRef](#)]
44. The StatsD protocol 2019. Available online: <https://github.com/etsy/statsd/wiki> (accessed on 15 April 2019).
45. Netdata 2019. Available online: <https://my-netdata.io/> (accessed on 15 April 2019).
46. Petrucci, C.-M.; Puiu, B.-A.; Ivanciu, I.-A.; Dobrota, V. Automatic management solution in cloud using ntopng and Zabbix. In Proceedings of the 2018 17thRoEduNet Conference: Networking in Education and Research (RoE-duNet), Cluj-Napoca, Romania, 6–8 September 2018; pp. 1–6.
47. Taherizadeh, S.; Jones, A.C.; Taylor, I.; Zhao, Z.; Stankovski, V. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *J. Syst. Softw.* **2018**, *136*, 19–38. [[CrossRef](#)]
48. Bader, A.; Kopp, O.; Falkenthal, M. Survey and comparison of opensource time series databases. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*; Gesellschaft für Informatik e.V.: Bonn, Germany, 2017; pp. 249–268.
49. Jeyakumar, V.; Madani, O.; Parandeh, A.; Kulshreshtha, A.; Zeng, W.; Yadav, N. Explainit!—A declarative root-cause analysis engine for timeseries data (extended version). *arXiv* **2019**, arXiv:1903.08132.
50. Bizer, C.; Schultz, A. The berlin sparql benchmark. *Int. J. Semant. Web Inf. Syst. (IJSWIS)* **2009**, *5*, 1–24. [[CrossRef](#)]
51. Taherizadeh, S.; Stankovski, V.; Grobelnik, M. A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. *Sensors* **2018**, *18*, 2938. [[CrossRef](#)]