






Article

Extending TOSCA for Edge and Fog Deployment Support

Andreas Tsagkaropoulos ^{1,*}, Yiannis Verginadis ^{1,2}, Maxime Compastie ³, Dimitris Apostolou ^{1,4}
and Gregoris Mentzas ¹

¹ Information Management Unit (IMU), Institute of Communication and Computer Systems, National Technical University of Athens (NTUA), 15780 Athens, Greece; jverg@aueb.gr (Y.V.); dapost@unipi.gr (D.A.); gmentzas@mail.ntua.gr (G.M.)

² Department of Business Administration, Athens University of Economics and Business (AUEB), 10434 Athens, Greece

³ ActiveEon S.A.S, 06560 Sophia-Antipolis, France; maxime.compastie@activeeon.com

⁴ Department of Informatics, University of Piraeus, 18534 Piraeus, Greece

* Correspondence: atsagkaropoulos@mail.ntua.gr

Abstract: The emergence of fog and edge computing has complemented cloud computing in the design of pervasive, computing-intensive applications. The proximity of fog resources to data sources has contributed to minimizing network operating expenditure and has permitted latency-aware processing. Furthermore, novel approaches such as serverless computing change the structure of applications and challenge the monopoly of traditional Virtual Machine (VM)-based applications. However, the efforts directed to the modeling of cloud applications have not yet evolved to exploit these breakthroughs and handle the whole application lifecycle efficiently. In this work, we present a set of Topology and Orchestration Specification for Cloud Applications (TOSCA) extensions to model applications relying on any combination of the aforementioned technologies. Our approach features a design-time “type-level” flavor and a run time “instance-level” flavor. The introduction of semantic enhancements and the use of two TOSCA flavors enables the optimization of a candidate topology before its deployment. The optimization modeling is achieved using a set of constraints, requirements, and criteria independent from the underlying hosting infrastructure (i.e., clouds, multi-clouds, edge devices). Furthermore, we discuss the advantages of such an approach in comparison to other notable cloud application deployment approaches and provide directions for future research.

Keywords: TOSCA; function-as-a-service; fog computing; cloud computing; model-driven engineering; cloud applications



Citation: Tsagkaropoulos, A.; Verginadis, Y.; Compastie, M.; Apostolou, D.; Mentzas, G. Extending TOSCA for Edge and Fog Deployment Support. *Electronics* **2021**, *10*, 737. <https://doi.org/10.3390/electronics10060737>

Academic Editor: Rashid Mehmood

Received: 24 February 2021

Accepted: 16 March 2021

Published: 20 March 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

Cloud computing has become a widely valued commodity with constantly increasing popularity among large and small to mid-sized enterprises (SMEs), as well as public organizations. According to the report by Gartner, an 18.4% increase in total public cloud computing revenue is expected for 2021 compared to 2020, reaching 304.9 billion USD [1]. Furthermore, the infrastructure-as-a-service segment of cloud computing is forecasted to experience the highest growth.

As more applications make use of the cloud and more cloud providers appear, “vendor lock-in” becomes an increasingly important issue. The lack of common standards and the heterogeneity of cloud provider solutions put at risk the portability of data and applications, as moving to a technology supported by a different provider may be associated with high costs [2]. The need for a common and interoperable standard is further augmented due to the appearance of fog computing, which relies on edge devices to complement cloud resources. Fog computing has been gaining a firm foothold as a means to reduce cost, both in the form of networking and hosting expenditures. To handle the massive volume of IoT data with lower latency and to address privacy concerns (e.g., by ensuring that the processing of data is performed in the vicinity of its generation) [3], fog computing

requires advanced deployment capabilities. Therefore, an advanced common standard can not only contribute towards provider-agnostic orchestration, but can also fill gaps in the management of application assets in fog environments.

Based on the review by Bergmayr et al. [4], one of the most prominent modeling languages for handling principally cloud deployments is TOSCA [5], which is an Organization for the Advancement of Structured Information Standards (OASIS) standard that sets the tone for all other cloud modeling languages that try to be compatible with it. There are numerous [6–8] implementations based on this standard that have been regularly maintained.

Notwithstanding the efforts towards its adoption, TOSCA orchestration is impeded by several limitations preventing its direct usage within a hybrid cloud and edge environment. Firstly, TOSCA lacks a native mechanism to track run time deployments and factor-in any reconfiguration decisions [9]. For example, TOSCA's most prominent implementations Alien4Cloud [6], Cloudify [7], and OpenTOSCA [8] focus on the design of the application topology—the modification of the initial template requires the manual intervention of the DevOps or else risks a discrepancy between the model and the real deployment. Especially due to this first issue, the adoption of TOSCA to address the needs of cloud application deployments in production has been met by solutions such as HashiCorp's Terraform (<https://www.terraform.io>) and Red Hat's Ansible (<https://www.ansible.com>). Secondly, a TOSCA template created based on the approaches in [6–8] is required to explicitly define the properties of the application and configure the usage of different cloud providers, along with the characteristics of the VMs that are needed. The need to specify this information limits the dynamicity in deployments using these platforms, which is a prerequisite in edge and fog environments. Finally, even when the application is manually reconfigured, optimization cannot be supported [4]. Thus, it becomes difficult to maintain a model of a real-world application featuring a dynamic topology, which may not scale predictably. These last two issues are also evident in the Terraform and Ansible solutions. These open-source platforms use declarative and pre-defined run time configuration languages to state the desired final state of the cloud application deployment. As a consequence, this channels the responsibilities of Development and Operations (DevOps) engineers towards maintaining a cloud application topology that must address contradicting requirements, such as the cost and quality of service. Therefore, such solutions may be popular at the moment since they provide a straightforward way of deploying applications in several cloud vendors, however they still lack real and automatic cross-cloud optimization capabilities. Additionally, their ability to cope with new paradigms, such as edge computing and other distributed execution approaches, may rely on tedious procedures that will result in bespoke solutions that may endanger their portability.

The VM-based paradigm has lately been challenged by the newer serverless computing paradigm. A serverless computing platform is defined in [10] as “a platform that hides server usage from developers and runs code on-demand, automatically scaled and billed only for the time the code is running”. Serverless computing has given rise to numerous function-as-a-service (FaaS) platforms, some of which have been successfully coupled with deployments on edge resources, e.g., OpenWhisk [11] and OpenWhisk Lean [12]. The available TOSCA specification [5] is very extensible but also quite generic and lacking explicit modeling artefacts for fog applications, serverless approaches, or other distributed execution approaches—e.g., the Java Parallel Processing Framework (JPPF) [13]. Thus, any custom solutions that are developed pose a barrier to wider adoption of a reference TOSCA-based methodology to handle basic concepts in these fields.

Based on our experience working on the PrEstoCloud framework [14], we argue that in such dynamic environments there is an urgent need to shift some of the core responsibilities of DevOps to an appropriate middleware. This middleware will automatically support the separation of resources between the design time and run time and will provide appropriate error-free maintenance of dynamic topologies. Therefore, the DevOps should be able to model the deployment requirements as generically as possible, without having to make

concrete decisions on the initial deployment, optimization, and reconfiguration of cross-clouds and fog computing applications. This is needed in modern organizations that need to make sure they optimally use their resources. To achieve this, we first provide elaborated extensions to TOSCA, which is the current cloud modeling standard, and secondly we present a proof-of-concept on the implementation of a middleware that can exploit the extended TOSCA. We decided to propose extensions to TOSCA, since they can be made a part of this standard, which then can be used to extend in a bespoke manner any other solution that is used now or in the future for cloud applications in production (e.g., Terraform, Ansible). Additionally, continuous cloud modeling support (through the use of standards and TOSCA specifically) has been argued before [4] as desirable for aligning existing and potential cloud modeling languages, and therefore achieving interoperability. As such, in this article, the research questions we aim to answer are the following:

- Which semantic enhancements should be made to TOSCA to describe and enact fog deployments?
- What is the methodology that should be followed to model deployments featuring distributed execution paradigms?
- How can the optimization aspects regarding the deployment be included in the modeling artefacts?

To address the above research questions, we propose a modeling approach for the definition of applications using TOSCA. We argue that TOSCA should support generic use-case patterns and deployments using serverless and other distributed execution paradigms by providing a set of relevant generic constructs. To this end, we propose a series of custom approaches, addressing deployments in hybrid clouds (i.e., combined use of private and public cloud resources), multi-clouds (i.e., combined use of public cloud resources from different vendors), edge-based applications, and FaaS-based applications, extending the base YAML Ain't Markup Language (YAML) TOSCA implementation [5]. To allow the optimization of the topology, we suggest that two versions of the TOSCA model should be used—initially an abstract version focusing on the structure of the topology and subsequently a more concrete version, which would include more specific details of the actual deployment. We validate our approach using a qualitative evaluation method based on a motivational scenario.

The remainder of the article is structured as follows. In Section 2, we discuss the state-of-the-art related to our approach. In Section 3, we introduce a motivating scenario, which is used to highlight the new additions. Section 4 presents a model-driven approach to guide the definition of the TOSCA semantic enhancements and artefacts to address our research questions. In Section 5, the extensions to TOSCA are detailed, while Section 6 details the contributions for supporting FaaS and other distributed execution paradigms. Section 7 includes details on the new TOSCA structures, which support optimization factors and placement constraints. Section 8 includes an evaluation of our approach, which is compared with one of the most prominent commercial offerings. Finally, we present a discussion of the results of this work and conclude the paper.

2. Related Work

The OASIS TOSCA standard [5] is based on the definition of a cloud application through the usage of templates or blueprints. There are several implementations (although none are officially endorsed), some of which support more features and are more actively maintained than others. Indicative examples include Alien4Cloud (also used by the Apache Brooklyn project) [6], Cloudify [7], and OpenTOSCA [8]. All of these implementations allow the definition of new node types, the generation of TOSCA deployment templates, and the orchestration of the deployment. They are designed for single deployments of a cloud application and do not incorporate any optimization capabilities. Moreover, the high level of detail in these TOSCA templates provides a complete view of the application, however imposes difficulties in terms of the comprehension of its overall structure. Conversely,

the description of the model of an application is difficult in these platforms, without first describing a complete proof-of-concept. For this reason, we introduce a clear distinction between an initial, modeling-oriented (and more abstract), “type-level” “flavor” and a final, “instance-level” flavor of TOSCA. Moreover, the blueprints that are generated by our type-level TOSCA generator are vendor-neutral and can be deployed on any (combination of) cloud(s) and edge resources. In doing so, we maintain consistency with the intent-based design, which is endorsed by TOSCA. This is an application design method combining the power of expression of policy-based approaches with the modeling power of steady-state approaches [15]. Our approach emphasizes the maintenance of relationships between components by defining appropriate TOSCA relationships and capabilities.

In Reference [16], Wurster et al. proposed an extension of OpenTOSCA to describe FaaS-based applications. In addition to purely FaaS-focused applications, their work is also valid for mixed architectures consisting of FaaS-based and VM-based solutions. Their modeling scheme is specifically applied on an Amazon Web Services (AWS)-Lambda-based application. Our solution can work alongside such a modeling approach, as it does not focus on the support of provider-specific components or services. The contribution of our work on FaaS support in TOSCA is a FaaS application abstraction based on general-purpose VMs and concretely defined relationships mainly targeted to the development of FaaS applications.

RADON [17] is another approach that is based on TOSCA. RADON extensively uses TOSCA inheritance to define abstract and derived concrete, deployable entities. Overall, RADON is considered closer to our concept of instance-level TOSCA, as it contains detailed information related to deployment parameters of particular cloud application types and serverless functions. Similar to our approach, the modeling specification of RADON allows the definition of a custom FaaS architecture. VM deployment is mentioned in the RADON reference technologies [18], however we did not find any reference for edge deployment. Without edge deployment support, the low cost and high data processing locality offered by edge nodes is impossible to exploit. Moreover, the appropriateness of particular edge nodes for particular fragments cannot be modeled.

CAMEL [19,20] is described as a domain-specific language (DSL), which enables dynamic and cross-cloud deployments. The authors support that while TOSCA and CAMEL are similar, the latter can also be used not only during design, but also at run time because it can specify the instances to be used. CAMEL relies on multiple specialized DSLs, each focusing on a particular aspect. It emphasizes the creation of UML-based metamodels, enriched with additional domain semantics. The models that are created are always synchronized with the actual topology that is deployed at the time. Both direct (manual) and programmatic access to these models is allowed, enabling self-adaptive cross-cloud applications. Furthermore, CAMEL has already been extended to support the specification of commercial FaaS services [9]. While CAMEL provides some advanced features and can already manage cross-cloud deployment and adaptivity, we have found significant aspects requiring improvement. First, no language features specifically target edge devices (for example to account for the volatility of the devices or the migration of components from the cloud to the edge). Thus, the topology can only partially be optimized to consider the benefits of fog computing. Also, FaaS services are modeled from the perspective of a FaaS framework consumer (i.e., user of already-existing commercial offerings such as AWS Lambda), rather than a FaaS framework designer (i.e., creator of any FaaS service).

Another significant modeling effort is the Open Cloud Computing Interface (OCCI), which according to [21] is a protocol and API for all kinds of management tasks. It is also stated that the main focus of OCCI is to create a remote management API for IaaS model-based services allowing for the development of interoperable tools for common tasks, including deployment, autonomic scaling, and monitoring. In Reference [22], the authors support the idea that the focus of OCCI is to provide a standardized API and that it does not define concepts to handle reusability, composability, and scalability. Conversely, TOSCA offers means to express reusability, composability, and scalability. These advantages

grant TOSCA a superiority in its modeling capabilities over OCCI. Moreover, TOSCA can be used alongside OCCI [22,23] to achieve full-standard-based deployments [23].

Terraform is a popular declarative language oriented towards cloud deployments, also supporting FaaS services, backed by open-source implementation. Different plugins exist to instantiate nodes on different cloud providers and interact with external services providing content delivery network (CDN) and domain name service (DNS) facilities using a unified syntax. All of these features grant versatility and robustness to Terraform. Pulumi (<https://www.pulumi.com>) is another open-source framework similar to Terraform, which provides the additional advantage of using programming languages to express cloud topologies rather than requiring the use of a specific cloud application language. As with Terraform, it is also capable of handling FaaS deployments. However, when considering model-driven deployments, Terraform and Pulumi present certain disadvantages compared to TOSCA. These disadvantages originate from the fact that while resources are properly declared, there are no language features offered that can be used to generalize the relationships among application components. Thus, no means are offered to (i) extract generic, reusable blueprints and (ii) optimize the deployment of components, taking into account any other dependent components. Additionally, the computing resources and their roles are very specific and detailed, which while providing a concrete view of the state of the deployment, also obstructs the higher-level understanding of the model of the cloud application. The use of edge devices is possible, but it requires manual configuration of the details of the topology, as we illustrate in Section 8. TOSCA, on the other hand, excels in its capability for modeling and abstraction of an application, while also being capable of specifying concrete actions that should be considered when instantiating a topology (through its workflows feature). In this work, we enhance the modeling capabilities of TOSCA, suggesting a set of new constructs to assist the representation of hybrid clouds, multi-clouds, edge-, VM-, and FaaS-based applications.

In addition, proprietary software systems such as Amazon Cloud Development Kit (Amazon CDK) have been developed, providing capabilities traditionally offered by Cloud DSLs. Using Amazon CDK, a DevOps can model the application directly from the integrated development environment (IDE) used and specify the requirements of the application using a preferred programming language (TypeScript, JavaScript, Python, Java, and C#/.Net are currently available). Although such an offering allows deep integration with the existing Amazon constructs and offers a good abstraction over the Amazon services that are used, it is nevertheless difficult to introduce it without the prior expertise of the DevOps with the specific technology products offered by Amazon. Furthermore, services can only be developed in connection with the AWS cloud computing provider, leading to vendor lock-in.

The vendor lock-in problem also applies to other DSL-based solutions, such as Azure Resource Manager, Google Cloud Compute, OpenStack Heat, and CloudFormation AWS templates. Although Heat templates strive to maintain compatibility with CloudFormation templates, these templates are not recognized as a global standard. Moreover, all of the templates cited above contain too many technical details that are associated with the solution offered by a particular vendor. Our approach, on the other hand, strives to simplify the model of the application, providing two views: a model view before the deployment of the application featuring the least amount of technical details and an instance view after the instantiation of the topology (type-level and instance-level TOSCA, respectively).

Wurster et al. [24] reviewed prominent deployment automation approaches to derive the essential deployment metamodel. The metamodel refers to a technology-independent baseline, containing the core parts of deployment automation technologies such as Chef, Puppet, Ansible, Kubernetes, and OpenStack Heat. The authors state in their work that the generated metamodel uses only a subset of the entities described in TOSCA. Approaches similar to the metamodel can further be used to introduce or map other technologies to the terminology of TOSCA, and vice versa.

Edge applications have attracted considerable research interest over recent years. Applications that use processing nodes at the edge of the network can attain considerably better performance for applications that are either response-time-sensitive or privacy-oriented or that aim to minimize energy consumption [25]. Among the 384 studies related to fog or edge computing analyzed in [3], we are aware of only three of these studies containing a proposal to model the handling of processing for fog resources. The study by van Lingen et al. [26] extended the YANG language [27] with support for fog nodes. We also follow a similar approach using TOSCA, which is directly aimed at cloud deployments and is already an OASIS standard. Noghabi et al. [28] worked on Steel, a high-level abstraction for the development and deployment of edge–cloud applications. Their work emphasizes the ability to migrate services from the cloud to the edge, and the ability to optimize the placement of services while respecting constraints. Mortazavi et al. [29] proposed CloudPath, a multitier computing framework, in which the location and rest path of a FaaS system running on fog resources were configured using web.xml Java deployment descriptors. Our semantic enhancements of the TOSCA standard can also support the definition of placement constraints, while we additionally allow the definition of conflicting optimization criteria. Since there are plenty of deployment automation tools built on top of it, extending TOSCA would make sense from the perspective of reusability, as the implied extensions to the TOSCA-based deployment tools should be manageable.

3. Motivating Scenario: Fog Surveillance Application

In this section, we introduce a motivating scenario to assist the reader in understanding our approach. Below, we describe an application deployment based on the need for a surveillance company to deploy a number of processing components in both the edge and the cloud. For our scenario, we assume that our test bench includes a number of Raspberry Pi devices equipped with cameras and connected to the Internet, ARM-based servers situated near the edge, and VM assets in public or private clouds, conforming to the budget allocation. The processing components—or “fragments”—that are considered in the scenario are described in Table 1.

Table 1. Fog surveillance application fragments.

Fragment Name	Description
VideoStreamer	The fragment is responsible for the transmission of video from the edge to the host of the VideoTranscoder fragment
VideoTranscoder	The fragment is responsible for changing the format of a video
FaceDetector	The fragment is responsible for face detection in a captured video scene
AudioCaptor	The fragment is responsible for continuously capturing audio
PercussionDetection	The fragment is responsible of detecting any captured percussion sounds and triggering the FaceDetector component
MultimediaManager	The fragment hosts various necessary assets to perform the detection of suspects and present alerts to a user of the platform

Hereafter, the term “processing component” will be used interchangeably with the term “fragment”, as we consider that the “whole” of an application consists of one or more fragments. Throughout this work, we assume that fragments are containerized, using the de facto standard for containerization, Docker.

For all fragments that can be executed on edge resources, we pursue the deployment on suitable edge hosts. However, if a fragment cannot be executed on edge resources (or none are available), we use deployment criteria to govern its deployment on one or more cloud providers. For the VideoStreamer and MultimediaManager fragments, the primary objective that should govern the cloud deployment is the reduction of latency. The secondary objective preferred is the usage of the AWS cloud provider, and finally the reduction of cost is considered an additional business goal. For the FaceDetector fragment,

the first priority is the usage of the AWS provider, as a stringent agreement was reached with the particular provider on the handling of sensitive data.

Since the VideoTranscoder, FaceDetector, and PercussionDetector fragments are assumed to perform stateless operations, all of them can be attached to a common FaaS Proxy, which will balance and redirect incoming requests appropriately.

The operating system for all fragments is defined to be Ubuntu Linux. In addition, the Google Cloud Compute provider must be excluded. Moreover, the budget available for cloud deployments is set as equal to €1000 and the time-frame for which it will be available is set to 1 month (720 h). Fine-grained optimization criteria (cost, distance, friendliness—explained in detail in Section 7.2) are set for the VideoStreamer, FaceDetector, and MultimediaManager fragments, according to the requirements of the fragments described. The overall optimization objective is cost reduction, through the reduction of fragment instances scheduled for execution in the cloud.

To illustrate the data flow dependencies, a deployment graph was created in a prototype UI, corresponding to a deployment using the above fragments. The application graph is shown in Figure 1. The arrows indicate (from right to left) that the MultimediaManager fragment depends on data from the VideoTranscoder, FaceDetector, and PercussionDetector fragments, which in turn depend on data from the VideoStreamer, VideoTranscoder, and AudioCaptor fragments, respectively.

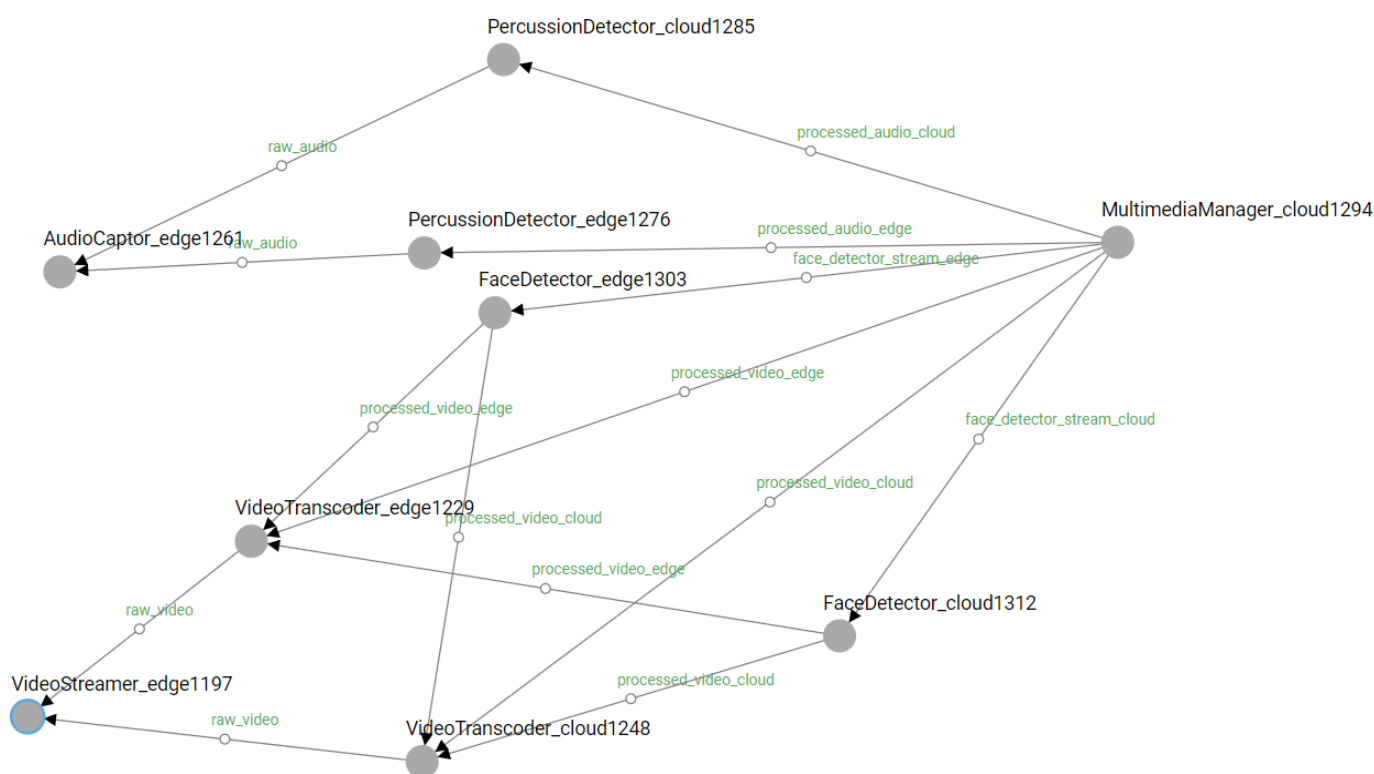


Figure 1. The deployment graph for the illustrative example. The green text on each line indicates the name of the interface between two components (relevant only for the user interface).

The requirements for each fragment are listed in Table 2.

Table 2. Fragment processing requirements and constraints.

Fragment Name	Hosting Requirements (CPU Cores–RAM GBs–Free Disk GBs)	Acceptable Hosting Resource	Processing Architectures	Collocation Dependencies/Anti-Affinity Requirements	Precedence Dependencies	Optimization Criteria ¹ (Cost–Distance–Friendliness)	Elasticity Mechanism
VideoStreamer	1-1-4	Edge	arm64, armel, armhf	VideoTranscoder (collocation)	-	2-8-{aws:5, gce:0, azure:1}	None
VideoTranscoder	2-4-4	Edge/Cloud	arm64, armel, armhf, x86_64, i386	VideoStreamer (collocation)	VideoStreamer	1-1-(1, implied)	FaaS (Lambda) Proxy
FaceDetector	1-1-4	Edge/Cloud	arm64, armel, armhf, x86_64, i386	-	VideoTranscoder	1-1-{aws:5, gce:0, azure:1}	FaaS (Lambda) Proxy
AudioCaptor	1-1-4	Edge	arm64, armel, armhf	PercussionDetector (collocation)	-	1-1-(1,implied)	None
PercussionDetector	1-1-4	Edge/Cloud	arm64, armel, armhf	AudioCaptor (collocation)	AudioCaptor	1-1-(1,implied)	FaaS (Lambda) Proxy
MultimediaManager	2-4-128	Cloud	x86_64, i386	-	FaceDetector, VideoTranscoder, PercussionDetector	2-8-{aws:5, gce:0, azure:1}	None

¹ The definitions for these optimization criteria are presented in Section 7.2.

4. Model-Driven Application Specification Using Extended TOSCA

In accordance with most efforts analyzed in Section 2, we follow a model-driven engineering approach because it offers portability and reusability, which are important considerations for cloud applications. Specifically, we propose an extension of the TOSCA modeling standard, adopting two views of the topology—type-level and instance-level TOSCA. Each of these is used to create a document reflecting the initial abstract view and the processed deployment view of the topology, respectively. We claim that the decoupling of the information contained in type-level TOSCA and instance-level TOSCA aids modeling and allows for the optimization of applications across the cloud computing continuum. These models are not bound to a specific provider, and given a TOSCA orchestrator can be used at any time.

Type-level TOSCA encapsulates the user's requirements, preferences, and business goals and provides a high-level overview of the topology to be deployed. This model—which is the main contribution of this work—can subsequently be optimized, finalized with provider-specific characteristics (e.g., network parameters), and deployed. During this process, the final “instance-level” model of the application is created. External monitoring mechanisms, e.g., Prometheus (<https://prometheus.io>), can be used to create an updated “type-level” deployment, which will trigger a reconfiguration of the platform. This workflow is in line with the challenges and research directions for cloud adaptations that were described in our previous paper [30].

The definition of a fog application (e.g., similar to the one depicted in Figure 1 can be seen in the steps presented in Figure 2.

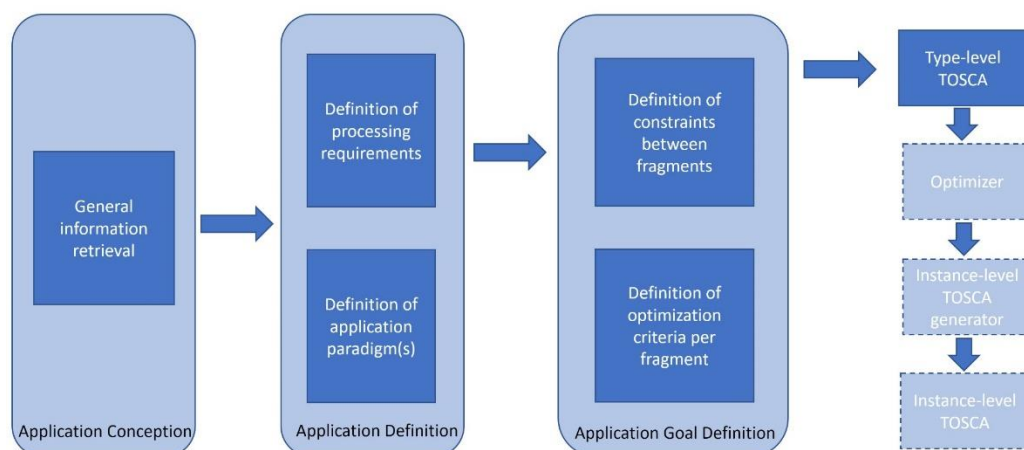


Figure 2. Overview of our approach. Although the optimization of the type-level TOSCA template and the generation of the instance-level TOSCA are essential constituents of our approach, we do not provide an implementation approach for an optimizer or an instance-level TOSCA.

To facilitate the use of our modeling extensions, we provide a tool [31] that aids in application definition and application goal definition as a proof-of-concept for our approach. Our tool was developed as part of the PrEstoCloud framework [14], and is able to parse the input requirements, optimization criteria, and constraints, generating type-level TOSCA as its output. In the following subsections, we provide more details for each of the steps illustrated in Figure 2.

4.1. Application Conception

In the application conception stage, the base for our modeling approach is established, as the DevOps structures the fragments that will comprise the application. Firstly, the architecture of the application is determined, as well as its input(s) and output(s) and the components that will be used in it. Following, the DevOps examines the fragments used in the application to determine whether there are common application paradigms that

are used in it (e.g., load-balancing or functions-as-a-service) and the number of instances necessary for the processing of each fragment. In addition, generic information related to the fragments themselves (e.g., a known functional configuration, as well as the interfaces that are provided and the interfaces that are needed) should be collected. Then, the DevOps should determine for each fragment the Docker image and Docker registry that should be used, the respective environmental variables, their port mappings, and their application paradigm (if any—see Section 6 for details). All of these attributes should later be mapped to the respective TOSCA constructs.

Finally, global constraints and preferences can be provided by the DevOps, specifying the providers that are preferred or should be excluded and the budget constraints that should be respected for this application.

4.2. Application Definition

During this step, the DevOps should enter more precise information describing the processing context of each of the fragments, as well as any application paradigm that is followed by the fragment. The information provided for the processing instances and the processing resources for each fragment are critical, as all further deployments and optimization steps will be primarily based on these factors. The DevOps is required to provide data for the fragment attributes, which are described in Table 3.

Table 3. Fragment attributes that should be defined by the DevOps.

Fragment Processing Attribute	Description
Number of cores	The minimum and maximum numbers of CPU cores needed to process the particular fragment
Processor architecture	The architecture of the CPU processor
Memory requirements	The minimum and maximum amounts of main memory required by a fragment
Disk capacity requirements	The minimum and maximum amounts of disk space required by a fragment
Sensors required	The input or output sensors required for the processing of this fragment
Operating system	The operating system and version that should be used for the processing of this fragment
Number of instances	The minimum and maximum numbers of instances of a particular fragment
Execution zone	The eligible locations for the processing of a particular fragment (cloud, edge)
Elasticity mechanism—paradigm	The elasticity mechanism that is required for a fragment implementing a particular paradigm (we propose extensions to TOSCA for FaaS, load-balanced, and JPPF applications)

The medium used to define these properties (as well as those mentioned in Section 4.3) is agnostic to our approach. However, our tool [31] supports both the usage of an external prototype UI, as well as code-level annotations coupled with a policy file. In the first case, informative application graphs similar to the one illustrated in Figure 1 can be retrieved and the input is provided to our component through graphical forms in a machine-readable json format. In the second case, annotations can capture details concerning the deployment requirements of each fragment, while the policy file contains global constraints and preferences that guide the deployment of the topology. The handling and specification of annotations is an integral part of our TOSCA generator, allowing the software to be used without depending on an external component. The main advantage of using an annotation-driven deployment lies in the potential to combine annotations with code introspection techniques to influence the behavior of the application. In Reference [32], information present in annotations was used to govern the addition of new workers and removal of existing workers or to modify their behavior.

An annotation is expected firstly to include information on the memory, CPU, and storage load that the particular application fragment will use, which is translated into a range of values using information from the policy file. The translation is currently performed using configurable static mappings (e.g., for CPU or memory, VERY_LOW = 1 core/1Gb of ram, LOW = 2 cores/2Gb of ram, MEDIUM = 4 cores/4Gb of ram, HIGH = 8 cores/8Gb of ram

and VERY_HIGH = 16 cores/16Gb of ram). Each annotation on the workload of a particular resource reflects the least amount of resources required to carry out the processing needed by the particular fragment.

Then, information on the possible processing zone should be provided—whether the fragment is onloadable (i.e., can be deployed on edge devices), offloadable (i.e., can be deployed on cloud VMs), or both. Afterward, information on the Docker configuration (here using the `edge_docker_registry` and `edge_docker_image` fields) of the component should be provided. Additionally, the application paradigm followed by the particular fragment can be specified using the `elasticity_mechanism` annotation. It should be noted that each of the fragments can follow a different paradigm and multiple paradigms may co-exist in the same application.

Our annotations also enable us to provide the minimum and maximum numbers of instances that should be provided for a fragment. While one may argue that it is difficult to enforce a particular number of instances on a volatile edge topology, we include this capability as it is a business requirement that is very common and that provides to the DevOps the capability to precisely define the requirements of the topology. Further, we assume that the information that is input by the DevOps in a type-level TOSCA template is not static but subject to (automatic) updates based on the state of the processing topology, as illustrated in our previous work [30].

Listing 1 contains an example annotation for a Java class representing the percussion detector fragment (it can either be a placeholder or its actual implementation), which is defined in the illustrative scenario. In the case of an annotation-driven deployment, one annotation needs to exist over each fragment that can be autonomously executed. As mentioned before, our tool can also accept input from a prototype user interface.

```
@PrestoFragmentation(
    memoryLoad = PrestoFragmentation.MemoryLoad.LOW,
    cpuLoad = PrestoFragmentation.CPULoad.VERY_LOW,
    storageLoad = PrestoFragmentation.StorageLoad.LOW,
    onloadable = true,
    offloadable = true,
    edge_docker_registry = "prestocloud.test.eu",
    edge_docker_image = "percussion_detector:latest",
    elasticity_mechanism = faas,
    min_instances = 1,
    max_instances = 5,
    dependencyOn = {"imu_fragments.AudioCaptor"},
    precededBy = {"imu_fragments.AudioCaptor"},
    optimization_cost_weight = 1,
    optimization_distance_weight = 1,
    optimization_providerFriendliness_weights = {"aws", "1",
        "gce", "1",
        "azure", "1"}
)
public class PercussionDetector {
}
```

Listing 1. Annotations example for the PercussionDetector fragment of the illustrative scenario.

Listing 1 also contains (for completeness) certain annotations that are not mapped to the requirements, which should be set for each fragment at this stage—these are discussed in Section 4.3.

4.3. Application Goal Definition

During this step—with the architecture of the application determined and the application components fully described—the DevOps can finally specify the application goals in terms of the optimization criteria that should be used for the deployment of the fragments, as well as any constraints that should be applied. The available fragment optimization criteria and constraints appear in Table 4.

Table 4. Fragment goals (optimization criteria and constraints) available to be defined by the DevOps.

Application Goal	Type	Definition Level
Precedence	Constraint	Defined for groups of two or more fragments
Collocation	Constraint	Defined for groups of two or more fragments
Anti-Affinity	Constraint	Defined for groups of two or more fragments
Distance	Optimization criterion	Defined for each fragment
Cost	Optimization criterion	Defined for each fragment
Friendliness	Optimization criterion	Defined for each fragment

If no constraints are chosen, the application deployment will be guided only by the automatic device exclusion constraint(s) (see Section 7.2), the optimization criteria specified for each fragment (where these exist), and the overall business goal that has been set for the application (e.g., the minimization of cost). If no optimization criteria are specified for one or more fragments, the application deployment will be performed based on the overall business goal, fulfilling any placement or precedence constraints. Naturally, the set of constraints and optimization values, which will be adopted for the fragments of the application, can lead to completely different deployments.

Using the example of the illustrative scenario, based on Table 2 we can determine that a collocation constraint is required between the VideoStreamer and VideoTranscoder fragments and the PercussionDetector and AudioCaptor fragments. Moreover, the prevalent optimization criterion that will govern the processing zone (cloud or edge) and processing location (provider data center or edge device) of the VideoStreamer instances will be “distance” (for more details see Section 7.2). For annotation-driven deployments (as in Listing 1), the precedence constraints are indicated using the precededBy annotation, while the collocation constraints are indicated using the dependsOn annotation. Individual optimization criteria are indicated using the optimization_cost_weight, optimization_distance_weight, and optimization_providerFriendliness_weights annotations, respectively.

As soon as the requirements of the application have been provided, the initial type-level TOSCA model should be created. This procedure is undertaken by a TOSCA generator (e.g., [31]) which receives the input gathered at the first stage of the processing and converts it to TOSCA format, producing a type-level model of the topology. We completed this process for the application described in Section 3 and created the type-level TOSCA model corresponding to it using a prototypical, open-source TOSCA generator (the full type-level document is included in Appendix A). This model file should then be sent to a TOSCA orchestrator—for example by uploading to a repository node, which will enable file artefacts to be communicated and stored.

4.4. Processing and Deployment of Requirements

During this step, the application requirements and structure that have been defined in the previous steps are received in the form of a type-level TOSCA template.

Then, a dedicated optimizer component (e.g., developed using Choco Solver [33] or BtrPlace [34]) is required to parse the received type-level TOSCA and solve the constraint programming problem associated with the optimization goals and placement constraints in it. The output of this process should include as a minimum the zone of the processing instances (cloud or edge), the provider(s) to be used, the optimal number of instances, and their flavor in a machine-readable format (e.g., json). Then, the network details

can be specified and the Docker environmental variables can be updated. Finally, the instance-level TOSCA file reflecting the deployment can be generated. We consider that the optimization process and instance-level TOSCA generation process should be triggered automatically once a new type-level TOSCA document is published to provide a fully automated deployment process.

4.5. TOSCA Fog Application Definition Algorithm

To aid the comprehension of our approach, we provide in Listing 2 the algorithmic steps that should be performed by a DevOps prior to the automatic generation of the TOSCA template.

```

Input
F:Fragments
FPA:Fragment processing attributes//Defined in Table 3
FG:Fragment goals//Defined in Table 4

Algorithm

//Application Concept Definition

AC ← determine_coarse_grained_application_constraints
F ← determine_application_fragments

for fragment in F
    fragment.determine_application_paradigm
    fragment.determine_interfaces
    fragment.determine_environmental_variables
    fragment.determine_docker_properties
    fragment.determine_necessary_number_of_processing_instances

//Application Definition
for fragment in F
    for processing_attribute in FPA
        fragment.assign_value(processing_attribute,DevOps_value)

//Application Goal Definition
for fragment in F
    for fragment_goal in FG
        fragment.assign_value(fragment_goal,DevOps_value)

type_level_TOSCA_document ← TOSCAgenerator(AC,F)
return type_level_TOSCA_document

```

Listing 2. Algorithmic steps necessary for the definition of a fog application in type-level TOSCA.

In the next sections, we will detail how each of the three research questions set in the introduction of our work was answered. First, we will discuss the type-level TOSCA semantic enhancements to support fog deployments, then we will elaborate on our approach to support additional distributed software paradigms, and thirdly we will detail the support provided for placement constraints and optimization criteria.

5. Type-Level TOSCA Semantic Enhancements

The official TOSCA specification [5] provides sample configurations of processing nodes and cloud application topologies, however no reference is made to fog topologies, which commonly need processing nodes both on the edge and the cloud. While a fog

topology could be implemented using TOSCA or any other DSL, an important issue that would not be solved would be the ability to use a different configuration for a fragment depending on its processing zone (cloud or edge), while still qualifying as the “same” fragment type for scaling purposes. A second challenge encountered when creating a service template is to accurately describe the model of the service while still allowing for its optimization.

To resolve the first issue and be able to seamlessly describe cloud-only, edge-only, and fog applications, we introduce a modeling schema, which decouples the software from the hardware it is installed on but still maintains their relationship. The second challenge is mainly tackled by the separation of concerns between type-level and instance-level TOSCA approaches.

Although the extensions of TOSCA involve numerous aspects of the deployment, the proposed changes to the language are non-intrusive and can be used within the original language features. This means that we do not modify the core logic of TOSCA templates and that our model maintains the traditional structure of a TOSCA application. A summary of these changes is presented in Table 5.

An important consideration related to the implemented improvements is the small learning curve of the type-level TOSCA model, which allows a DevOps to quickly become familiar with and inspect the structure of a cloud application.

Table 5. Overview of the extensions to core TOSCA concepts.

TOSCA Feature Extended	Summary of Changes	Indicative Extensions	Related Sections
Metadata	Introduction of new fields relating to optimization support	TimePeriod, CostThreshold	Section 7.1
Node_types	New node types are introduced to denote particular processing characteristics that are desired on a processing node	prestocloud.nodes.agent.faas, prestocloud.nodes.agent.loadBalanced	Section 5.1
Node_templates	New node templates are presented to allow Docker support, optimization support, as well as the expression of edge-related attributes and coordination paradigms	prestocloud.nodes.fragment.faas, prestocloud.nodes.fragment.jpff	Section 5.1 , Section 5.2
Policies	New policies are added to indicate the manner in which application deployment should be managed	prestocloud.placement.Gather, prestocloud.placement.Spread	Section 7.1
Relationships	New TOSCA relationships are developed to indicate the relationship of the processing between components	prestocloud.relationships.executedBy.faas, prestocloud.relationships.executedBy.loadBalanced	Section 5.1 , Section 5.2 , Section 6
Capabilities	New TOSCA capabilities are developed to indicate the special processing capabilities offered by some devices	prestocloud.capabilities.proxying.faas, prestocloud.capabilities.proxying.jpff	Section 5.1 , Section 5.2 , Section 6

5.1. Fragment and Processing Host Decoupling

A crucial aspect of our approach is the introduction of optimization capabilities to the TOSCA template. Unfortunately, as most of the existing approaches follow an analytical approach by specifying the topology in detail (i.e., specifying the provider to be used at the time of model formulation and the exact VM details), there is little room left for optimization. However, using our approach, the optimizer determines the exact processing zone and processing location. Furthermore, the number of fragment instances can be easily changed from components external to those involved in the TOSCA generation process (e.g., a topology scaling director—see Figure 2) without adding unnecessary complexity to our type-level model.

These advantages are only possible if a clear distinction between the software components of the application from the hardware that they are installed on is made using distinct TOSCA structures. Each application, therefore, consists of fragment nodes (reflecting the software components), which are each related to a processing node (reflecting the hosting hardware). Fragment nodes contain a description of a fragment, which will run independently within the context of the application, while the hardware and operating-system level requirements that are imposed by each fragment are modeled on so-called “processing nodes”. Each processing node defines a relevant TOSCA node type and each “fragment node” corresponds to an instance of a TOSCA node template. While processing nodes are generic (and could each be used by many fragments), fragment nodes are tightly coupled to the fragment that they describe—hence the naming of the nodes. Fragment nodes are mapped to processing nodes, using “mapping nodes” (minimal TOSCA node templates). The definitions of both processing and fragment nodes are based on the hosting requirements expressed either through the annotations mechanism or the UI. Processing, fragment, and mapping nodes themselves are each defined in a new TOSCA node type, respectively. An example of the relationships between processing, fragment, and mapping nodes is illustrated in Figure 3.

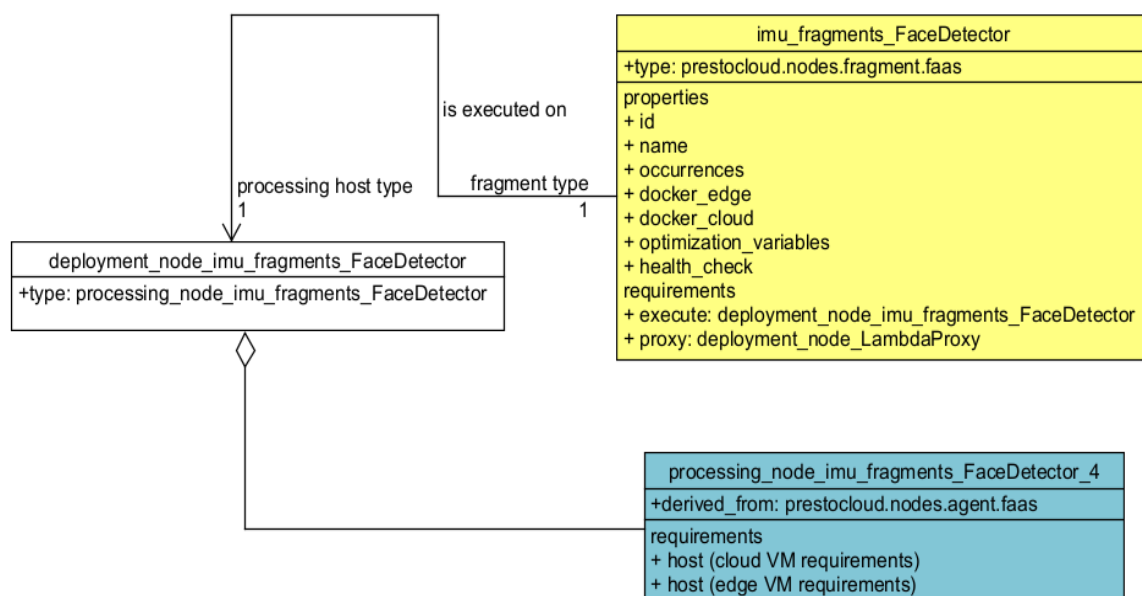


Figure 3. The relationships between fragment, processing, and mapping nodes.

An excerpt from the type-level TOSCA showing the sections related to the connection of application fragments (according to our motivation scenario of Section 3) with their respective processing nodes is shown in Listing 3.

```

deployment_node_imu_fragments_FaceDetector:
  type: processing_node_imu_fragments_FaceDetector_4

imu_fragments_FaceDetector:
  type: prestocloud.nodes.fragment.faas
  .....
  requirements:
    - execute: deployment_node_imu_fragments_FaceDetector
    - proxy: deployment_node_LambdaProxy

```

Listing 3. The connection of fragments with processing nodes.

In the example in Listing 3, the “deployment_node_imu_fragments_FaceDetector” is the mapping node, while the “imu_fragments_FaceDetector” is the fragment node. The “proxy” field of the fragment node indicates the mapping node that will host the Lambda Proxy of the topology (this node is not part of Listing 3—see Section 6 for more details). When the TOSCA blueprint is parsed, the TOSCA orchestrator will deploy a Lambda Proxy, which will enable the deployment of the FaceDetector fragment as a serverless function.

The processing node defining the type of mapping node (processing_node_imu_fragments_FaceDetector_4) that will be used to deploy the above fragment (imu_fragments_FaceDetector) is specified in Listing 4.

```

processing_node_imu_fragments_FaceDetector_4:
  description: A TOSCA representation of a processing node
  derived_from: prestocloud.nodes.agent
  requirements:
    - host:
      capability: tosca.capabilities.Container
      node: prestocloud.nodes.compute
      relationship: tosca.relationships.HostedOn
      node_filter:
        capabilities:
          - host:
            properties:
              - num_cpus: { in_range: [1, 2] }
              - mem_size: { in_range: [1024 MB, 2048 MB] }
              - storage_size: { in_range: [4 GB, 32 GB] }
          - os:
            properties:
              - architecture: { valid_values: [x86_64, i386] }
              - type: { equal: linux }
              - distribution: { equal: ubuntu }
        resource:
          properties:
            - type: { equal: cloud }
    - host:
      capability: tosca.capabilities.Container

```

Listing 4. Cont.

```

node: prestocloud.nodes.compute
relationship: tosca.relationships.HostedOn
node_filter:      capabilities:
  - host:
    properties:
      - num_cpus: { in_range: [1, 2] }
      - mem_size: { in_range: [1024 MB, 2048 MB] }
      - storage_size: { in_range: [4 GB, 32 GB] }
  - os:
    properties:
      - architecture: { valid_values: [ arm64, armel, armhf ] }
      - type: { equal: linux }
      - distribution: { equal: raspbian }
  - resource:
    properties:
      - type: { equal: edge }

```

Listing 4. The description of a processing node.

As can be noted in Listing 4, the processing node definition allows for different requirements for the cloud and the edge version of a fragment. This facility can be used to adjust the processing requirements on edge devices, to account for the disparity in performance between them and cloud instances.

Furthermore, in the case of fragments that should be only executed on edge devices, a further constraint can be specified to ensure that candidate devices possess the necessary sensors to acquire input. To represent this constraint, we introduced a sensors property on extended TOSCA compute node specification, which includes a list of all required sensors. Any device that does not possess one or more of these sensors is not eligible to host the particular fragment.

An example of the sensors property used to limit (using the TOSCA nodefilter structure) possible hosts to those possessing a microphone and camera is shown in Listing 5.

```

- sensors:
  properties:
    - microphone: { equal: "/dev/snd/mic0" }
    - camera: { equal: "/dev/video/camera0" }

```

Listing 5. Example of the “sensors” property.

5.2. TOSCA Specification of Fragment Nodes

Fragments are the central elements in our extended TOSCA document, as they contain the actual business logic that will be carried out by the cloud application. Fragments can represent software components that run on the edge, on the cloud, or both—using the same or a different set of properties. The connection of fragments with processing nodes and with their processing requirements is achieved using a new TOSCA relationship.

The definition of a fragment node begins with a declaration of the type to which it belongs. Immediately afterward follows the property segment, which begins with generic information on the id and the name of the fragment. This information is followed by the scalable and occurrences fields, which indicate if the fragment is scalable and the number of instances that are needed for it to operate, respectively. The specification of the number of occurrences at the fragment level drastically reduces the size of type-level TOSCA files (the alternative would be to copy large, identical specification blocks) and aids their comprehensibility. We consider that it is necessary to allow the DevOps to define the number of instances available to a fragment as a starting point to deploy the topology.

The major share of the fragment specification belongs to the definition of Docker properties, either for edge or cloud versions of the fragment, or both. These properties include the Docker image, the registry, the environmental variables, the specification of port forwarding within Docker, and a custom Docker command line that can be executed by the fragment (if needed). The ability to specify different properties on the cloud and edge versions of a fragment provides the means for applications to adapt their execution according to the resources of the host (generally inferior in an edge device compared to a cloud VM). This is exemplified in Listing 6, where lower precision is used for the edge version of FaceDetection and for more iterations in order to diminish the probability of an edge device becoming overloaded (it is presumed here that the face detection algorithm can either iterate more times using a coarse model to detect faces in the image or fewer times by running calculations with higher precision). On the other hand, the cloud version can operate at full effectiveness, requiring a much lower number of iterations to verify the result. Further, it is important to mention that environmental variables may be dynamic, using the `get_property` function available to TOSCA. Thus, IP addresses that are unknown at the time of the deployment of the fragment may be denoted by a variable, which is later replaced in instance-level TOSCA by an appropriate IP address. Following the specification of Docker properties, the optimization variables section contains the weights for the cost and distance criteria, while the friendliness criterion accepts a list of providers and the weight that is assigned to each of them. The optimization criteria for each fragment of the motivating scenario are specified in Table 2. For example, the DevOps has set for the VideoStreamer fragment a weight value of 8 for distance, a weight value of 2 for cost, a weight value of 1 for the friendliness of the “Azure” provider, and a weight value of 5 for the friendliness of the “AWS” provider (meaning that providers favoring low latency should be favored, then the AWS provider, then providers offering low cost, and then any provider—higher weights involve a higher preference for this criterion). This segment is concluded by the definition of a custom health check command line and an integer interval between two successive health check commands.

The last elements in the specification of a fragment are the mapping node that will execute the particular fragment and the coordinator node (in this case a Lambda proxy) that is related to the particular fragment. Coordinator nodes are discussed in depth in Section 6.

Listing 6 provides the full specifications for an application fragment.

```
imu_fragments_FaceDetector:
  type: prestocloud.nodes.fragment.faas
  properties:
    id: 3
    name: imu_fragments.FaceDetector
    scalable: true
    occurrences: 1
    docker_edge:
      image: "face_detector_edge:latest"
      registry: "local.prestocloud.test.eu"
      variables: { "PRECISION": "50", "ITERATIONS": "10" }
    docker_cloud:
      image: "face_detector_cloud:latest"
      registry: "prestocloud.test.eu"
      variables: { "PRECISION": "100", "ITERATIONS": "2" }
    optimization_variables:
```

Listing 6. *Cont.*

```

cost: 1
distance: 1
friendliness: { "aws": "5", "gce": "0", "azure": "1" }
health_check:
  interval: 1
  cmd: "curl health.prestocloud.test.eu FaceDetector"
requirements:
  - execute: deployment_node_imu_fragments_FaceDetector
  - proxy: deployment_node_LambdaProxy

```

Listing 6. Full application fragment specifications for the FaceDetector fragment.

5.3. Description of Instance-Level TOSCA

Although our contribution emphasizes the modeling capabilities offered by the introduction of type-level TOSCA, instance-level TOSCA is also a major asset for the reconfiguration of the application topology. As the instance-level document contains the processing zone selected for each fragment, this information can be consumed by components external to the generation of TOSCA templates to understand the mixture of fragment instances that were deployed on edge devices and cloud VM's and to improve the quality of the updated blueprint (e.g., by adding one more instance for a component that is chiefly deployed on edge devices, which typically have lower processing power).

The creation of the instance-level TOSCA document should be automatically triggered each time a new type-level document is produced. This involves as a first step the extraction of the information contained in the type-level TOSCA document by the optimizer. The cost constraints and the time interval included in type-level TOSCA are evaluated to create an average cost that is admissible for the topology. Then, the collocation and precedence constraints are evaluated, as well as the optimization preferences provided in each fragment, to determine the final deployment of the application (for more details on the optimization process see Section 7.2). It is clear that the policies segment contained in type-level TOSCA is not necessary in instance-level TOSCA, as the constraints included there are taken into account during the allocation of resources. If the optimizer can produce a valid configuration satisfying the constraints of the topology described in the type-level document, this configuration is sent to the instance-level TOSCA generator, which produces the final instance-level TOSCA document.

The average cost of the suggested deployment is evaluated against the maximum admissible cost threshold of the DevOps. If it is lower, it is admitted and the deployment can be implemented. The processing resources, networking configuration details, as well as the cost of the VM or edge device used are then added for each fragment to new "node_type" definitions of the processing nodes that will host them. The TOSCA type of each processing node is different to represent different edge devices and cloud providers. Thus, provider-specific information can be abstracted in the definition of certain normative TOSCA "provider types".

However, the instance-level TOSCA template shares with the type-level TOSCA template the definitions of fragments, as well as the relationships between fragment nodes and mapping nodes.

An example description of a processing node in instance-level TOSCA is described in Listing 7.

```

processing_node_fragments_FaceDetector_1:
  type: prestocloud.nodes.compute.cloud.amazon
  properties:
    type: cloud
  network:
    network_id: s-gbdpnc4s
    network_name: subnet1
    addresses:
      - 192.168.1.1
  capabilities:
    resource:
      properties:
        type: cloud
      cloud:
        cloud_name: amazon_public1
        cloud_type: amazon
        cloud_region: us-east-1
  host:
    properties:
      num_cpus: 2
      mem_size: 4.0 GB
      disk_size: 50 GB
      price: 0.120000

```

Listing 7. Processing node specifications for instance-level TOSCA.

In instance-level processing nodes, information described in the form of constraints in type-level TOSCA should be concretized into specific details. The number of CPUs, the memory, and the disk size are all fixed values; the cloud provider and the cloud region are also chosen. These fixed values come from the solving process of the optimizer (see Figure 2), which considers the available hosting candidates with respect to the pre-defined optimization goals, as detailed in Section 7.2. Moreover, networking information is available for the particular instance. The information included in instance-level TOSCA processing nodes can also be customized by editing the relevant TOSCA type.

6. Support for FaaS and Other Coordinator-Based Paradigms

The new node types proposed for TOSCA both allow the representation of FaaS and other distributed software paradigms, in which there is a coordinator of execution that handles a number of workers. It is assumed that these distributed software paradigms are followed by one or more of the application components. We propose their definition here as a means to provide a reusable type of collection for certain common application patterns, which can also be exploited by other TOSCA applications.

FaaS-based applications are assumed to consist of a set of application fragments that are independent of other application components and have a self-contained execution flow. Fragments are assumed to be hosted inside Docker containers, which are in turn hosted inside VMs. Access to fragments is allowed through REST calls, which are managed by a publicly-facing load-balancer component. If more than one fragment type are managed by the load-balancer, the component is referred to as a Lambda Proxy, since it serves as a proxy for AWS-Lambda-like, serverless functions. Unlike some serverless platforms, which limit the processing time and the languages that can be used to develop functions, our approach supports all fragment types that can be dockerized, running for any desired amount of time.

Fragments following the FaaS paradigm use the custom `prestocloud.nodes.fragment.faaS` fragment type. The “proxy” field of the fragment type accepts the name of the Lambda Proxy

mapping node that will manage requests to this fragment. FaaS fragments are installed on FaaS agents (workers), which are modeled using the `prestocloud.nodes.agent.faaS` type. FaaS agents satisfy by definition the `prestocloud.relationships.executedBy.faaS` relationship required by FaaS fragments. FaaS Lambda Proxies are modeled with the `prestocloud.nodes.proxy.faaS` type and possess the `prestocloud.capabilities.proxying.faaS` capability, which allows them to coordinate worker agents hosting a FaaS fragment. The relationships between FaaS fragments, their executing processing nodes, and the Lambda Proxy can be seen in Figure 4.

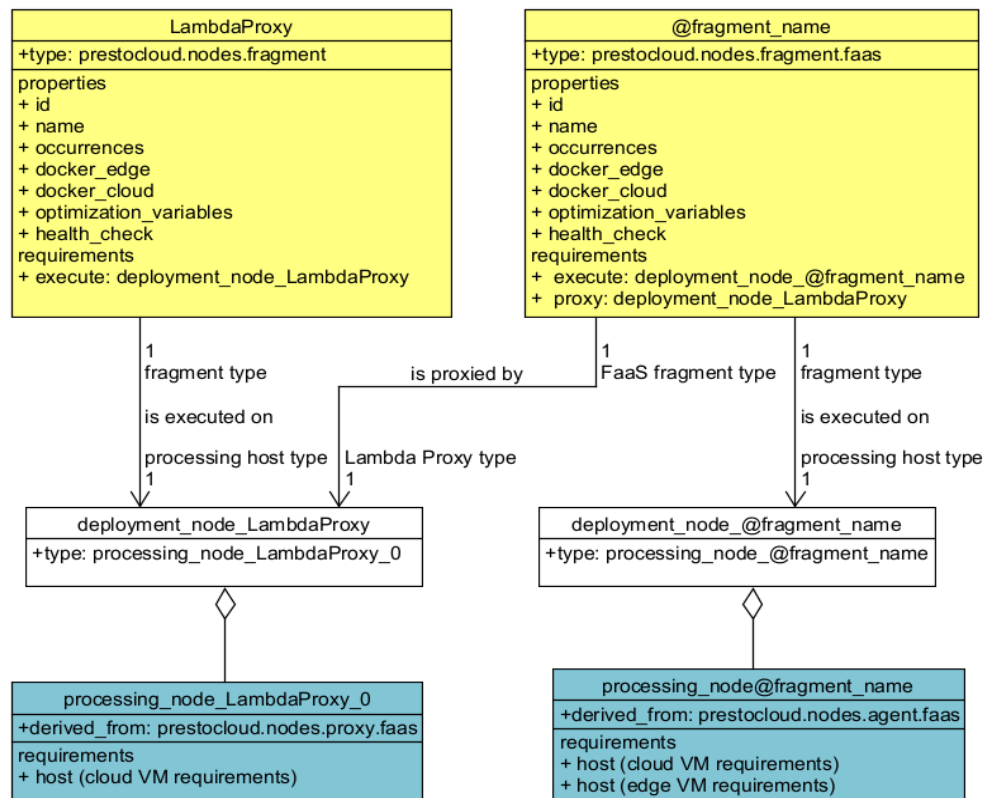


Figure 4. The relationships between the fragment, processing, and mapping nodes of the Lambda Proxy and the proxied FaaS fragments.

We also propose a similar modeling approach for applications using the load-balancing paradigm. In this paradigm, we consider that the application fragment can run in parallel on a number of workers, which each host an instance of the fragment and can handle a fraction of its workload. This is achieved with the help of a load-balancing node, which receives all incoming HTTP REST calls and invokes the correct worker. As is the case for FaaS-based applications, fragments are assumed to be deployed in Docker containers, while the container of each fragment is deployed on its own VM. The custom type that is created for load-balanced fragments is `prestocloud.nodes.fragment.loadBalanced`. Load-balanced fragments are installed on `prestocloud.nodes.agent.loadBalanced` agents (workers), which by definition satisfy the `prestocloud.relationships.executedBy.faaS` relationship. Workers are load-balanced by the Load-Balancer component, which is modeled by the `prestocloud.nodes.proxy` and possesses the `prestocloud.capabilities.proxying` capability. The connection of a worker to its load-balancer is achieved with the help of the “proxy” field in the `prestocloud.nodes.fragment.loadBalanced` TOSCA. The relationships between load-balanced fragments, their executing processing nodes, and the Load-Balancer can be seen in Figure 5 below.

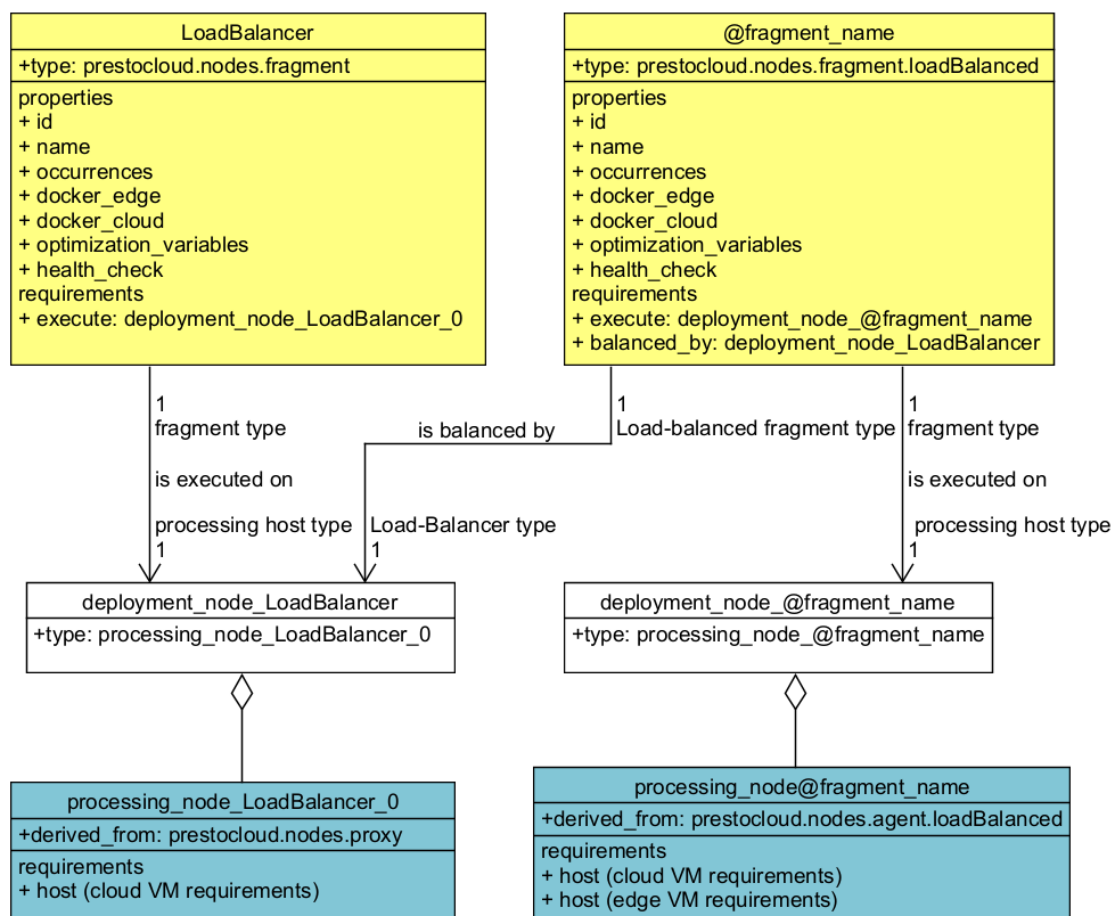


Figure 5. The relationships between the fragment, processing, and mapping nodes of the Load-Balancer and the balanced fragments.

Specification of JPPF applications is also supported by our approach. JPPF allows a Client to offload a processing task to a JPPF Agent with the help of a suitable coordinator (JPPF Master) node. Any JPPF fragment can be modeled using the `prestocloud.nodes.fragment.jppf` type and is installed on a JPPF Agent that uses the `prestocloud.nodes.agent.jppf` type, which by definition satisfies the `prestocloud.relationships.executedBy.jppf` relationship required by JPPF fragments. JPPF Masters are modeled using the `prestocloud.nodes.jppf.master` type and have the `prestocloud.capabilities.endpoint.jppf` capability, which indicates that they provide an endpoint that JPPF Agents can connect to in order to retrieve processing tasks.

The relationships between the JPPF Clients, their executing processing nodes, and the JPPF Master can be seen in Figure 6 below.

In our approach, more than one of these software paradigms can co-exist in the same application and each functions independently from the others. The relationships between coordinator, processing, and fragment nodes within an application topology are depicted in Figure 7.

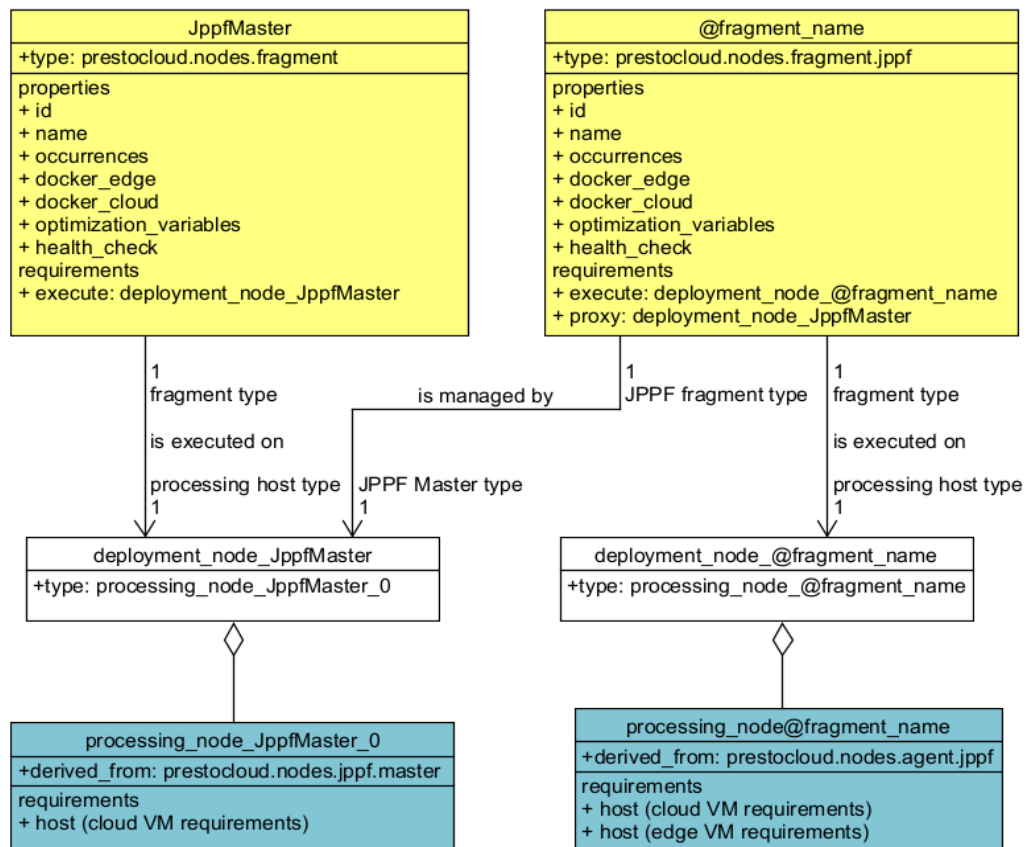


Figure 6. The relationships between the fragment, processing, and mapping nodes of the JPPF master and the JPPF tasks (fragments).

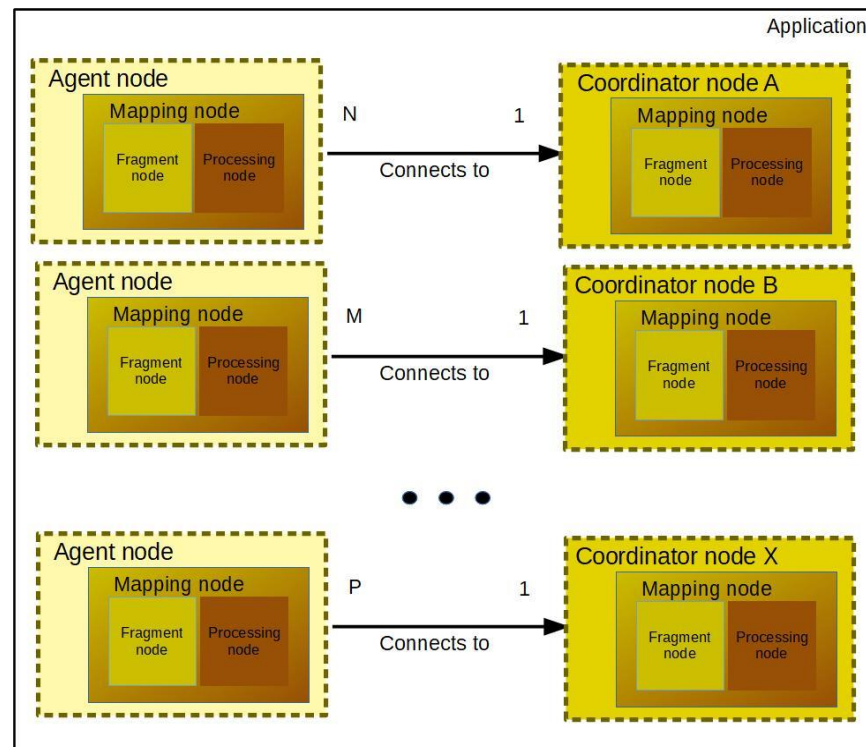


Figure 7. The structure of the topology template segment in a coordinator-driven, type-level TOSCA blueprint. Multiple agent nodes may be connected to one coordinator node and there can be an arbitrary number of coordinator nodes.

7. Optimization and Application Constraints

The TOSCA extensions that are introduced include support for the optimization of the deployment of fragments. Type-level TOSCA processing nodes specify ranges of satisfactory values for most of their attributes (e.g., cpu cores, available ram and disk space). This permits a reasonable number of alternative providers to be researched for the availability of similar VMs (in resources), while ensuring a minimum performance standard. The final selection of provider resources and edge devices should obey the coarse-grained and fine-grained constraints set for the application, as well as any placement policies set for one or more fragments.

7.1. Coarse-Grained Application Constraints

The vanilla TOSCA language specification [5] already permits the definition of a metadata segment inside the TOSCA file. In our approach, this native construct is used to hold extended constraints and preferences of the application. Its key-value body contains: the data relevant to the business goals pursued by the application; the providers that are preferred or excluded; and the budget constraints that should be respected. These constraints can be used by the optimization engine to select the most appropriate flavor and locations for the processing nodes.

An example of the usage of metadata fields to denote some of the constraints outlined above is presented in Listing 8:

```
metadata:
  template_name: IMU generated types definition
  template_author: IMU
  template_version: 1.0.0-SNAPSHOT
  CostThreshold: 1000
  TimePeriod: 720
  ProviderName_0: OpenStack_local
  ProviderRequired_0: false
  ProviderExcluded_0: true
  MetricToMinimize: Cost
```

Listing 8. Example of metadata fields used for application-level constraints.

In the above example, aside from some generic informative fields concerning the particular template version, the name, and the author, definitions exist for the business goals, provider, and budget requirements related to the application as a whole. The budget available is set to 1000 monetary units (e.g., euros), which should be used over a time period of 720 h. The “OpenStack_local” provider is set to be excluded, and the primary objective to be minimized is set to “cost”.

7.2. Fine-Grained Constraints and Optimization Criteria

As mentioned in Section 5.2, in each fragment definition, a number of optimization criteria are identified, namely the cost, distance, and friendliness. The cost optimization criterion reflects the monetary cost of choosing a particular hosting VM for a time period. The distance optimization criterion reflects the distance of the host of the fragment from the centroid of edge devices. The friendliness criterion reflects a preference towards a particular cloud provider (for any reason, e.g., data locality). While these criteria are modeled in a specific construct, more optimization criteria can of course be defined to supplement or replace the above. An example of the specification of optimization criteria appears in Listing 6.

Additionally, these application constraints can be coupled with a set of tools that enables the DevOps to guide the deployment of fragments by considering their relationships. We suggest implementing these using a set of proper placement optimization policies. In the context of TOSCA, we introduce three different optimization policies: collocation policies,

anti-affinity policies, and precedence policies. The formal definitions of these policies appear in Listing 9.

<p><u>Input</u></p> <p>x: Cloud providers</p> <p>f: Fragments</p> <p>d: Devices</p> <p>Provider: $f \rightarrow x$ Function mapping from fragments to providers</p> <p>Hosting: $d \rightarrow f$ Function mapping from hosting (edge) devices to fragments</p> <p>DeploymentTime: $f \rightarrow \mathbb{R}^+$ Function mapping from fragments to the positive real numbers</p> <p><u>Policy definitions</u></p> <p>Collocated (f_i, f_j) \rightarrow Provider(f_i) = Provider (f_j)</p> <p>Collocated($f_i, f_j, f_k, \dots, f_{n-1}, f_n$) = Collocated($f_i, f_j$) and Collocated($f_i, f_k$) and ... and Collocated(f_i, f_n)</p> <p>and Collocated(f_j, f_k) and...and Collocated(f_j, f_n) and ... and Collocated(f_{n-1}, f_n)</p> <p>Antiaffinity(f_i, f_j) \rightarrow Provider(f_i) \neq Provider(f_j)</p> <p>Antiaffinity ($f_i, f_j, f_k, \dots, f_{n-1}, f_n$) = Antiaffinity($f_i, f_j$) and Antiaffinity(f_i, f_k) and ... and Antiaffinity(f_i, f_n) and Antiaffinity(f_j, f_k) and ... and Antiaffinity(f_j, f_n) and ... and Antiaffinity(f_{n-1}, f_n)</p> <p>Precedence(f_i, f_j) \rightarrow DeploymentTime(f_i) < DeploymentTime(f_j)</p> <p>Precedence(f_i, f_j, \dots, f_n) \rightarrow DeploymentTime(f_i) < DeploymentTime(f_j) < ... < DeploymentTime(f_n)</p> <p>Excluded($f_i, (d_1, d_2, \dots, d_n)$) \rightarrow not Hosting(d_1, f_i) and not Hosting(d_2, f_i) and ... and not Hosting(d_n, f_i)</p>

Listing 9. Formal definitions of optimization policies.

Our collocation policies indicate that a fragment should be collocated with other fragments (using the same cloud provider), unlike the collocation policies that are briefly mentioned in the TOSCA specifications, which imply the use of the same compute node. This allows low-latency communication and results in improved compatibility and communication between processing nodes. However, the optimization component cannot consider the option of using different cloud providers for the fragments to lower the total costs. In the motivating example, a collocation policy is needed for the VideoStreamer and VideoTranscoder fragments (Table 2).

Our anti-affinity policies specify that a fragment should not be collocated with other fragments. This results in the placement of this fragment and all target fragments in different cloud providers (and is, thus, different from the anti-collocation policies briefly mentioned in the TOSCA specifications). Using this policy enhances the security of a critical information system that is communicating with a potentially vulnerable component, as it is

easier to isolate systems in case of a breach. However, this also means that the optimization component cannot request instances from the same provider for the fragments, and as a result some of the lower-cost options might be lost. In the motivating example (Table 2), an anti-affinity policy is needed for the AudioCaptor and PercussionDetector to ensure that (violent percussion) detection happens reliably and quickly (i.e., away from edge nodes on which AudioCaptor fragments are hosted).

Precedence policies describe that fragments should be instantiated and deployed in the order that is mentioned. An advantage of precedence policies is that all required interfaces—indicating data needed from a data flow for each component—are automatically satisfied by the time they are instantiated. Precedence policies guarantee the satisfaction of interfaces, however increased deployment time is required in return, as Docker containers should be spawned sequentially.

Device exclusion policies ensure that fragments are optimally scheduled for processing at the edge. They enhance the response of the system by marking a certain set of devices as unsuitable for deployment—therefore being excluded from the scheduling of instances of a particular fragment. The suitability of a device for a fragment depends on historical data processing results and the availability status, which can be detected and analyzed using machine learning techniques (for more details see [35]). The details of such a component are not detailed here, since this is considered out of the scope of this article.

The placement policies are modeled at the level of TOSCA using the fragment nodes. They offer a significant benefit over the usage of native TOSCA relationships, in that they permit the easy visualization of the most important constraints associated with the deployment application. In addition, the implementation of their enforcement is much simpler than the resolution of TOSCA relationships between fragments.

An example of the four deployment policies based on the motivating example is included in Listing 10.

```

topology_template:
  policies:
    - collocation_policy_group_0:
      type: prestocloud.placement.Gather
      targets: [ imu_fragments_VideoStreamer, imu_fragments_VideoTranscoder ]

    - anti_affinity_policy_group_0:
      type: prestocloud.placement.Spread
      targets: [ imu_fragments_PercussionDetector, imu_fragments_AudioCaptor ]

    - precedence_policy_group_0:
      type: prestocloud.placement.Precedence
      targets:
        [ imu_fragments_VideoStreamer, imu_fragments_VideoTranscoder, imu_fragments_Face
        Detector, imu_fragments_MultimediaManager, imu_fragments_AudioCaptor, imu_fragments_PercussionDetector ]

    - exclude_fragment_from_devices_0:
      type: prestocloud.placement.Ban
      properties:
        excluded_devices:
          - "a6f2:d8bd:bf45a:de2a:d1e8:5f58:c256:0492",
          - "c2c1:def1:2c38:c83b:6b0d:b7bd:a0d2:95c2",
          - "b3ef:58d8:39d0:86ce:81d2:6e93:5f7d:23cd",
          - "b28e:9f32:2076:3599:39c3:6cc5:794a:5140"
      targets: [ imu_fragments_VideoStreamer ]

```

Listing 10. Full example of the available deployment policies.

7.3. Constraints and Optimization Handling

The enforcement of the optimization policies presented in this work is delegated to the optimization engine that consumes the type-level TOSCA. Additionally, in order to provide a clearer understanding of the effects of each optimization policy, we describe the necessary steps to be performed by any optimizer implementation process.

We consider that the optimizer first retrieves the available cloud provider VM types and edge devices that can be used for the deployment. Then, any device exclusion policies are applied and the excluded edge devices are removed from the candidate hosts. If there are not any contradictory policies (e.g., an anti-affinity policy and a collocation policy set for the same set of fragments, or cyclic precedence policies), a list of valid configurations that satisfy all collocation and anti-affinity policies is proposed; otherwise, the list of valid configurations is set as empty. In the creation of valid configurations, precedence is given to the assignment of fragment instances to edge processing hosts satisfying the requirements of a fragment. If the list of valid configurations is not empty, the configurations are sorted according to the optimization criteria that have been defined, and the best configuration that satisfies the global constraints should be chosen to be translated to instance-level TOSCA.

Otherwise, if no configuration is found to satisfy the constraints and deploy all fragment instances, the deployment fails and the DevOps should resubmit a new template. Finally, at deployment time, the containers of different fragments should be started according to the priority, which is set in one or more precedence policies.

The process described above appears in pseudocode in Listing 11. Note that we intentionally do not proceed with optimization of the data structures and algorithmic logic, as we aim to provide an easy-to-follow overview of our proposal.

Input

X:Cloud providers

F:Fragments

D:Candidate Edge Devices

CLP:collocation policies

AAP:anti-affinity policies

PRP:precedence policies

DEP:device exclusion policies

Algorithm

for fragment **in** F

for collocation-policy **in** fragment_CLP

for anti-affinity policy **in** fragment_AAP

if collocation-policy **contradicts** anti-affinity-policy **then** return
 FAILED_DEPLOYMENT

Listing 11. Cont.


```

for precedence-policy in fragment_PRP

  for other-precedence-policy in fragment_PRP

    if precedence-policy contradicts other-precedence-policy then return
    FAILED_DEPLOYMENT

  for fragment in F

    for device in fragment_DEP

      D ← D-{device} //remove the device from the eligible hosts

    Configurations ← find_eligible_configurations (F,D,X)

    Configurations ← apply_coarse_grained_application_constraints (Configurations)

    if Configurations is not Empty

      maximum_utility ← -∞

      best_configuration ← None

      for configuration in Configurations

        configuration.utility ← calculate_utility_from_optimization_criteria
        (configuration)

        if configuration.utility > maximum_utility then

          maximum_utility ← configuration.utility

          best_configuration ← configuration

      return best_configuration

    else return FAILED_DEPLOYMENT

```

Listing 11. Optimization process pseudocode.

8. Evaluation

8.1. Comparative Assessment

In order to evaluate our approach, we illustrate a comparison of our extended TOSCA with one of the most prominent commercial solutions, Terraform, as the most representative of the other approaches in terms of features that are offered. Following this, we provide a definition of the example application presented in Section 3 with Terraform and highlight the advantages of both approaches. We consider that the application deployment should try to respect the optimization criteria specified in Table 2.

The first question that should be answered concerns the choice of the cloud provider that should be used. To create the Terraform template, we assume that the DevOps of the application invests a thorough amount of time balancing the pros and cons of deployment on a particular cloud provider, while also factoring in the requirements of a particular fragment in order to choose the VMs that have the lowest price for a satisfying deployment. This process is difficult, error-prone, and time-consuming. On the other hand, our ap-

proach depends on an initial investment of time to create a component able to solve a constraint programming problem. Subsequently, the optimizer component will be able to automatically evaluate the available cloud offerings and provide the most appropriate processing location for each fragment instance.

Firstly, we consider the creation of the topology template of our fog application using Terraform. A major disadvantage of Terraform—and the majority of the approaches listed in Table 6—is that the mixture of edge and cloud devices should be known beforehand and be static in order to accurately describe the topology. However, the assumption of a static topology is very difficult to make, as it requires considerable expertise on the offerings of cloud providers, while more importantly the available edge resources are opportunistic. On the other hand, if the topology is not static (and a tool similar to Kubernetes is used to abstract this), then there is a risk the instances will be deployed on a possibly suboptimal location (in the case that the cluster consists of both edge devices and cloud VMs, a component may be assigned to a VM when an edge device could support it). Furthermore, in the case that two processing clusters are used, one for edge devices and one for cloud VMs, it is possible that the topology will not be deployable at all (for example, if a component is set to be deployed on the edge cluster but there are insufficient resources in it).

For our comparison, let us assume that aside from the public cloud resources, the application can also use Raspberry Pi devices to host application fragments.

Based on these requirements, a simplified topology template (omitting most of the network-related details), which can be created using Terraform, appears in Appendix B. Listing 12 contains excerpts from this template, which will help us to perform a comparison with our approach.

```

provider "aws" {
  profile = "default"
  region = "us-east-1"
}
# Network configuration ...
resource "aws_instance" "FaceDetector" {
  ami = "ami-2757f631"
  instance_type = "t2.micro"
  key_name = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id = "${aws_subnet.default.id}"
  depends_on = [aws_instance.VideoTranscoder]
}
# Other cloud components...
resource "aws_instance" "MultimediaManager" {
  ami = "ami-2757f611"
  instance_type = "t3a.medium"
  key_name = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id = "${aws_subnet.default.id}"
  depends_on =
[aws_instance.FaceDetector,aws_instance.VideoTranscoder,docker_container.percussion
_detector]
}
# Configure the Docker providers
provider "docker" {

```

Listing 12. *Cont.*

```

    host = "tcp://192.168.1.2:2375/"
  }
  provider "docker" {
    alias = "worker_2"
    host = "tcp://192.168.1.3:2375/"
  }
  provider "docker" {
    alias = "worker_3"
    host = "tcp://192.168.1.4:2375/"
  }
  # Create a container
  resource "docker_container" "video_streamer" {
    image = docker_image.vs_image.latest
    name = "vs_cont"
  }
  resource "docker_container" "audio_captor" {
    provider = docker.worker_2
    image = docker_image.ac_image.latest
    name = "ac_cont"
  }
  resource "docker_container" "percussion_detector" {
    provider = docker.worker_3
    image = docker_image.pd_image.latest
    name = "pd_cont"
    depends_on = [docker_container.audio_captor]
  }
  resource "docker_image" "vs_image" {
    name = "video_streamer:latest"
  }
  resource "docker_image" "ac_image" {
    provider = docker.worker_2
    name = "audio_captor:latest"
  }
  resource "docker_image" "pd_image" {
    provider = docker.worker_3
    name = "percussion_detector:latest"
  }
}

```

Listing 12. Sample deployment using a Terraform template.

Table 6. Notable cloud application deployment approaches.

Name	Type	Abstract Model View	Instance Model View	Multi-Cloud Topology Support	Cloud and Edge Modeling	Semantic Enhancements	Optimization Readiness	FaaS Support	Comments
Cloudify	DSL (TOSCA-Based)	No	Yes	Yes	Partial	No	No	AWS Lambda	-
Alien4Cloud	DSL (TOSCA-Based)	No	Yes	No	Partial	No	No	No	Partial edge deployment support could be implemented using the concept of “bring your own node” (BYON) for hosting applications
OpenTOSCA	DSL (TOSCA-Based)	No	Yes	Yes	Partial	No	No	Yes	FaaS support can be implemented as in [16]
CAMEL	DSL	Yes	Yes	Yes	Partial	No	Yes	Yes	Multi-DSL language built for multi-clouds deployment and recently extended for FaaS support
OCCI	API	No	No	No	Partial	No	No	No	Only few providers are actively backed by an OCCI implementation
Provider-specific languages/tools	DSL	No	Yes	No	Provider-Dependent	Provider-Dependent	No	Provider-Dependent	e.g., OpenStack Heat, Azure Resource Manager, etc.
Terraform	DSL	No	Yes	Yes	Partial	No	No	Yes	Partial edge deployment support could be manually implemented using a Docker provider
Pulumi	Programming-language-based	No	Yes	Yes	Partial	No	No	Yes	Partial edge deployment support could be manually implemented using a Docker provider
Our approach	DSL	Yes	Yes	Yes	Full	Yes	Yes	Yes	Modeling of a custom FaaS architecture is possible

Similar to our approach, this template can be used for repeated deployments of the application relieving the DevOps from the need to manually provision new processing nodes and instantiate software on them. Additionally, Terraform (and other similar approaches) provides a mature, industry-backed, domain-specific language—unlike our approach, which is based on a well-recognized standard but is a research effort. Furthermore, Terraform allows the use of variables, which is not yet exploited in our approach. As a result, components that are used in one use-case can also be used in a similar but different setting by changing only a few values (for example by changing a variable holding the deployment region for a resource or changing a variable holding the Amazon Machine Image (AMI) that will be used by some resources).

However, the explicit nature of Terraform templates also means that they are not easily adaptable. It is also difficult to define relationships between components, which is a native characteristic of TOSCA. As a result, the detailed Terraform templates need to be cautiously inspected to reveal any possible relationships between components. In Terraform, expressing software architecture paradigms in a provider-independent manner is a rather difficult task, as the only relevant tool that can be used is the “depends_on” statement. In contrast, building on the ability of TOSCA to create relationships and capabilities, our approach renders a template that is much simpler and easier to understand (see Appendix A). This is especially relevant in the cases of JPPF-based, load-balanced, function-as-a-service applications.

Continuing this evaluation, we consider that the topology should be adapted due to increased load and that two more processing nodes are required for the VideoTranscoder fragment. This leads to a need for a template update to depict the two new nodes that should be added to the topology. However, where should these nodes be physically instantiated? Even if we make the unrealistic assumption that the DevOps can know the most appropriate cloud site (e.g., in terms of cost and performance), choosing a cloud-based VM may be a suboptimal solution if one or more edge devices could handle the processing of a fragment. Thus, the DevOps should be additionally burdened with the knowledge of all edge devices that are available for processing if any degree of optimization is sought. Clearly, while this approach is inefficient with a small number of devices, it is totally inapplicable when a large number of edge devices are used. The same argument applies to the knowledge of all VM instance types offered by the cloud vendors. Even if we consider that automated helper services are used (e.g., a script calculating a DevOps-defined utility value over all nodes, also providing the best edge candidate nodes), the final confirmation of the DevOps will be needed for any reconfiguration of the platform, which is impractical if we consider large-scale applications. Even if we only consider small-scale applications, there will always be a “man-in-the-loop”, devoting non-negligible amounts of time and effort to implementing topology adaptation actions. In addition, while the handling of collocation and anti-affinity constraints is tedious in a processing topology with a large number of fragments, it is impossible when the optimization of costs is also sought. Our approach, however, paves the way for the usage of multi-objective scheduling based on approaches similar to those in [36–38], which can handle multiple conflicting optimization criteria that should be implemented by the optimizer component.

The current state-of-the-art in cloud application deployment is summarized in Table 6, listing the most prominent approaches. The first column contains the names of each approach, while the second column contains the type, which can be a programming framework, a domain-specific-language (DSL), or an application programming interface (API). The third and fourth columns discuss the availability of an abstract and an instance model view, which allow an overview of the application and a more precise view of the topology, respectively. Our TOSCA-based approach is the only one aside from the CAMEL-based approach providing a type-level model and instance-level model. Unlike CAMEL, however, we also support the modeling of execution on edge devices. The fifth column discusses the ability of the approach to deploy a topology utilizing multiple clouds. The sixth column indicates whether certain steps have been taken by approaches to support the

modeling topologies using both the cloud and edge. In this context, if a documented methodology to handle edge devices as part of the native language, API, or programming facilities (alongside cloud VMs) is natively offered by an approach, it is considered to fully support cloud and edge deployments. On the other hand, approaches that allow cloud deployment and permit deployment on edge devices (although with manual modeling steps or limited optimization opportunities) are considered to offer partial support for cloud and edge deployments. The seventh column indicates whether the approach has the semantic enhancements required to represent cloud-only, edge-only, and hybrid edge-cloud fragments in a unified way. The eighth column indicates the ability to support the definition of optimization criteria. The ninth column indicates whether there is support for the representation of serverless functions. While the support of commercially available function-as-a-service platforms (already offered by some of the existing approaches) is considered a reasonable extension of this work, here we propose automatic modeling of an OpenFaaS-inspired function-as-a-service implementation, allowing the fine-grained use of the available infrastructure.

We conclude this assessment with an overview of the most important expressivity constructs introduced in this work compared to what is available in Terraform, as shown in Figure 8. Terraform does not support any optimization factors but it does support one placement constraint (the `depends_on` constraint). Terraform does also provide support for load-balancing features offered by individual cloud providers but no normative load-balancing modeling across clouds. In contrast, our approach supports three provider-independent distributed execution paradigms (see Section 6), three optimization factors, and four placement constraints (see Section 7.2).

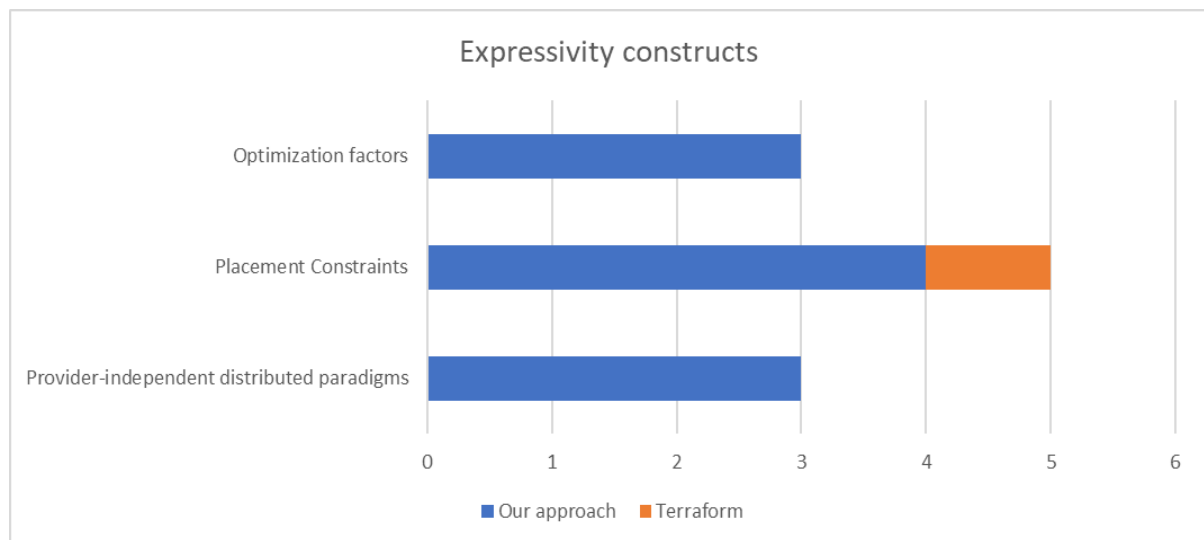


Figure 8. Expressivity construct support using our approach and Terraform, related to optimization factors, placement constraints, and provider-independent distributed paradigms.

8.2. Validating the Extended TOSCA in a Prototype Evaluation

In this section, we detail the implementation work on a TOSCA interpreter, exploiting our type-level TOSCA to determine edge deployment scenarios. The tooling used to exploit the aforementioned TOSCA extensions is available in a public repository. The TOSCA interpreter source code is available at (<https://github.com/ow2-proactive/prestocloud-tosca-interpreter>) and can be tested as a proof-of-concept with topologies based on the load-balancing and FaaS execution paradigms. This interpreter originated from the TOSCA parsing engine implemented in the Alien4cloud orchestrator (<https://alien4cloud.github.io/>).

The optimization of fragment deployment scenarios is achieved through the integration of the Btrplace scheduler library [34]. The Btrplace scheduler computes placement solutions for VMs on nodes—and is used in our case to place fragments on processing

infrastructure resources. This framework follows the constraint programming paradigm by using the Choco solver [33]. When provided a source model describing a VM deployment and a set of constraints, the Btrplace scheduler solves a (i) destination model describing a VM placement in conformance with expressed constraints and (ii) the necessary steps to migrate one model to another. The uniqueness of the solution is acquired by selecting a variable to optimize. The TOSCA interpreter proceeds with the following workflow to generate a deployment scenario from a type-level TOSCA specification:

1. The interpreter parses the supporting type-level TOSCA file and the supporting TOSCA definition and infrastructure resources files. If one file does not comply with TOSCA standards, the interpreter interrupts the process;
2. The identified computing resources are checked for their availability to receive a fragment. If a resource fails this availability check, it is excluded from the deployment. If a fragment is already being operated on this resource, it is considered for reallocation;
3. A Btrplace source model is programmed with available infrastructure resources, already deployed fragments, and their constraints. Fragments to be deployed are specified as constraints for the destination model;
4. Btrplace is invoked to turn the model into a destination model and a deployment plan. In the case that no solution is found (e.g., due to lack of resources or conflicting constraints), the interpreter interrupts the process. The choice of processing resources comprising the destination model is based on the reduction of the overall cost and is influenced by the optimization criteria set for each fragment;
5. The deployment plan is interpreted into a set of technical steps constituting deployment scenarios, encoded in a JSON file.

9. Results and Discussion

Following the approach that we introduced, a major advantage related to the installation of any cloud application is brought to the DevOps, as it is now possible to view an easily reviewable, provider-agnostic, standards-based model of the topology to be instantiated. The prototype type-level TOSCA generator we developed [31] can handle both Java annotations and JSON inputs reflecting an application graph. How this topology will later be managed and revised is outside the scope of this work, however we have demonstrated above that type-level and instance-level TOSCA can cope with dynamic setups. The enhancement of TOSCA should be impossible to achieve without creating new node types and new policies. All new language structures use the existing TOSCA syntax and each one targets a specific enhanced behavior. The extended TOSCA specification enables the modeling of self-managed FaaS-based applications alongside more traditional VM-based applications. Moreover, our enhancements also deliver modeling support for the more traditional paradigm of load-balanced applications, as well as JPPF [13], which is capable of creating dynamic processing infrastructures. All of these paradigms are “coordinator-driven”, using “coordinator” nodes for the deployment—either a FaaS Proxy, a load-balancer, or a JPPF master.

Applications comprised of coordinator-driven and more traditional architectures may coexist or can be completely separated. It is known [39] that cloud functions as a part of a FaaS deployment enjoy much lower startup times and offer more fine-grained cost execution options. These advantages make FaaS-based deployments preferable to conventional VM-based deployments in certain use-case scenarios. In other cases, however, the stability and predictability of VMs are preferred over FaaS. Our improvements enable gradual migrations from VM-based deployments to FaaS-based deployments and vice versa, using the expressiveness of TOSCA.

In addition, the extensions in the TOSCA specification also target the correct modeling of edge devices. Both fog and edge-only deployments are targeted by our approach, supporting mobility from the edge to the cloud and inversely where this can be implemented. Naturally, fog architectures can be combined in modeling with FaaS-based applications to describe FaaS fog deployments.

Additionally, irrespective of the topology that is actually deployed, we introduced support for placement constraints and optimization factors at the level of each software component (fragment) or the whole of the topology. Using our illustrative scenario, example optimization factors are presented both at the topology and fragment levels, permitting the run time optimization of the topology as required.

Regarding the relationship with existing solutions, in Section 8 we chose to compare our work with one of the most successful current commercial solutions, Terraform. We showed that Terraform is unable to provide an abstract view of a processing topology, and is, therefore, unable to enforce cross-cloud optimization policies without significant manual intervention. Secondly, exact knowledge of the edge devices that are available, as well as the instance types that are offered by each provider, appears to be the only path that can be taken to optimize costs while retaining performance to an acceptable level for each application fragment. Additionally, Terraform and other vendor-specific, template-based approaches require the meticulous configuration of all fine-grained networking parameters on behalf of the DevOps.

In contrast, our approach isolates the specification of application components from their deployment, can handle both small-scale and large-scale applications, can provide for different optimization criteria, and is based on the open standard of TOSCA. Moreover, when coupled with a proper TOSCA interpreter and topology reconfiguration tools, such as those described in [40], it provides a fully automated approach to application deployment and reconfiguration in a mixed edge–cloud infrastructure. Studies on improvements of TOSCA can readily be performed and will benefit companies requiring cross-cloud deployments on heterogeneous fog infrastructure, as TOSCA has the capacity to streamline and abstract the view of an application topology. Using type-level and instance-level TOSCA, the application description is not bound to any cloud provide, which allows the application to be preserved for long-term deployments.

For larger topologies, it is expected that the type-level TOSCA templates will become very lengthy. Even in this case, we believe that since the type and relative locations of nodes related to a particular fragment are known beforehand (processing, fragment, and mapping nodes), it will be easy to understand how a particular component is situated in the processing topology, provided that the structure of the application is based on the distributed software paradigms described in Section 6. When more complex relationships are involved, additional software paradigms should be defined to streamline the understanding of the application topology.

As part of this work, we provided a prototype type-level TOSCA generator, which can be used to create type-level TOSCA for an application topology. We also provided a prototype type-level TOSCA interpreter able to suggest the processing resources that should be spawned. However, we showed that other components are needed, such as the optimizer and the instance-level TOSCA generator, along with a TOSCA orchestrator, in order to allow the full exploitation of our results. A definite future research direction involves the development of components that can create the required outputs and further extend the capabilities of the TOSCA ecosystem.

The creation, updating, and parsing processes for type-level TOSCA and instance-level TOSCA are currently the responsibility of platform components that are installed beforehand. Thus, a direction for future research could be the determination of the most appropriate way to express the deployment and operation of these components in TOSCA, without adding complexity to the topology template of the actual application. Additionally, a greater number of software architecture and application-specific structures can be modeled in TOSCA by any adopter choosing to customize the TOSCA generation procedure. Another interesting research line could involve the modeling and the optimization of services—complete topologies of microservices, as discussed in [41]—as entities in TOSCA, which will be especially useful for organizations handling hundreds of microservices and tens of services.

While our approach provides for the modeling of edge and cloud CPU-based resources, we do not model the use of hardware accelerators such as field-programmable gate arrays (FPGAs) and graphics processing units (GPUs). However, FPGAs and GPUs are popular processing resources when increased performance, energy efficiency, and flexibility are desired in various application domains [42]. Research on the modeling of accelerators and the further abstraction of application fragments could lead to application topologies concurrently featuring accelerator-based and CPU-based versions of a fragment. Coupled with the definition of appropriate constraints for accelerator-based fragments, this modeling effort will enable the automatic selection of the most appropriate processing form by a suitable TOSCA orchestrator.

Finally, applications that are defined using our methodology in TOSCA can easily profit from software components able to deploy and scale application topologies based on monitoring data. Using the suggested approach, scaling in and out or up and down is very easy to model and users can readily understand—even from a terminal window—the changes that the platform has undergone. This support can be further extended, by proposing specific modeling for scaling directives, e.g., using TOSCA policies [5]. Based on the scaling directives, it is trivial to automatically create a new TOSCA template, which paves the way for a complete cloud adaptation approach, as described in [30].

10. Conclusions

In conclusion, we presented an approach to model some of the most important technologies that have appeared in the context of cloud computing over the past few years, using a two-flavored TOSCA scheme, which allows the DevOps to unlock the potential of optimized hybrid cloud–edge deployments and easily configure the criteria governing the deployment of components. Our work is backed by software implementation of the most important modeling stage, the type-level TOSCA generation. In our approach, classic architecture schemes can coexist with newer ones and can make use of edge devices, the cloud, or both. The expression of constraints is built into the topology template, and complete configuration over each fragment of the topology is possible. Moreover, each node template and node type have the same structure and follow the same conventions, therefore improving the understanding of the TOSCA document, whether at the instance or type level.

This work presents many research opportunities, involving the enhancement of TOSCA modeling at different levels. A first opportunity relates to the improvement of the modeling of the entities involved in TOSCA generation and processing in order to integrate them in the processing topology, and thus create self-contained deployment descriptions. Further, the extension of TOSCA with definitions for more processing architectures (e.g., GPUs, FPGAs) can be considered. Our modeling effort and our type-level TOSCA generator [31] can be used as a basis to support these extensions. Additional modeling artifacts to guide the optimization and the automated reconfiguration of even more diverse cloud application topologies can be defined. We have a vision of unified, scalable multi-cloud and multi-edge deployments, and believe that our work can be used as a basis for more ambitious definitions of cloud topologies.

Author Contributions: Conceptualization Y.V.; methodology, A.T., Y.V., D.A., and M.C.; software A.T.; validation A.T.; formal analysis A.T.; investigation A.T., M.C., and Y.V.; resources, G.M. and Y.V.; data curation, A.T.; writing-original draft preparation A.T.; writing-review and editing, Y.V., D.A., M.C., and G.M.; visualization, A.T.; supervision G.M.; project administration G.M. and Y.V.; funding acquisition, G.M. All authors have read and agreed to the published version of the manuscript.

Funding: Research reported in this article was funded by the European Union's Horizon 2020 Research and Innovation program, grant agreements Prestocloud No. 732339 and Morphemic No. 871643.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No significant datasets were analyzed in this study or created to support it.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A

```

tosca_definitions_version: toscaprestocloud_mapping_1_2

metadata:
  template_name: IMU generated types definition
  template_author: IMU
  template_version: 1.0.0-SNAPSHOT
  CostThreshold: 1000
  TimePeriod: 720
  ProviderName_0: Google_Cloud_Compute
  ProviderRequired_0: false
  ProviderExcluded_0: true
  MetricToMinimize: Cost

description: Types Description

imports:
  - toscanormative-types:1.2
  - iccsnormative-types:1.1
  - resourcedescriptions:1.0
  - placementconstraints:1.0

node_types:
  #Processing node selection:
  processing_node_LambdaProxy_0:
    description: A TOSCA representation of a processing node
    derived_from: prestocloud.nodes.proxy.faas
    requirements:
      - host:
          capability: toscacapabilities.Container
          node: prestocloud.nodes.compute
          relationship: toscarelationshihs.HostedOn
          node_filter:
            capabilities:
              - host:
                  properties:
                    - num_cpus: { in_range: [ 2, 4 ] }
                    - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
                    - storage_size: { in_range: [ 10 GB, 50 GB ] }
              - os:
                  properties:
                    - architecture: { valid_values: [ x86_64, i386 ] }
                    - type: { equal: linux }
                    - distribution: { equal: ubuntu }
              - resource:
                  properties:
                    - type: { equal: cloud }

  processing_node_imu_fragments_MultimediaManager_1:
    description: A TOSCA representation of a processing node
    derived_from: prestocloud.nodes.agent
    requirements:
      - host:
          capability: toscacapabilities.Container
          node: prestocloud.nodes.compute
          relationship: toscarelationshihs.HostedOn
          node_filter:
            capabilities:
              - host:

```

Listing A1. Cont.

```

    properties:
      - num_cpus: { in_range: [ 2, 4 ] }
      - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
      - storage_size: { in_range: [ 128 GB, 1024 GB ] }
  - os:
    properties:
      - architecture: { valid_values: [ x86_64, i386 ] }
      - type: { equal: linux }
      - distribution: { equal: ubuntu }
  - resource:
    properties:
      - type: { equal: cloud }

processing_node_imu_fragments_VideoTranscoder_2:
description: A TOSCA representation of a processing node
derived_from: prestocloud.nodes.agent
requirements:
  - host:
    capability: tosca.capabilities.Container
    node: prestocloud.nodes.compute
    relationship: tosca.relationships.HostedOn
    node_filter:
      capabilities:
        - host:
          properties:
            - num_cpus: { in_range: [ 2, 4 ] }
            - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
            - storage_size: { in_range: [ 4 GB, 32 GB ] }
        - os:
          properties:
            - architecture: { valid_values: [ x86_64, i386 ] }
            - type: { equal: linux }
            - distribution: { equal: ubuntu }
        - resource:
          properties:
            - type: { equal: cloud }
      - host:
        capability: tosca.capabilities.Container
        node: prestocloud.nodes.compute
        relationship: tosca.relationships.HostedOn
        node_filter:
          capabilities:
            - host:
              properties:
                - num_cpus: { in_range: [ 2, 4 ] }
                - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
                - storage_size: { in_range: [ 4 GB, 32 GB ] }
            - os:
              properties:
                - architecture: { valid_values: [ arm64, armel, armhf ] }
                - type: { equal: linux }
                - distribution: { equal: raspbian }
            - resource:
              properties:
                - type: { equal: edge }

processing_node_imu_fragments_AudioCaptor_3:
description: A TOSCA representation of a processing node
derived_from: prestocloud.nodes.agent
requirements:
  - host:
    capability: tosca.capabilities.Container
    node: prestocloud.nodes.compute
    relationship: tosca.relationships.HostedOn
    node_filter:
      capabilities:
        - host:
          properties:
            - num_cpus: { in_range: [ 1, 2 ] }

```

Listing A1. Cont.

```

- mem_size: { in_range: [ 1024 MB, 2048 MB ] }
- storage_size: { in_range: [ 4 GB, 32 GB ] }
- os:
  properties:
    - architecture: { valid_values: [ arm64, armel, armhf ] }
    - type: { equal: linux }
    - distribution: { equal: raspbian }
- resource:
  properties:
    - type: { equal: edge }
- sensors:
  properties:
    - microphone: { equal: "/dev/snd/mic0" }

processing_node_imu_fragments_FaceDetector_4:
description: A TOSCA representation of a processing node
derived_from: prestocloud.nodes.agent.faas
requirements:
- host:
  capability: tosca.capabilities.Container
  node: prestocloud.nodes.compute
  relationship: tosca.relationships.HostedOn
  node_filter:
    capabilities:
      - host:
        properties:
          - num_cpus: { in_range: [ 1, 2 ] }
          - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
          - storage_size: { in_range: [ 4 GB, 32 GB ] }
      - os:
        properties:
          - architecture: { valid_values: [ x86_64, i386 ] }
          - type: { equal: linux }
          - distribution: { equal: ubuntu }
      - resource:
        properties:
          - type: { equal: cloud }
- host:
  capability: tosca.capabilities.Container
  node: prestocloud.nodes.compute
  relationship: tosca.relationships.HostedOn
  node_filter:
    capabilities:
      - host:
        properties:
          - num_cpus: { in_range: [ 1, 2 ] }
          - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
          - storage_size: { in_range: [ 4 GB, 32 GB ] }
      - os:
        properties:
          - architecture: { valid_values: [ arm64, armel, armhf ] }
          - type: { equal: linux }
          - distribution: { equal: raspbian }
      - resource:
        properties:
          - type: { equal: edge }

processing_node_imu_fragments_PercussionDetector_5:
description: A TOSCA representation of a processing node
derived_from: prestocloud.nodes.agent.faas
requirements:
- host:
  capability: tosca.capabilities.Container
  node: prestocloud.nodes.compute
  relationship: tosca.relationships.HostedOn
  node_filter:
    capabilities:
      - host:
        properties:

```

Listing A1. Cont.

```

        - num_cpus: { in_range: [ 1, 2 ] }
        - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
        - storage_size: { in_range: [ 4 GB, 32 GB ] }
    - os:
        properties:
        - architecture: { valid_values: [ arm64, armel, armhf ] }
        - type: { equal: linux }
        - distribution: { equal: raspbian }
    - resource:
        properties:
        - type: { equal: edge }

processing_node_imu_fragments_VideoStreamer_6:
description: A TOSCA representation of a processing node
derived_from: prestocloud.nodes.agent.faas
requirements:
    - host:
        capability: toska.capabilities.Container
        node: prestocloud.nodes.compute
        relationship: toska.relationships.HostedOn
        node_filter:
            capabilities:
            - host:
                properties:
                - num_cpus: { in_range: [ 1, 2 ] }
                - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
                - storage_size: { in_range: [ 4 GB, 32 GB ] }
            - os:
                properties:
                - architecture: { valid_values: [ arm64, armel, armhf ] }
                - type: { equal: linux }
                - distribution: { equal: raspbian }
            - resource:
                properties:
                - type: { equal: edge }
            - sensors:
                properties:
                - video_camera: { equal: "/dev/video/camera0" }

topology_template:
policies:
    - collocation_policy_group_0:
        type: prestocloud.placement.Gather
        targets: [ imu_fragments_VideoStreamer, imu_fragments_VideoTranscoder ]

    - collocation_policy_group_1:
        type: prestocloud.placement.Gather
        targets: [ imu_fragments_PercussionDetector, imu_fragments_AudioCaptor ]

    - precedence_policy_group_0:
        type: prestocloud.placement.Precedence
        targets:
[ imu_fragments_VideoStreamer, imu_fragments_VideoTranscoder, imu_fragments_FaceDetector, imu_fragments_Mul
timediaManager, imu_fragments_AudioCaptor, imu_fragments_PercussionDetector ]

node_templates:
    deployment_node_LambdaProxy:
        type: processing_node_LambdaProxy_0

    LambdaProxy:
        type: prestocloud.nodes.fragment
        properties:
        id: 6
        name: LambdaProxy
        scalable: false
        occurrences: 1
        docker_cloud:
        image: "traefik:latest"

```

Listing A1. Cont.


```

    registry: "hub.docker.com"
    ports:
      - target: 11111
        published: 11111
        protocol: TCP
      - target: 11198
        published: 11198
        protocol: TCP
    optimization_variables:
      cost: 1
      distance: 1
      friendliness: {}
    requirements:
      - execute: deployment_node_LambdaProxy

deployment_node_imu_fragments_MultimediaManager:
  type: processing_node_imu_fragments_MultimediaManager_1

imu_fragments_MultimediaManager:
  type: prestocloud.nodes.fragment
  properties:
    id: 0
    name: imu_fragments.MultimediaManager
    scalable: false
    occurrences: 1
    docker_cloud:
      image: "multimedia_manager:latest"
      registry: "prestocloud.test.eu"
      variables: { "VIDEO_TRANSCODER_SERVICE": "{ get_property:
[deployment_node_LambdaProxy,host,network,addresses,1] }", "FACE_DETECTOR_SERVICE": "{ get_property:
[deployment_node_LambdaProxy,host,network,addresses,1] }", "RUNNING_THREADS": "2" }
    optimization_variables:
      cost: 5
      distance: 4
      friendliness: { "aws": "5", "gce": "0", "azure": "1" }
    requirements:
      - execute: deployment_node_imu_fragments_MultimediaManager

deployment_node_imu_fragments_VideoTranscoder:
  type: processing_node_imu_fragments_VideoTranscoder_2

imu_fragments_VideoTranscoder:
  type: prestocloud.nodes.fragment
  properties:
    id: 1
    name: imu_fragments.VideoTranscoder
    scalable: true
    occurrences: 1
    docker_edge:
      image: "video_transcoder_edge:latest"
      registry: "prestocloud.edge.test.eu"
      ports:
        - target: 10000
          published: 10000
          protocol: TCP_UDP
    docker_cloud:
      image: "video_transcoder_cloud:latest"
      registry: "prestocloud.test.eu"
      ports:
        - target: 10000
          published: 10000
          protocol: TCP_UDP
    optimization_variables:
      cost: 2
      distance: 8
      friendliness: { "aws": "5", "gce": "0", "azure": "1" }
    requirements:
      - execute: deployment_node_imu_fragments_VideoTranscoder

```

Listing A1. Cont.

```

deployment_node_imu_fragments_AudioCaptor:
  type: processing_node_imu_fragments_AudioCaptor_3

imu_fragments_AudioCaptor:
  type: prestocloud.nodes.fragment
  properties:
    id: 2
    name: imu_fragments.AudioCaptor
    scalable: false
    occurrences: 1
    docker_edge:
      image: "audiocaptor:latest"
      registry: "prestocloud.test.eu"
      variables: { "SAMPLING_RATE": "22 KHZ" }
    optimization_variables:
      cost: 1
      distance: 1
      friendliness: { }
    health_check:
      interval: 1
      cmd: "cat /proc/meminfo"
  requirements:
    - execute: deployment_node_imu_fragments_AudioCaptor

deployment_node_imu_fragments_FaceDetector:
  type: processing_node_imu_fragments_FaceDetector_4

imu_fragments_FaceDetector:
  type: prestocloud.nodes.fragment.faas
  properties:
    id: 3
    name: imu_fragments.FaceDetector
    scalable: true
    occurrences: 1
    docker_edge:
      image: "face_detector_edge:latest"
      registry: "local.prestocloud.test.eu"
      variables: { "PRECISION": "50", "ITERATIONS": "10" }
    docker_cloud:
      image: "face_detector_cloud:latest"
      registry: "prestocloud.test.eu"
      variables: { "PRECISION": "100", "ITERATIONS": "2" }
    optimization_variables:
      cost: 1
      distance: 1
      friendliness: { "aws": "5", "gce": "0", "azure": "1" }
    health_check:
      interval: 1
      cmd: "curl health.prestocloud.test.eu FaceDetector"
  requirements:
    - execute: deployment_node_imu_fragments_FaceDetector
    - proxy: deployment_node_LambdaProxy

deployment_node_imu_fragments_PercussionDetector:
  type: processing_node_imu_fragments_PercussionDetector_5

imu_fragments_PercussionDetector:
  type: prestocloud.nodes.fragment.faas
  properties:
    id: 4
    name: imu_fragments.PercussionDetector
    scalable: true
    occurrences: 1
    docker_edge:
      image: "percussion_detector_edge:latest"
      registry: "prestocloud.test.eu"
    docker_cloud:
      image: "percussion_detector_cloud:latest"
      registry: "prestocloud.test.eu"

```

Listing A1. Cont.

```

    optimization_variables:
      cost: 1
      distance: 1
      friendliness: {}
    requirements:
      - execute: deployment_node_imu_fragments_PercussionDetector
      - proxy: deployment_node_LambdaProxy

  deployment_node_imu_fragments_VideoStreamer:
    type: processing_node_imu_fragments_VideoStreamer_6

  imu_fragments_VideoStreamer:
    type: prestocloud.nodes.fragment.faas
    properties:
      id: 5
      name: imu_fragments.VideoStreamer
      scalable: true
      occurrences: 3
      docker_edge:
        image: "video_streamer:latest"
        registry: "prestocloud.test.eu"
        variables: { "VIDEO_TRANSCODER_SERVICE": "{ get_property:
[deployment_node_LambdaProxy,host,network,addresses,1] }", "VIDEO_RESOLUTION": "HD1080p" }
      optimization_variables:
        cost: 1
        distance: 1
        friendliness: {}
    requirements:
      - execute: deployment_node_imu_fragments_VideoStreamer
      - proxy: deployment_node_LambdaProxy

```

Listing A1. Generated Type-level TOSCA.

Appendix B

```

provider "aws" {
  profile  = "default"
  region  = "us-east-1"
}

# Network configuration ...Create a VPC to launch our instances into
resource "aws_vpc" "default" {
  cidr_block = "10.0.0.0/16"
}

# Create an internet gateway to give our subnet access to the outside world
resource "aws_internet_gateway" "default" {
  vpc_id = "${aws_vpc.default.id}"
}

# Grant the VPC internet access on its main route table
resource "aws_route" "internet_access" {
  route_table_id      = "${aws_vpc.default.main_route_table_id}"
  destination_cidr_block = "0.0.0.0/0"
  gateway_id          = "${aws_internet_gateway.default.id}"
}

# Create a subnet to launch our instances into
resource "aws_subnet" "default" {
  vpc_id            = "${aws_vpc.default.id}"
  cidr_block        = "10.0.1.0/24"
  map_public_ip_on_launch = true
}

resource "aws_security_group" "default" {
  name        = "terraform_example_lambda_proxy"
  description = "Used in the terraform"
  vpc_id      = "${aws_vpc.default.id}"
}

```

Listing A2. Cont.

```

    ingress {
      from_port = 22
      to_port   = 22
      protocol  = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }

# A security group for the Lambda Proxy
resource "aws_security_group" "lambda_proxy" {
  name        = "terraform_example_lambda_proxy"
  description = "Used in the terraform"
  vpc_id      = "${aws_vpc.default.id}"

  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port = 11111
    to_port   = 11111
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port = 11198
    to_port   = 11198
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

variable "key_name" {}
variable "public_key_path" {}

resource "aws_key_pair" "auth" {
  key_name   = "${var.key_name}"
  public_key = "${file(var.public_key_path)}"
}

resource "aws_instance" "FaceDetector" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
  key_name      = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id    = "${aws_subnet.default.id}"
  depends_on = [aws_instance.VideoTranscoder]
}

resource "aws_instance" "VideoTranscoder" {
  ami           = "ami-2757f621"
  instance_type = "c5.large"
  key_name      = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id    = "${aws_subnet.default.id}"
  depends_on = [docker_container.video_streamer]
}

resource "aws_instance" "LambdaProxy" {
  ami           = "ami-2757f622"
  instance_type = "c5.large"
  key_name      = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id    = "${aws_subnet.default.id}"
}

```

Listing A2. Cont.

```

    depends_on = [docker_container.percussion_detector,aws_instance.FaceDetector,aws_instance.VideoTranscoder]
  }

resource "aws_instance" "MultimediaManager" {
  ami           = "ami-2757f611"
  instance_type = "t3a.medium"
  key_name      = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id    = "${aws_subnet.default.id}"
  depends_on = [aws_instance.FaceDetector,aws_instance.VideoTranscoder,docker_container.percussion_detector]
}

# Configure the Docker providers
provider "docker" {
  host = "tcp://192.168.1.2:2375/"
}

provider "docker" {
  alias = "worker_2"
  host = "tcp://192.168.1.3:2375/"
}

provider "docker" {
  alias = "worker_3"
  host = "tcp://192.168.1.4:2375/"
}

# Create a container
resource "docker_container" "video_streamer" {
  image = docker_image.vs_image.latest
  name  = "vs_cont"
}

resource "docker_container" "audio_captor" {
  provider = docker.worker_2
  image = docker_image.ac_image.latest
  name  = "ac_cont"
}

resource "docker_container" "percussion_detector" {
  provider = docker.worker_3
  image = docker_image.pd_image.latest
  name  = "pd_cont"
  depends_on = [docker_container.audio_captor]
}

resource "docker_image" "vs_image" {
  name = "video_streamer:latest"
}

resource "docker_image" "ac_image" {
  provider = docker.worker_2
  name = "audio_captor:latest"
}

resource "docker_image" "pd_image" {
  provider = docker.worker_3
  name = "percussion_detector:latest"
}

```

Listing A2. Generated Type-level TOSCA.

References

1. Gartner Forecasts Worldwide Public Cloud End-User Spending to Grow 18% in 2021. Available online: <https://www.gartner.com/en/newsroom/press-releases/2020-11-17-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-18-percent-in-2021> (accessed on 12 February 2021).
2. Opара-Martins, J.; Sahandi, R.; Tian, F. Critical Analysis of Vendor Lock-in and Its Impact on Cloud Computing Migration: A Business Perspective. *J. Cloud Comp* **2016**, *5*, 4. [CrossRef]

3. Yousefpour, A.; Fung, C.; Nguyen, T.; Kadiyala, K.; Jalali, F.; Niakanlahiji, A.; Kong, J.; Jue, J.P. All One Needs to Know about Fog Computing and Related Edge Computing Paradigms: A Complete Survey. *J. Syst. Archit.* **2019**, *98*, 289–330. [CrossRef]
4. Bergmayr, A.; Breitenbücher, U.; Ferry, N.; Rossini, A.; Solberg, A.; Wimmer, M.; Kappel, G.; Leymann, F. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* **2018**, *51*, 1–38. [CrossRef]
5. TOSCA Simple Profile in YAML Version 1.3. Available online: <https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html> (accessed on 12 February 2021).
6. Alien4cloud/Alien4cloud. Available online: <https://github.com/alien4cloud/alien4cloud> (accessed on 12 February 2021).
7. Cloudify-Cosmo. Available online: <https://github.com/cloudify-cosmo> (accessed on 12 February 2021).
8. Binz, T.; Breitenbücher, U.; Haupt, F.; Kopp, O.; Leymann, F.; Nowak, A.; Wagner, S. OpenTOSCA—A Runtime for TOSCA-Based Cloud Applications. In Proceedings of the Service-Oriented Computing, Shanghai, China, 12–15 November 2012; pp. 692–695.
9. Kritikos, K.; Skrzypek, P.; Moga, A.; Matei, O. Towards the Modelling of Hybrid Cloud Applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 291–295.
10. Castro, P.; Ishakian, V.; Muthusamy, V.; Slominski, A. The Rise of Serverless Computing. *Commun. ACM* **2019**, *62*, 44–54. [CrossRef]
11. Baldini, I.; Castro, P.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Suter, P. Cloud-Native, Event-Based Programming for Mobile Applications. In Proceedings of the International Conference on Mobile Software Engineering and Systems, New York, NY, USA, 14 May 2016; pp. 287–288.
12. Kpavel/Incubator-Openwhisk. Available online: <https://github.com/kpavel/incubator-openwhisk> (accessed on 12 February 2021).
13. Jppf-Grid/JPPF. Available online: <https://github.com/jppf-grid/JPPF> (accessed on 12 February 2021).
14. Verginadis, Y.; Apostolou, D.; Taherizadeh, S.; Ledakis, I.; Mentzas, G.; Tsagkaropoulos, A.; Papageorgiou, N.; Paraskevopoulos, F. PrEstoCloud: A Novel Framework for Data-Intensive Multi-Cloud, Fog, and Edge Function-as-a-Service Applications. *Inf. Resour. Manag. J.* **2021**, *34*, 66–85. [CrossRef]
15. Tamburri, D.A.; Van den Heuvel, W.-J.; Lauwers, C.; Lipton, P.; Palma, D.; Rutkowski, M. TOSCA-Based Intent Modelling: Goal-Modelling for Infrastructure-as-Code. *SICS Softw.-Inensiv. Cyber-Phys. Syst.* **2019**, *34*, 163–172. [CrossRef]
16. Wurster, M.; Breitenbücher, U.; Kepes, K.; Leymann, F.; Yussupov, V. Modeling and Automated Deployment of Serverless Applications Using TOSCA. In Proceedings of the 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA), Paris, France, 20–22 November 2018; pp. 73–80.
17. Casale, G.; Artač, M.; van den Heuvel, W.-J.; van Hoorn, A.; Jakovits, P.; Leymann, F.; Long, M.; Papanikolaou, V.; Presenza, D.; Russo, A.; et al. RADON: Rational Decomposition and Orchestration for Serverless Computing. *SICS Softw.-Inensiv. Cyber-Phys. Syst.* **2019**. [CrossRef]
18. RADON Public Deliverables—D2.4 Architecture and Integration Plan II. Available online: <https://radon-h2020.eu/wp-content/uploads/2020/07/D2.4-Architecture-and-integration-plan-II.pdf> (accessed on 12 February 2021).
19. Paasage Public Deliverables—D2.1.3 Camel Documentation. Available online: https://paasage.ercim.eu/images/documents/docs/D2.1.3_CAMEL_Documentation.pdf (accessed on 12 February 2021).
20. Achilleos, A.P.; Kritikos, K.; Rossini, A.; Kapitsaki, G.M.; Domaschka, J.; Orzechowski, M.; Seybold, D.; Griesinger, F.; Nikolov, N.; Romero, D.; et al. The Cloud Application Modelling and Execution Language. *J. Cloud Comp.* **2019**, *8*, 20. [CrossRef]
21. Nyrén, R.; Edmonds, A.; Papaspyrou, A.; Metsch, T.; Parák, B. Open Cloud Computing Interface—Core. Available online: <https://redmine.ogf.org/attachments/242/core.pdf> (accessed on 12 February 2021).
22. Glaser, F.; Erbel, J.; Grabowski, J. Model Driven Cloud Orchestration by Combining TOSCA and OCCl. In Proceedings of the 7th International Conference on Cloud Computing and Services Science, Porto, Portugal, 24–26 April 2017.
23. Challita, S.; Korte, F.; Erbel, J.; Zalila, F.; Grabowski, J.; Merle, P. Model-Based Cloud Resource Management with TOSCA and OCCl. *Softw. Syst. Modeling* **2021**. [CrossRef]
24. Wurster, M.; Breitenbücher, U.; Falkenthal, M.; Krieger, C.; Leymann, F.; Saatkamp, K.; Soldani, J. The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS Softw.-Inensiv. Cyber-Phys. Syst.* **2019**. [CrossRef]
25. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
26. van Lingen, F.; Yannuzzi, M.; Jain, A.; Irons-Mclean, R.; Lluch, O.; Carrera, D.; Perez, J.L.; Gutierrez, A.; Montero, D.; Marti, J.; et al. The Unavoidable Convergence of NFV, 5G, and Fog: A Model-Driven Approach to Bridge Cloud and Edge. *IEEE Commun. Mag.* **2017**, *55*, 28–35. [CrossRef]
27. Bjorklund, M. The YANG 1.1 Data Modeling Language. Available online: <https://www.rfc-editor.org/info/rfc7950> (accessed on 12 February 2021).
28. Noghabi, S.A.; Kolb, J.; Bodik, P.; Cuervo, E. Steel: Simplified Development and Deployment of Edge-Cloud Applications. In Proceedings of the HotCloud'18: Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing, Boston, MA, USA, 9 July 2018.
29. Mortazavi, S.H.; Salehe, M.; Gomes, C.S.; Phillips, C.; de Lara, E. Cloudpath: A Multi-Tier Cloud Computing Framework. In Proceedings of the Second ACM/IEEE Symposium on Edge Computing; Association for Computing Machinery, New York, NY, USA, 12 October 2017; pp. 1–13.

30. Tsagkaropoulos, A.; Papageorgiou, N.; Apostolou, D.; Verginadis, Y.; Mentzas, G. Challenges and Research Directions in Big Data-Driven Cloud Adaptivity. In Proceedings of the 8th International Conference on Cloud Computing and Services Science, Madeira, Portugal, 19–21 March 2018.
31. PrEstoCloud/Application-Fragmentation-Deployment-Recommender. Available online: <https://gitlab.com/prestocloud-project/application-fragmentation-deployment-recommender> (accessed on 12 February 2021).
32. Copil, G.; Moldovan, D.; Truong, H.-L.; Dustdar, S. RSYBL: A Framework for Specifying and Controlling Cloud Services Elasticity. *ACM Trans. Internet Technol.* **2016**, *16*, 1–18. [\[CrossRef\]](#)
33. Jussien, N.; Rochart, G.; Lorca, X. Choco: An Open Source Java Constraint Programming Library. In Proceedings of the CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08), Paris, France, 20–23 May 2008; pp. 1–10.
34. Hermenier, F.; Lawall, J.; Muller, G. BtrPlace: A Flexible Consolidation Manager for Highly Available Applications. *IEEE Trans. Dependable Secure Comput.* **2013**, *10*, 273–286. [\[CrossRef\]](#)
35. Papageorgiou, N.; Verginadis, Y.; Apostolou, D.; Mentzas, G. Fog Computing Context Analytics. *IEEE Instrum. Meas. Mag.* **2019**, *22*, 53–59. [\[CrossRef\]](#)
36. Sun, Y.; Lin, F.; Xu, H. Multi-Objective Optimization of Resource Scheduling in Fog Computing Using an Improved NSGA-II. *Wireless Pers. Commun.* **2018**, *102*, 1369–1385. [\[CrossRef\]](#)
37. Zhu, Z.; Zhang, G.; Li, M.; Liu, X. Evolutionary Multi-Objective Workflow Scheduling in Cloud. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 1344–1357. [\[CrossRef\]](#)
38. Zhang, F.; Cao, J.; Li, K.; Khan, S.U.; Hwang, K. Multi-Objective Scheduling of Many Tasks in Cloud Platforms. *Future Gener. Comput. Syst.* **2014**, *37*, 309–320. [\[CrossRef\]](#)
39. Jain, A.; Baarzi, A.F.; Alfares, N.; Kesidis, G.; Urgaonkar, B.; Kandemir, M. SplitServe: Efficiently Splitting Complex Workloads Across FaaS and IaaS. In Proceedings of the ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, 20 November 2019.
40. Verginadis, Y.; Alshabani, I.; Mentzas, G.; Stojanovic, N. 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, 24–26 April 2017.
41. Abdelaal, M.A.; Ebrahim, G.A.; Anis, W.R. Efficient Placement of Service Function Chains in Cloud Computing Environments. *Electronics* **2021**, *10*, 323. [\[CrossRef\]](#)
42. Cong, J.; Fang, Z.; Lo, M.; Wang, H.; Xu, J.; Zhang, S. Understanding Performance Differences of FPGAs and GPUs. In Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 29 April–1 May 2018; pp. 93–96.