applied sciences

MDPI

Article

Enabling Role-Based Orchestration for Cloud Applications

Yue Wang [1], Choonhwa Lee [1,*], Shuyang Ren [1], Eunsam Kim [2] and Sungwook Chung [3]

[1] Department of Computer Science, Hanyang University, Seoul 133-791, Korea; wylk2019@hanyang.ac.kr (Y.W.); syren@hanyang.ac.kr (S.R.)
[2] Department of Computer Engineering, Hongik University, Seoul 121-791, Korea; eskim@hongik.ac.kr
[3] Department of Computer Engineering, Changwon National University, Changwon 51140, Korea; swchung@changwon.ac.kr
* Correspondence: lee@hanyang.ac.kr; Tel.: +82-2-2220-1268

Abstract: With the rapidly growing popularity of cloud services, the cloud computing faces critical challenges to orchestrate the deployment and operation of cloud applications on heterogenous cloud platforms. Cloud applications are built on a platform model that abstracts away underlying platform-specific details, so that their orchestration can benefit from the abstract view and flexibility of the underlying platform configuration. However, considerable efforts are still required to properly manage complicated cloud applications. This paper proposes a model-driven approach to cloud application orchestration which promotes the concerns of distinct roles for cloud system provisioning and operation. By establishing a set of capabilities as modeling constructs, our approach allows TOSCA-based application topology itself and its orchestration needs to be specified in a way to provide a more targeted support for different needs and concerns of application developers and operators. With novel orchestration features like application topology description, platform capability modeling, and role-awareness for cloud application orchestration, it can significantly reduce the complexity of application orchestration in diverse cloud environments. To show the feasibility and effectiveness of our proposal for cloud application orchestration, we present a proof-of-concept orchestration system implementation and evaluate its deployment and orchestration results in a Kubernetes cluster.

Keywords: orchestration; TOSCA; model translation; role-based; capability-centric; Kubernetes

check for updates

Citation: Wang, Y.; Lee, C.; Ren, S.; Kim, E.; Chung, S. Enabling Role-Based Orchestration for Cloud Applications. *Appl. Sci.* **2021**, *11*, 6656. https://doi.org/10.3390/app11146656

Academic Editor: Arcangelo Castiglione

Received: 21 May 2021
Accepted: 14 July 2021
Published: 20 July 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-native applications are becoming increasingly complicated nowadays, which poses significant challenges to the design, development, and delivery of them [1]. To help application builders overcome the difficulties, PaaS(Platform-as-a-Service) was introduced as a cloud service model that provides a configurable computing platform [2]. PaaS platforms are also understood as DevOps environments that provides viable tools to quickly build, test, and deliver cloud-native applications [3,4]. It provides application builders with supports needed during the lifecycle of an application from development and testing to delivery and operations. The means for provisioning and managing a cloud application may vary in terms of involved technologies and underlying platform capabilities such as auto-scaling, monitoring, and rolling releases of the application. To keep the platform versatile, rich configuration options have to be provided, which would lead to a higher level of flexibility and maintainability. However, not all the configuration parameters may be of interest to ones. Some of the configuration fields are supposed to be used by platform builders, while others are intended for application builders [5]. A single application component might be backed up by multiple platform capabilities, which entails that application builders are expected to understand all the configuration parameters of those underlying platform resources. Obviously, the orchestration will likely be a time-consuming and error-prone task for application builders. When considering the orchestration task for

applications built upon various capabilities, it becomes even worse because the task must deal with all the details of resources and capabilities' configurations [6].

In general, DevOps practice involves two different roles of application developers and operators. Application developers are in charge of implementing their application logic, while application operators are responsible for keeping the implementation up and running properly. In this article, application builders are considered to play either of the two roles. Both groups make use of platform resources directly or capabilities built upon underlying resources to provide cloud applications. However, it is likely for them to find it challenging to identify what specific configuration points are relevant for their job. If they are left to perform their orchestration tasks solely on their own without any coordination, they may end up making conflicting decisions. Therefore, it is crucial to make a clear distinction among orchestration configurations from the perspective of roles, so that different roles can exist in a harmonious way to achieve the ultimate, shared goal of cloud-native applications' orchestration.

This article proposes a model-driven cloud application orchestration approach which distinguishes the concerns of distinct roles for cloud system provisioning and operation. Each role's concerns are encapsulated into modeling constructs of our orchestration model which is built upon prominent standards for cloud application deployment and management. Irrelevant configurations are shielded from the role and taken care of automatically, which results in a significant reduction of the orchestration burden.

The remainder of this paper is organized as follows. Section 2 summarizes key standard models for cloud application orchestration before presenting a sample orchestration scenario that motivates our work. Section 3 introduces our proposed approach and the architectural design of role-based, capability-centric orchestration system along with prototype implementation effort. Our evaluation study and its results are reported in Section 4. After discussing related work in Section 5, the paper concludes in Section 6.

2. Background Technology for Cloud Orchestration

First, we briefly discuss two outstanding efforts that push forwards orchestration standards for cloud applications. After then, we present a sample orchestration scenario that is intended to motivate our work.

2.1. Standard Models for Cloud Application Orchestration

Our orchestration approach leverages existing cloud orchestration standards, the most relevant ones of which are TOSCA (Topology and Orchestration Specification for Cloud Applications) and OAM (Open Application Model). OASIS TOSCA is an OASIS standard that aims to enable portable specification of cloud applications. According to the standard, an application topology is defined in an underlying platform-agnostic way in order to promote the application interoperability and portability across different cloud providers [7,8]. To define the topology of applications, TOSCA provides a type system that can model applications' components as typed Nodes and relationships among the components as typed Relationships. Besides basic properties and interfaces, typed Nodes and Relationships also have Capability and Requirement by which application designers specify dependencies among components in terms of component functionalities. A component may be defined to provide a certain ability as a typed entity or require an ability also defined as a typed entity provided by others. As an example, consider a WordPress application, connected to a MongoDB database that can adapt to input fluctuations with auto-scaling capability enabled. As shown in Figure 1, the application can be defined as a TOSCA service template that is composed by a Topology Template along with accompanying types. In other words, the application's parts and their relationships can be modeled and specified using the TOSCA vocabulary. Additionally, TOSCA offers rich design primitives for specifying orchestration constraints and behaviors within a declarative service template configuration, such as policy and workflow [9]. Policy can be defined to express non-functional requirements applied to a group of typed nodes. Workflow is also known as Plan in TOSCA that

includes a set of sequential tasks for fulfilling a certain orchestration task. Our approach employs the TOSCA standard as a domain-specific language for describing the topology of applications and defining their orchestration workflows.

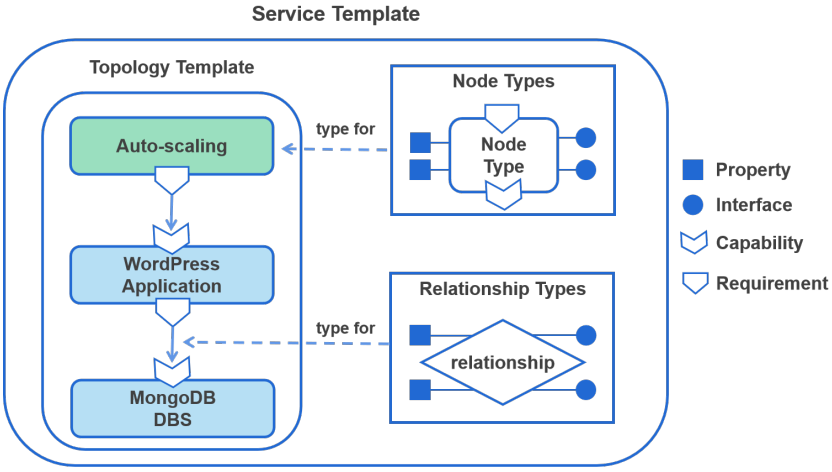

Figure 1. TOSCA definition for a sample application.

OAM is a runtime-agnostic modeling and specification standard for defining cloud applications [10]. Focused on the application layer rather than underlying platform resources or the infrastructure layer, OAM aims to model applications in a modular, extensible, and portable way, enabling application delivery to diverse target platforms. Distinguishing between parts that application developers and operators are responsible for, OAM argues for separation of roles' concerns when defining applications. With regard to the standard, an application is composed of two parts mainly; the first is the component inside which the application logic resides to provide its service, while the other is the specification of operational characteristics for the component. More specifically, an OAM component is introduced for application developers to declare a runtime environment where the application runs, which might be containers, virtual machines, IoT devices, or cloud platforms. The runtime environment is also known as workload in the context of OAM. For example, an OAM workload, called containerized workload , represents a standard definition of runtime for containerized applications. On the other hand, an OAM trait is a discretionary runtime overlay that empowers workloads with additional operational features. Examples of it include scaling workloads dynamically to achieve high availability and routing the traffic towards a specific host domain. Both component and trait are designed as reusable templates that can be easily instantiated to compositions of different applications. Application configuration is a top-level OAM resource manifest in which all building blocks of an application are to be instantiated and interconnected. Figure 2 depicts the structure of an application containing one component for a containerized workload with two traits, scale and route, attached, and Listing 1 shows its corresponding OAM YAML snippet.

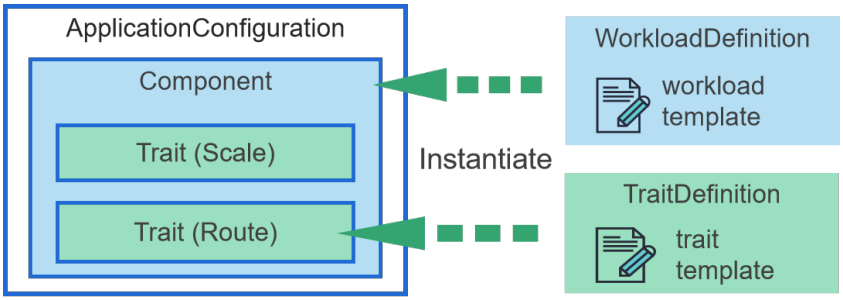

Figure 2. OAM application configuration sample.

Listing 1. OAM configuration YAML snippet.

```yaml
apiVersion: core.oam.dev/v1alpha2
kind: ApplicationConfiguration
metadata:
  name: example-app
spec:
  components:
  - componentName: example-comp
    traits:
    - name: scale
      properties:
        replicas: 2
    - name: route
      properties:
        ... # properties of route
---
apiVersion: core.oam.dev/v1alpha2
kind: Component
metadata:
  name: example-comp
spec:
  workload:
    apiVersion: core.oam.dev/v1alpha2
    kind: ContainerizedWorkload
    spec:
      ... # specification of workload
```

Kubernetes [11] is an open-source container orchestration framework that automates the management of container lifecycle as well as execution of containerized applications. Kubernetes cluster architecture consists of a master node and a set of worker nodes. The master node is responsible for the maintenance and management of the cluster, while the worker node runs a set of Docker containers for applications. This master/slave architecture is illustrated in Figure 3. The minimal deployment unit in Kubernetes is a pod that contains one or more Docker containers and provides shared storage and networking capability for the containers. Several management components in charge of the maintenance of the cluster state are located in the master node, which include API server, Controller Manager, Scheduler, and etcd component. The API server provides an entry point to control the entire Kubernetes cluster, and etcd is a highly available distributed key-value store that keeps data for the cluster state. The Scheduler determines which worker to assign which pod. To bring the system to a desired functioning state, Kubernetes relies on the Controller Manager that consists of a few controller processes such as node controller, job controller, etc. Each worker node is equipped with Kubelet, Kube-Proxy, and Docker container runtime. More specifically, the Kubelet process handles the pod state based on the commands from the master, and the Kube-Proxy process is a network proxy that resides on each node.

Recently, container technology has been a popular choice as the base on which to build cloud orchestrator runtime for multi-component applications. Being represented by containers, the components of the applications can run fast and efficiently. In addition, Kubernetes as container orchestration technology provides a means to collectively manage related containers, which is found useful in deploying and managing multi-component cloud application.

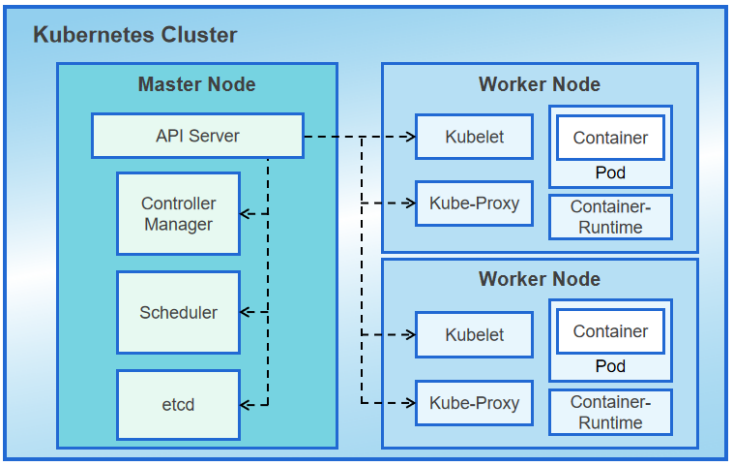

Figure 3. Kubernetes cluster architecture.

2.2. Orchestration Scenario for Cloud Application

To motivate our work, we consider a sample orchestration scenario for a WordPress application deployed in a container runtime environment, which is an open-source content management system written in PHP. The application is basically made up of two functional components: WordPress servers providing back-end services as well as web pages and a MongoDB database node connected to the server. It is noted that the TOSCA topology of the application is illustrated in Figure 1.

Our orchestration system considers Kubernetes as the deployment platform. With a combination of well-defined declarative APIs, clear abstractions, and comprehensive extensions, Kubernetes is widely accepted as a solid foundation for DevOps platforms [11,12]. Kubernetes offers a rich set of fundamental computing resources and configurable capabilities. As illustrated in Figure 4, two components of WordPress application are running inside a Kubernetes Deployment, while a Kubernetes Service serves as a load balancer as well as an entry point for users to access the WordPress service. It is also noted that our orchestration scenario involves two fundamental orchestration capabilities, auto-scaling and progressive delivery, to cope with realistic production environments.

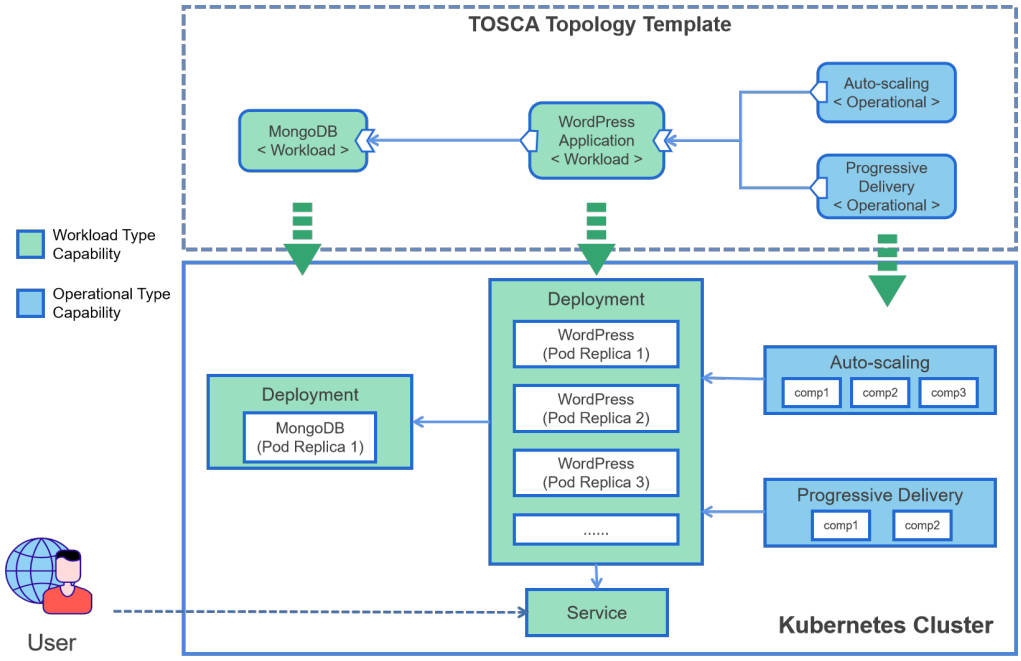

Figure 4. WordPress cloud application in action.

Automatic scaling is an operational capability used to adjust the application capacity to dynamically changing demands [13,14]. It allows the number of the replicas of a particular component to be dynamically adjusted according to performance metrics such as CPU utilization rate or number of requests per second to the application. In our scenario, auto-scaling is applied to the WordPress server Deployment in such a way that the number of replicas is automatically increased or decreased in response to the fluctuation of incoming requests. Auto-scaling capability is composed of two main modules, which are a monitoring module and a scaling one. The former is responsible for monitoring its target component and collecting relevant metrics data, while the latter automatically scales the number of replicas based on its scaling rules.

Progressive delivery is a widespread practice for gradually rolling out new features in order to avoid adverse impact on application operations [15]. Our scenario plans to upgrade the WordPress server replicas from version v1.0 to version v2.0 in batches. A certain portion of them is upgraded in each round rather than all replicas at once, so that remaining v1.0 replicas continue to provide the service, if something goes wrong during the upgrade. In production environments, an application often has a number

of running replicas created and managed by its auto-scaling capability to guarantee the service availability. Progressive delivery provides a means to upgrade the replica set to a new version at a controlled pace.

These two capabilities are frequently utilized and found beneficial. However, their setup is not straightforward for application builders because they oftentimes lack sufficient knowledge about underlying platform resources upon which the capabilities are fulfilled. In case of OAM, two roles are involved with the DevOps practice. Application operators are the primary users of those two operational capabilities. Obviously, they are not expected to know how to set up or implement these capabilities, even though they attach desired operational characteristics to running workloads using the capabilities. On the other hand, it is application developers who configure a workload runtime environment. To manage the complexity in performing orchestration configurations and dealing with low-level details, we propose a new orchestration scheme that can separate different roles' concerns from oftentimes overwhelming configuration specifics for cloud applications.

3. TOSCA Model-Based Approach to Cloud Application Orchestration

In this section, we start off by proposing an orchestration approach which is based on a TOSCA orchestration model and OAM application specification model [16]. As depicted in Figure 5, our approach permits platform builders to identify common configurable components and package them as platform capabilities with minimal configuration fields through an abstraction layer between underlying platform resources and applications at the top. Application builders, including application developers and operators, no longer directly manipulate underlying platform resources. Instead, they utilize platform capabilities to develop, test, and deliver their applications, as the capability layer shields the details of resource configurations and usages from them. Moreover, the capabilities are thought to belong to either of the two generic roles of application developers and operators. By establishing a set of capabilities as modeling constructs, our approach allows TOSCA-based application topology itself and its orchestration needs to be specified in a way to provide a more targeted support for different needs and concerns of application developers and operators. With support for features like application topology model, platform capability, and role-awareness for cloud application orchestration, our approach significantly reduces the complexity of application orchestration on cloud platforms.

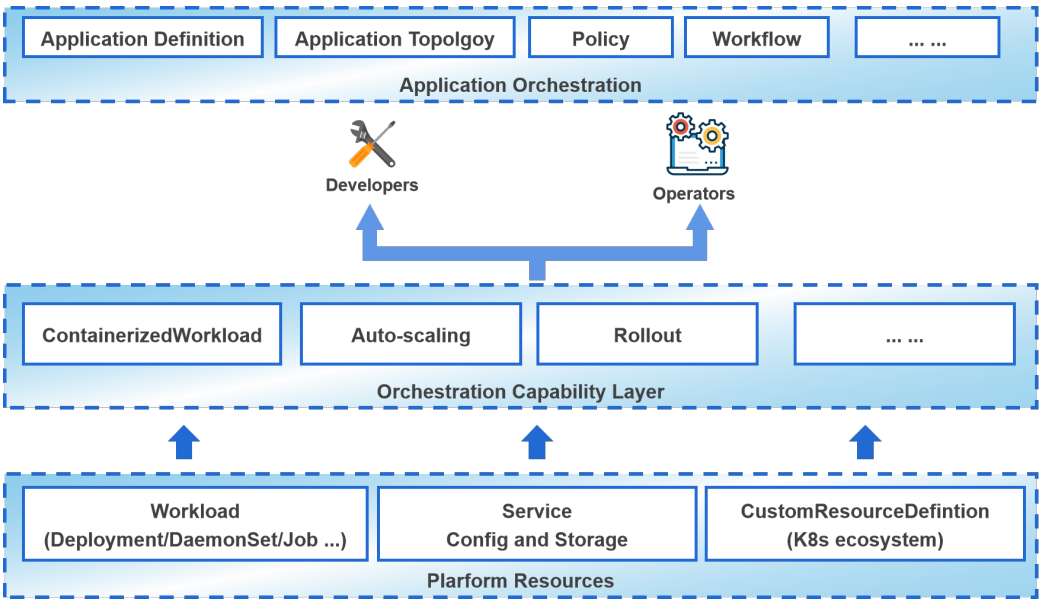

Figure 5. Role-based cloud application orchestration.

Our proposed approach can be realized by extending state-of-the-art technologies for cloud orchestration. As introduced in Section 2, TOSCA offers rich primitives and

extensible types but has no support for role-awareness and separation of concerns. Focused on a model specification for cloud applications, TOSCA remains neutral with regard to run-time implementations or how a TOSCA-conforming orchestration system may deploy orchestration manifests on cloud platforms. By contrast, OAM provides role-awareness support and an official runtime implementation based on Kubernetes. However, it lacks a comprehensive and robust meta-model to build an orchestration system, including interface, policy, and workflow definitions. In particular, without support for application topology definition, OAM hardly makes for a promising candidate technology for cloud orchestration models. Therefore, complementing each other's shortcomings for cloud application orchestration, a combination of TOSCA and OAM would provide a solid foundation on which to build up our model-driven, role-based cloud orchestration system.

3.1. Architectural Design of a Topology Model-Based Orchestration System

As illustrated in Figure 6, the architecture of our orchestration system consists of three parts: TOSCA modeling layer at the top, extended OAM runtime at the bottom, and model conversion engine that bridges the two domains:

1. First, TOSCA Service Templates in YAML files are loaded and processed by the parser in TOSCA module. Besides built-in TOSCA normative types and the meta-model, the parser also consumes a set of types tailored to specifying entities introduced by our approach. Outputs from the TOSCA parser are stored in the memory to be passed on to the conversion engine.
2. The conversion engine performs translation from TOSCA to OAM. Each of the normative TOSCA types as well as customized ones has a corresponding OAM typed entity stored in the TOSCA-to-OAM type mapping repository. The conversion engine is responsible for indexing the mappings and converting the components of TOSCA Topology Templates into OAM Application Configurations. Additionally, if any TOSCA workflow definition is contained in the Service Template, the engine converts it into workflow directives executed by the workflow engine.
3. Once OAM Application Configurations are prepared, our OAM-extended Kubernetes runtime can fulfill the deployment and orchestration of the application in a Kubernetes cluster.

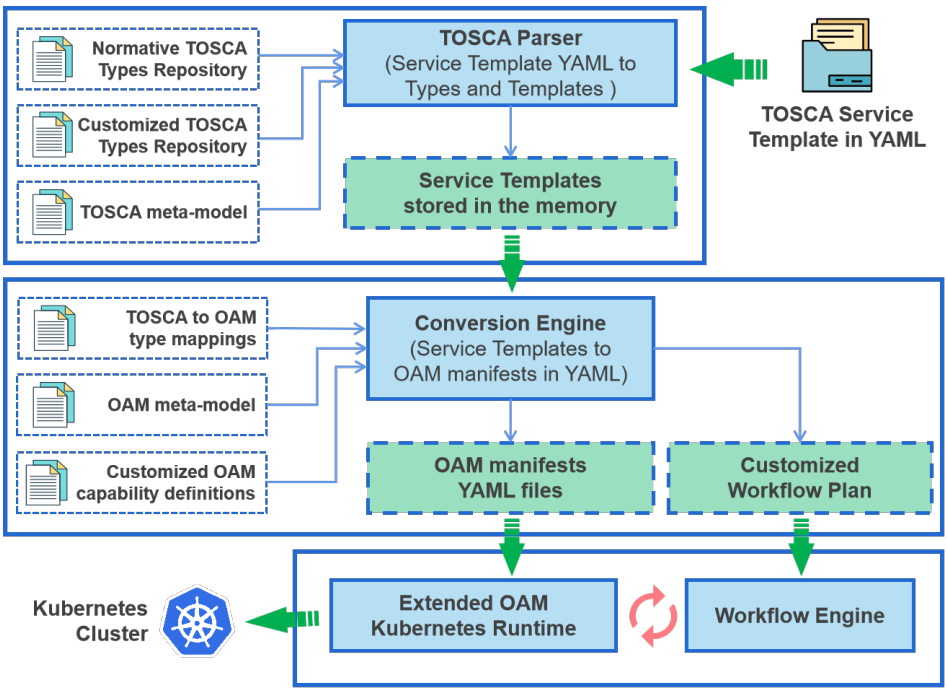

Figure 6. TOSCA-based orchestration system architecture.

Further details regarding the composition and operation of each of the three parts are discussed below.

3.1.1. TOSCA Modeling

We have defined a set of customized TOSCA types that are categorized into two groups. One group represents workload capabilities for application developers who are concerned about workloads, where the application resides, as well as the runtime configuration of the workloads. The second group models operational capabilities for application operators who are responsible to attach operational characteristics to workloads. In the motivational scenario introduced in the previous section, the WordPress server resides inside a Kubernetes Deployment and relies on a Kubernetes Service to deliver its service to users. A combination of Deployment and Service is a typical workload capability that represents an instantiation of runtime environment for containerized applications equipped with a scaler and a load balancer. Auto-scaling capability represents an operational capability that can dynamically scale workloads to a growing demand. Similarly, rollout capability is another example of an operational functionality that gradually introduces a new version to a group of replicas.

Based on the motivational scenario, we introduced extension node types for each capability, including *dcc.workload.containerized* derived from *dcc.workload.root* for the combination of Kubernetes Deployment and Service, and *dcc.operational.autoScaling* and *dcc.operational.simpleRollout* derived from *dcc.operational.root* for auto-scaling and rollout capability, respectively. As shown in Listing 2, the node *dcc.workload.containerized* contains a set of properties to specify the runtime configuration of a containerized application whose service is accessible through exposed container ports. Its *image* field defines the container image used to run the service, while its *ports* field defines the container port used to access the service.

The definition of two operational capability node types is shown in Listing 3. The node *dcc.operational.autoScaling* represents the auto-scaling capability. From the perspective of application operators, the *dcc.operational.autoScaling* should provide properties to specify basic auto-scaling rules. Some important fields include *maxReplicaCount* and *minReplicaCount* for replica count constraints, and *promThreshold* to indicate the threshold of triggering a scaling action. The node *dcc.operational.simpleRollout* represents the operational capability that can control the application behavior during its progressive delivery. The *maxUnavailable* is the maximum number of replicas that can become unavailable during rolling out. The *replicas* is the final number of replicas once progressive delivery completes. Additionally, the *batch* defines the number of replicas that should be created in each round of rolling.

We have chosen to model operational capabilities as TOSCA Node types rather than TOSCA Policy types. An operational capability allows us to define operation interfaces for application operators to manipulate operational characteristics. Additionally, one instance of an operational capability is only allowed to attach to one instance of workload. In other words, multiple workloads cannot share operational capability instances. TOSCA Policy is designed to apply non-functional constraints to a group of nodes and does not provide a means to define operation interfaces. Therefore, modeling operational capabilities as node types rather than policy types is more suitable in our approach and treats operational capability and workload capability in the same way.

Two root Node types, *dcc.workload.root* and *dcc.operational.root*, indicate several common Requirements and Capabilities of workload and operational capabilities. The Workload nodes derived from *dcc.workload.root* have the *workload* capability in common, which means that an operational capability node can be attached to it or other workload capability nodes can connect to it. In addition, a common requirement *connect-to* indicates that they can connect to another workload node. Additionally, the node *dcc.operational.root* empowers its child node types with common capability *operational* and common requirement *attach-to*. Thus, a workload node and an operational node are interconnected through the normative relationship type of *AttachesTo* that can fulfill the requirement *attach-to*. When defining

the properties of a capability node type, it is sufficient for application builders to specify a small number of configuration parameters to control the behaviour of the capabilities rather than each and every field of its underlying resources. This way, the complexity of application orchestration specification can be kept low. In a real-world scenario, deciding which properties or parameters concern a particular role is a task for the platform builders who leverage our approach to build platform capabilities. They should be familiar with overall configuration of a specific capability and have good knowledge on the concerns of roles who will consume this capability in designated orchestration scenarios. Taking all of the above into account, they can decide which configuration points are supposed to present through capability properties while others should be hidden. With these extended types, we can define the TOSCA Topology Template of the motivational scenario as diagrammed in Figure 1.

Listing 2. Extension Node Type of dcc.workload.containerized.

```
node_types:
  dcc.workload.containerized:
    derived_from: dcc.workload.root
    attributes:
      arch:
        type: string
        required: false
      containers:
        type: list
        required: true
        entry_schema:
          description: runtime configuration
          type: dcc.datatypes.containerinfo
---
# extension data types
data_types:
  dcc.datatypes.containerinfo:
    derived_from: tosca.datatypes.root
    properties:
      name:
        type: string
        required: true
      image:
        type: string
        required: true
      ports:
        type: list
        require: false
        entry_schema:
          type: dcc.datatypes.container.port
      cmd:
        type: list
        required: false
        entry_schema:
          type: string
      ...
  dcc.datatypes.container.port:
    derived_from: tosca.datatypes.root
    properties:
      name:
        type: string
        required: true
      container_port:
        type: integer
        required: true
      protocol:
        type: string
        required: false
        default: tcp
```

Listing 3. Extension Node Types of dcc.operational.autoScaling and dcc.operational.simpleRollout.

```
node_types:
  dcc.operational.autoScaling:
    derived_from: dcc.operational.root
    attributes:
      maxReplicaCount:
        type: integer
        required: true
      minReplicaCount:
        type: integer
        required: true
      promQuery:
        type: string
        required: true
      promThreshold:
        type: integer
        required: true

  dcc.operational.simpleRollout:
    derived_from: dcc.operational.root
    attributes:
      batch:
        type: string
        required: true
      replicas:
        type: integer
        required: true
      maxUnavailable:
        type: integer
        required: true
```

In order to guarantee that parsed TOSCA specifications can be input into the model conversion module, we have developed a TOSCA parser based on an open-source TOSCA processor named Puccini [17]. Puccini provides a stand-alone TOSCA parser that can fulfill normalization, validation, inheritance, and assignment of TOSCA Service Templates. It outputs a flat, serializable data structure for the ensuing model conversion.

3.1.2. Application Model Conversion

To convert TOSCA Service Templates to OAM Application Configurations, we have identified and defined a set of mapping rules between two models as summarized in Table 1. In our approach, TOSCA is used to describe application topology and orchestration configuration. We employ the primitives of TOSCA type system to build a set of extension types to represent specific capability entities. Thus, our conversion task focuses on in-house extension types but not all TOSCA normative types. Since our extension TOSCA Node types have now been further divided into two kinds for developers and operators, TOSCA types can be easily mapped to OAM entities that have also been refined into two corresponding sub-types. The conversion process is composed of five phases: validating whether each TOSCA Node type has a matching OAM entity, rendering OAM components for workload type nodes, rendering OAM traits for operational capability nodes, resolving the relationships between the nodes, and generating an OAM Application Configuration. Regarding the properties conversion in a node, we rely on a simple mapping strategy that ensures that each property has a matching OAM entity *spec* field with the same name. For example, the *dcc.operational.autoScaling* node type in Listing 4 is converted to the OAM trait instance *AutoScaling*. Its properties' value will be assigned to *AutoScaling spec* fields with the same name. The value of the property *maxReplicaCount* will be assigned to the field *spec.maxReplicaCount*, and the value of the property *promQuery* will be assigned to the field *spec.promQuery*.

Table 1. TOSCA-to-OAM entity mapping.

TOSCA Types/Templates	OAM Types
Node Type (dcc.workload.*)	WorkloadDefinition
Node Template (dcc.workload.*)	Component
Node Type (dcc.operational.*)	TraitDefinition
Node Template (dcc.operational.*)	Trait
Relationship (AttachesTo)	Attach Trait to Component
Relationship (Workloads_Connection)	Component Data Inputs/Outpues

Listing 4. TOSCA property conversion.

```
# TOSCA topology template
topology_template:

  node_template:
  ...
    auto-scaling:
      type: dcc.operational.autoScaling
      properties:
        maxReplicaCount: 5
        minReplicaCount: 1
        promQuery: sum(rate(http_request[2m]))
        promThreshold: 3
      requirements:
      - attach-to: wordpress-comp
        relationship: attachesTo
  ...

---
```

```
# OAM application configuration
apiVersion: core.oam.dev/v1alpha2
kind: ApplicationConfiguration
metadata:
  name: WordPress-app
spec:
  components:
    ...
    traits:
    - trait:
        apiVersion: extend.dcc/v1alpha1
        kind: AutoScaling
        spec:
          maxReplicaCount: 5
          minReplicaCount: 1
          promQuery: sum(rate(http_request[2m]))
          promThreshold: 3
    ...
```

For two nodes connected by an *AttachesTo* Relationship type in TOSCA, their OAM entities will be organized into an Application Configuration. Specifically, a Trait mapped from an operational capability type is made from part of a Component mapped from a workload capability node. Figure 7 depicts this kind of conversion from TOSCA Template Topology containing the *AttachesTo* relationship to OAM Application Configuration.

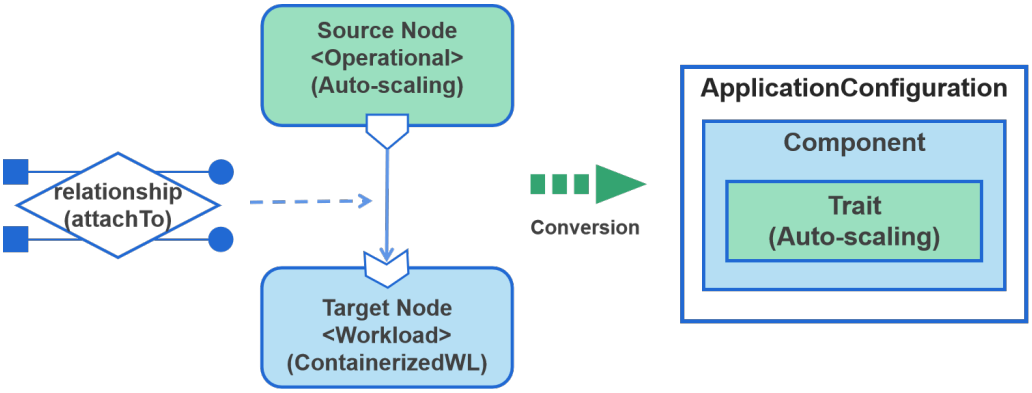

Figure 7. Conversion of AttachesTo Relationship.

Another important relationship is a dependency between two workloads. As illustrated in our motivational scenario, the connection between WordPress server and MongoDB database is modeled as such a dependency relationship. Different from relationships between a workload and an operational capability, dependency between workloads has no counterpart concept in OAM specification. To overcome this deficiency, we leverage a data transfer mechanism in OAM Kubernetes runtime that permits data passing among components within an Application Configuration. The idea is to create a custom relationship type *workloads_connection* with two end points. One is the field path to which the target node (producer) writes data as output, and the second is the field path of field from which the source node (consumer) reads input data. The conversion engine sets the data input and output in the OAM components converted from two workload Node templates connected by the *workloads_connection* relationship template. Figure 8 illustrates this conversion from TOSCA Topology Template containing *workloads_connection* relationship to OAM Application Configuration.

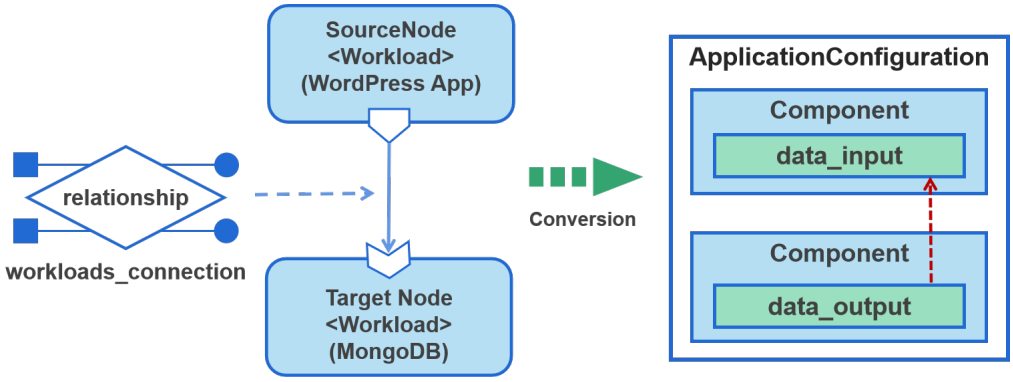

Figure 8. Conversion of Workloads_Connection Relationship from TOSCA to OAM.

3.1.3. Extension to OAM Kubernetes Runtime and Workflow Engine

To fulfill OAM configuration manifests of an application in a Kubernetes cluster, we have developed an extension to OAM Kubernetes runtime [18]. Basing our orchestration system on OAM Kubernetes, we integrated several in-house extensions to it. One of them is a set of Kubernetes controllers fulfilling the capabilities' functions, such as auto-scaling and rollout. Following Kubernetes operator pattern, these controllers can create underlying platform resources and organize them to work according to the configuration

defined by corresponding TOSCA capability Node types [19]. They take care of low-level configuration details intended to be hidden from the capability layer, allowing application builders to remain concentrated on their orchestration goals without being distracted by low-layer details.

Another extension is the workflow engine interacting with the runtime to synchronize the state of application components and execute workflow directives defined in TOSCA Service Templates. TOSCA provides an imperative workflow definition that can be used to deploy, manage, or terminate a TOSCA Service Template. It allows the specification of the cases that has not been planned in the definition of Node and Relationship types ahead of time.

In contrast, workflow is not supported by OAM runtime because OAM is positioned to be a purely declarative standard model. Therefore, we designed a workflow engine to parse and execute workflow definitions from the TOSCA part. With regard to the schema of TOSCA workflow, our workflow engine faces two challenges; one is to observe the state of components at runtime, and the other is to interpret and execute workflow directives, when a certain state is reached. Kubernetes API dictates a design convention that a resource should record the current state in its "*/status*" sub-resource. Our workflow engine makes use of it to keep track of the state changes. To address other challenges, we defined a series of custom directives that help the workflow engine to resolve operations defined by a TOSCA model into those of modified OAM configurations. An operation of a Node type will result in certain modifications to generated OAM entities. For instance, the *set_new_image* operation of *dcc.workload.containerized* Node type can accept an image tag as an input parameter to update the node's image tag property. If this operation is invoked, the workflow engine asks for inputs to set the image tag field of the OAM Component. Once a Component's configuration is updated, OAM Kubernetes runtime automatically applies the changes to the live workloads.

4. Experimental Evaluation

We performed a set of experiments with our prototype implementation of the proposed orchestration system (https://github.com/dcc-lab-2021/tosca-oam accessed on 15 May 2021). We choose the motivational orchestration scenario introduced earlier in the paper, featuring TOSCA-based modeling along with auto-scaling and progressive delivery capabilities. Running in a Kubernetes cluster, an application needs a Kubernetes Deployment as its hosting workload within which the application resides. A Kubernetes Service is also required to expose the application functionality to the outside of the cluster. The Deployment and its associated Service are central pieces for application developers who are responsible to choose suitable workloads to run the applications.

Our prototype implementation of the proposed orchestration system is diagrammed in Figure 9. In order to enable auto-scaling capability, Prometheus and KEDA (Kubernetes-based Event Driven Autoscaling) are employed to implement a standalone Kubernetes controller utilizing the operator pattern [20,21]. Prometheus collects monitoring data from target Kubernetes Services and calculates metric results according to specified rules. KEDA fetches the measurements from Prometheus and sets up a metric server in the Kubernetes cluster. KEDA also creates and associates a Kubernetes HPA to the target Deployment, which triggers scale-out or scale-in actions based on the inputs from the metric server. It is worth mentioning that the auto-scaling may cause unexpected interference with the progressive delivery that is underway at the moment, since a fixed number of replicas are assumed with the rolling update. Thus, the auto-scaling temporarily puts the HPA on hold to ensure that the replica set size remains the same during progressive delivery phase.

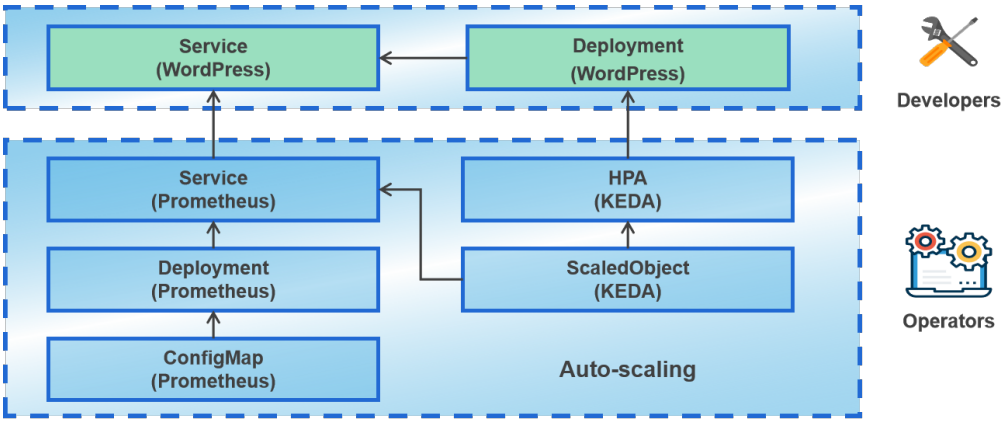

Figure 9. Key components of WordPress application deployment.

4.1. TOSCA Model Conversion Cost

We defined the WordPress application of our motivational scenario in our extended TOSCA model. Listing 5 is a YAML snippet of TOSCA Topology Template of the sample application. The TOSCA Topology Template consists of multiple parts: two workload Node templates representing the WordPress application and MongoDB database, two operational Node templates for auto-scaling and rollout capabilities, and a Relationship template connecting the nodes. Given the TOSCA Service Template specification as input, our prototype system generates a manifest of OAM resources including an Application Configuration and two Components as in Listing 6. Through observing the mapping relationship between the same name fields in the two listings, we can see how the conversion is actually performed. Each property of extended TOSCA types has a corresponding field in OAM entities. The OAM Kubernetes runtime will provision and deploy the application according to the OAM manifest in a Kubernetes cluster.

To evaluate the performance of the model conversion engine, we performed a series of conversions of different TOSCA Service Templates into OAM configurations. The motivational scenario serves as a baseline to compare the execution times of the cases with an increasing number of workloads and operational capability nodes. Conversions for the same service template with varying number of workloads were executed ten times. Elapsed times for the model translation are plotted in Figure 10 across different scales of the application. As seen in the graph, the conversion time increases gradually, as the application becomes more complicated.

Listing 5. TOSCA Topology Template of WordPress application.

```
topology_template:
  node_template:
    mongodb-comp:
      type: dcc.workload.containerized
      properties:
        ...

    wordpress-comp:
      type: dcc.workload.containerized
      properties:
        containers:
        - name: wordpress-app
          image: dcc-dev/wordpress-app:1.0
          ports:
          - containerPort: 8080
            name: wordpress-svc
      requirements:
      - connect-to: mongodb-comp
        relationship: workloads_connection

    auto-scaling:
      type: dcc.operational.autoScaling
      properties:
        maxReplicaCount: 5
        minReplicaCount: 1
        promQuery: sum(rate(http_request[2m]))
        promThreshold: 3
      requirements:
      - attach-to: wordpress-comp
        relationship: attachesTo

    rollout:
      type: dcc.operational.simpleRollout
      properties:
        ...
      requirements:
      - attach-to: wordpress-comp
        relationship: attachesTo
```

Listing 6. Resultant OAM Application Configuration.

```
apiVersion: core.oam.dev/v1alpha2
kind: ApplicationConfiguration
metadata:
  name: WordPress-app
spec:
  components:
  - componentName: mongodb-comp
    dataOutputs:
        - name: mongodb-conn-endpoint
          fieldPath: status.connEndpoint
        - name: mongodb-conn-secret
          fieldPath: status.connSecret
  - componentName: wordpress-comp
    dataInputs:
        - name: mongodb-conn-endpoint
          fieldPath: spec...connEndpoint
        - name: mongodb-conn-secret
          fieldPath: spec...connSecret
    traits:
    - trait:
        apiVersion: extend.dcc/v1alpha1
        kind: AutoScaling
        spec:
          maxReplicaCount: 5
          minReplicaCount: 1
          promQuery: sum(rate(http_request[2m]))
          promThreshold: 3
    - trait:
        apiVersion: extend.dcc/v1alpha1
        kind: SimpleRollout
        spec:
          ...
---
apiVersion: core.oam.dev/v1alpha2
kind: Component
metadata:
  name: wordpress-comp
spec:
  workload:
    apiVersion: core.oam.dev/v1alpha2
    kind: ContainerizedWorkload
    spec:
      containers:
      - name: wordpress-app
        image: dcc-dev/wordpress-app:1.0
        ports:
        - containerPort: 8080
          name: wordpress-svc
```

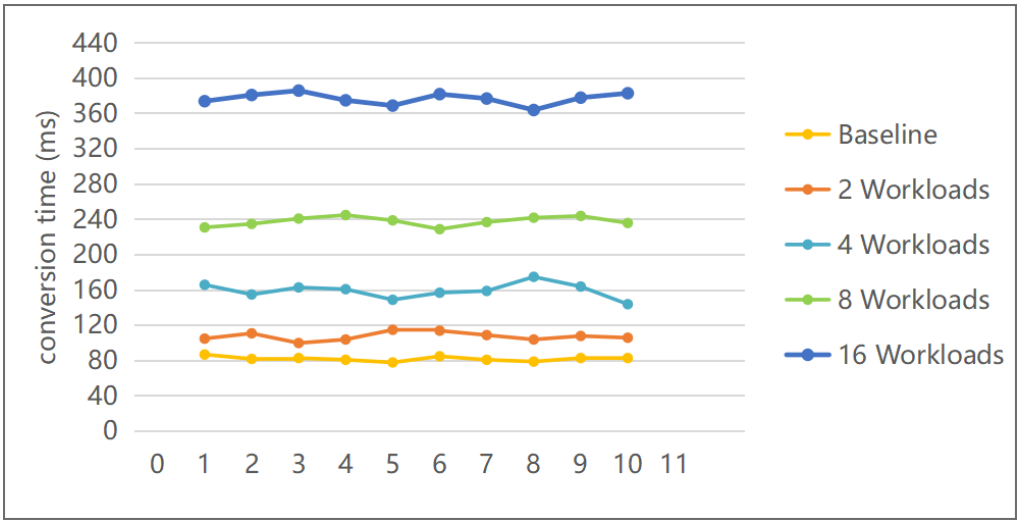

Figure 10. Conversion times of TOSCA to OAM specification.

4.2. Deployment and Complexity Reduction

The conversion module outputs OAM manifests that are fed into OAM Kubernetes runtime in a target Kubernetes cluster. To evaluate the deployment of applications across diverse cloud environments, we installed our orchestration system on four mainstream cloud platforms including Amazon Web Service, Microsoft Azure, Google Cloud, and Alibaba Cloud. We leveraged managed Kubernetes services of the cloud providers which enables higher availability and better performance than a manual setup in the local machine. We repeated the application deployment for our sample scenario multiple times on each cloud platform and averaged the times taken before the WordPress application becomes available and the auto-scaling feature gets ready. The results in Figure 11 show that our prototype system is able to deploy the application on these cloud platforms within a minute.

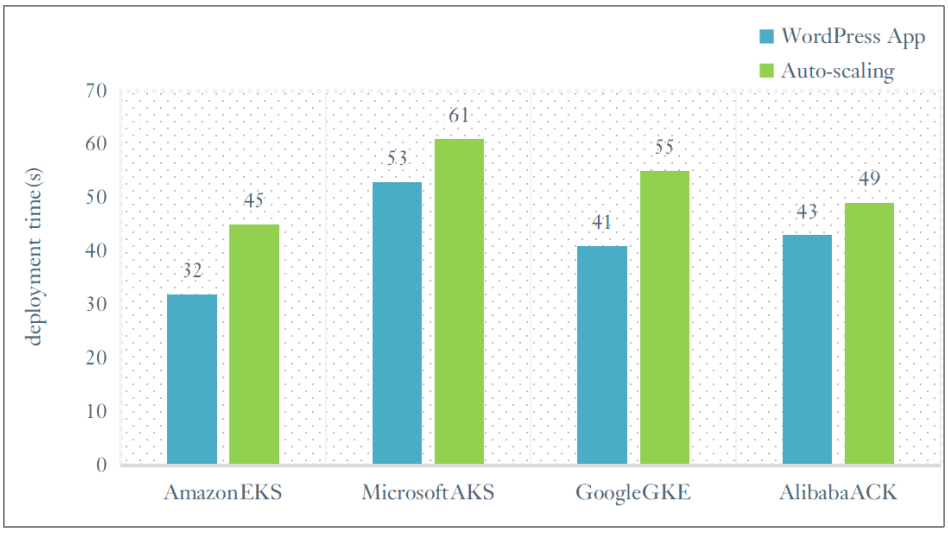

Figure 11. Deployment times on managed Kubernetes service.

To see how simplified it can make matters when it comes to orchestration configurations, we compared the number of configuration fields that have to be cared about. As presented in Table 2, only six fields at the top need to be configured in case of our approach. We'd like to emphasize that this is a drastic reduction from previous approaches where all the low-level details, as listed at the bottom of the table, should have been specified and configured. In our case, the six parameters are enough to set up an application with auto-scaling capability enabled. These configurable fields should be of interest for the two roles of application developers and operators. It is noted that our role-aware orchestration model makes this reduction possible, as its building blocks, from basic modeling types to top-level application topology, are identified and distinguished according to different roles' concerns. All other parts that do not concern particular roles remain hidden from them and are automatically handled by the orchestration system.

Table 2. Comparison of orchestration configuration fields.

Capability Configuration Fields	
ContainerizedWorkload (Workload Capability)	**AutoScaling (Operational Capabilitiy)**
.spec.template.spec.containers.*	.spec.promQuery
.spec.ports.*	.spec.promThreshold
	.spec.maxReplicaCount
	.spec.minReplicaCount
Related Underlying Resource Configuration Fields	
deployments.apps/v1 (main application)	**deployments.apps/v1 (Prometheus component)**
apiVersion/kind	apiVersion/kind
.metadata	.metadata
.spec.replicas	.spec.replicas
.spec.selector.matchLabels	.spec.selector.matchLabels
.spec.template.metadata.*	.spec.template.metadata.*
.spec.template.spec.containers.*	.spec.template.spec.containers[0].*
services(.core)/v1 (main application)	.spec.template.spec.containers[0].volumemounts
apiVersion/kind	.spec.template.spec.volumes.*

Table 2. *Cont.*

Related Underlying Resource Configuration Fields	
.metadata	**services(.core)/v1 (Prometheus component)**
.spec.ports.*	apiVersion/kind
.spec.selector	.metadata
scaledobjects.keda/v1alpha1	.spec.ports.*
apiVersion/kind	.spec.selector
.metadata	**configmap(.core)/v1 (Prometheus component)**
.spec.scaleTargetRef.deploymentName	apiVersion/kind
.spec.minReplicaCount	.metadata
.spec.maxReplicaCount	.spec.data
.spec.triggers[0].type	
.spec.triggers[0].serverAddress	
.spec.triggers[0].metricName	
.spec.triggers[0].query	
.spec.triggers[0].threshold	

4.3. Workflow and Progressive Delivery

In order to make sure that our workflow engine works as expected, we conducted an experiment of progressive delivery which involves orchestration workflows. Progressive delivery is a process of releasing updates of applications in a controlled manner, steering the automated promotion or rollback of the update. Progressive delivery involves both application developers and operators. Developers are responsible for providing the artifacts of a new version as well as a workload configuration for running the artifacts. By contrast, operators carry out multiple tasks, such as specifying a rolling strategy, modifying other operational capabilities to avoid interfering with the rollout, determining a rollback strategy, etc. This is where an orchestration workflow comes into play. It is composed of a series of steps to handle various tasks. Our approach promotes the principle of separation of concerns by allowing users to define a role-aware workflow in TOSCA. Each step of the workflow is relevant to specific roles.

The workflow is divided into five steps as shown in the Listing 7. During progressive delivery, the number of replicas of a new version should follow the defined rules. However, auto-scaling may interfere with progressive delivery. For example, with incoming requests decreasing, an auto-scaling decision may shut down newly-created replicas of the new version, which is probably not what is expected to happen. Therefore, before initiating progressive delivery, operators should lock the total number of replicas in the step *lock_autoscaling*. In the following step of *config_rollout*, operators assign proper values to the rollout parameters, such as rolling interval and batch size of each increment, according to which the workflow engine creates and attaches a rollout capability node template to the target component node. As the only step belonging to developers, *rolling_update* accepts a new version of the application as the input parameter. Once the configuration is complete, the workflow engine packs the new configuration of components as well as newly added rollout capability, and applies it to the cluster. The *rollout* controller can detect any modification on target components and trigger a batch of rolling update according to the rule defined in the step *config_rollout*. In each round of the rolling update, a certain portion of replicas are replaced with the new version. When all replicas are upgraded to a new version successfully, operators deactivate the rollout capability of the component. Finally, the auto-scaling gets unlocked to allow it to resume its normal operation.

Listing 7. Orchestration workflow of progressive delivery.

```
workflows:
  progressive_delivery:
    steps:
      lock_autoscaling:
        target: auto-scaling
        activities: [...]
        on_success:
        - config_rollout
      config_rollout:
        target: rollout
        activities: [...]
        on_success:
        - rolling_update
      rolling_update:
        target: wordpress-comp
        activities:
        - set_new_image: dcc-dev/wordpress-app:2.0
        on_success:
        - deactive_rollout
        - unlock_autoscaling
      deactive_rollout:
         target: rollout
         activities: [...]
      unlock_autoscaling:
         target: auto-scaling
         activities: [...]
```

The experiment considers upgrading the WordPress application from version v1 to version v2. For the sake of observation, we developed a service endpoint for WordPress server, observe/rollout/version, which returns the version name of the component. The progress of rolling update can be estimated by looking at accumulated response time from each version. Since incoming requests are evenly distributed across the replica set, elapsed times of each version would represent its share of the total processing capacity. Figure 12 illustrates the progress of the rolling update in terms of the response time, as its update round performs repeatedly. In addition, 10,000 requests are generated by using Apache server benchmarking tool (ab -n 10,000 -c 1 <service-host/observe/rollout/version>). As shown in the graph, all traffic is initially handled by v1 replicas. According to the rollout strategy batch: 20% that we use, every round, one fifth of the replicas are being replaced with a new version, which can be observed from the correspondingly increasing response time of the new replicas.

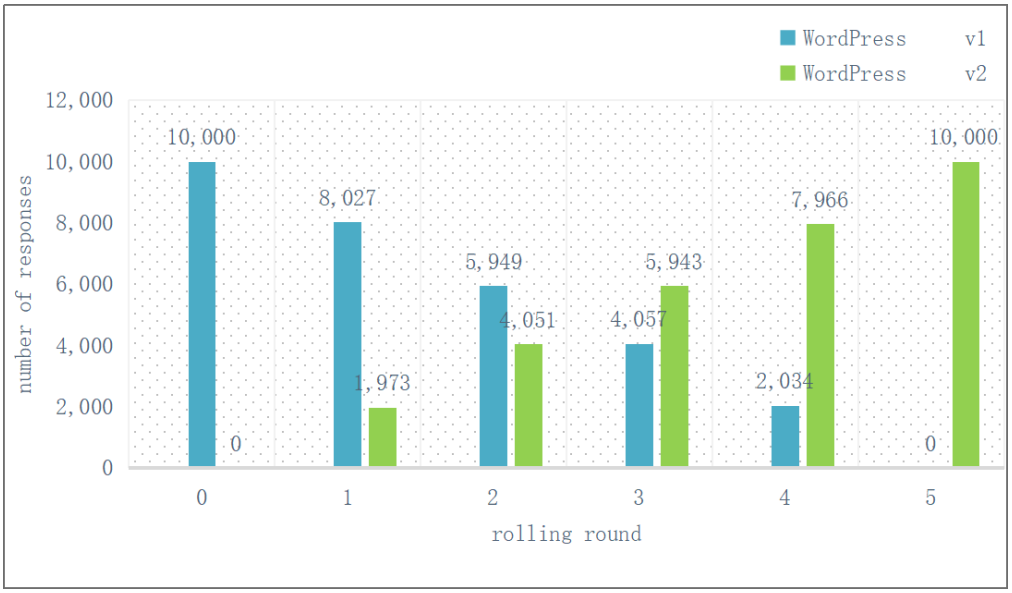

Figure 12. Progress of rolling update.

4.4. Discussion

To the best of the authors' knowledge, we are the first ones to explore the possibility of the integration of two prominent orchestration technologies of TOSCA and OAM. Unlike other approaches, TOSCA application components are mapped to underlying OAM entities in our orchestration scheme. As a result, the proposed approach can benefit from role separation of concerns and higher-level orchestration features enabled by OAM technology. Since our approach is geared towards higher-level abstractions, low-level details such

as infrastructure resource managements are hidden from the application developers and operators. This way, it can effectively ease much of the complexities and burdens inherent to cloud application orchestration. Consequently, application builders can focus on their own tasks without being worried about under-the-hood details. However, a downside of the proposal is that it lacks fine-grained orchestration ability for cloud services and applications. Being positioned as an orchestration solution at the capability level and above, our proposed system is notable for handling orchestration needs with regard to low-level infrastructure resources. It is noteworthy that there exist other approaches that can support both IaaS- and PaaS-level orchestration [22]. Regrading the runtime implementation, our system is built upon Kubernetes. As a result, it can handle containerized workloads only. Our prototype system can be extended to support other types of workloads, especially non-containerized workloads. In addition, further flexibility could be considered regarding the application component to container mapping relationships. It is noteworthy that some proposals already support the case of multiple components being placed within a single container [23]. Lastly, the workflow engine in our prototype could be augmented further. Even though TOSCA specifies primitives for workflow definition including interfaces of node types, our prototype implementation does not provide a full coverage of the primitives to generate corresponding execution directives. In addition, the current implementation is limited in that one workflow step can only specify the activities and states of a single node. However, it is likely that there exists an orchestration scenario where a workflow step involves multiple nodes in the cluster.

5. Related Work

Much attention has recently been paid to the technology of cloud orchestration. As a result, various alternative approaches to cloud application orchestration have been explored, each with different strengths and weaknesses [6,24,25]. Some of the efforts target the standardization of orchestration features and interfaces, while others have focused the development of real orchestration tools and solutions. In addition, when it comes to the orchestration layer of focus, they exhibit a wide range of low-level resources to application layer functionalities. In order to assess the novelty of our proposal against others, we have conducted a qualitative comparison study of several outstanding approaches in the field. Table 3 compares our approach with Tosker, TOSCAMP, Cloudify, and KubeVela.

This paper extends our previous work [26] where the idea of role-based cloud orchestration was proposed along with its early results presented. In this version, having advanced the orchestration system architecture with support for features like application topology model, platform capability abstraction, and role-awareness for cloud application orchestration, we have demonstrated that our proposal significantly reduces the complexity of application orchestration on cloud platforms.

Table 3. Qualitative analysis of orchestration approaches.

	Our Approach	**Tosker**	**TOSCAMP**	**Cloudify**	**KubeVela**
Base Specification	TOSCA 1.0 and OAM 0.2.1	TOSCA 1.0	TOSCA 1.0 and CAMP 1.1	TOSCA-based DSL	OAM 0.2.1
Application Model	TOSCA topology and capabilities	TOSCA topology	TOSCA topology	Models written in Cloudify DSL	Capabilities but no topology
Role-Awareness	Role-awareness	No role-awareness	No role-awareness	No role-awareness	Role-awareness
Capability Abstraction	Typed capability as model constructs	No capability abstraction	No capability abstraction	No capability abstraction	Typed capability as model constructs
Workflow Definition	Imperative workflow definition by TOSCA	Declarative workflow definition by TOSCA	Imperative workflow definition by TOSCA	Built-in deployment-specific workflows	No workflow definition
Multi-Cloud Support	Multi-cloud support based on Kubernetes	No multi-cloud support	Multi-cloud support based on IaaS/PaaS standards	Multi-cloud support based on Kubernetes	Multi-cloud support based on Kubernetes
Cross-Cloud Support	Cross-cloud based on Kubernetes Federation	No cross-cloud support	Cross-cloud support based on standarized IaaS/PaaS APIs	Cross-cloud based on Kubernetes Federation	Cross-cloud based on Kubernetes Federation
Containerization Support	Containerd, CRI-O, and Docker	Only Docker	No containerization support	Containerd, CRI-O, and Docker	Containerd, CRI-O, and Docker

Considered as a mature and prominent topology specification, TOSCA is adopted by a number of cloud application orchestration solutions. TosKer [27] presents a Docker-based orchestration system that can manage the deployment and operation of TOSCA-based representations of applications in Kubernetes clusters. As a means to model cloud application topology, it introduces new TOSCA node types of Container and Software that represent a Docker container and a software component of the application, respectively. Interconnection and dependencies among them are defined by the TOSCA normative relationship types, including *AttachesTo*, *ConnectsTo*, *HostedOn*, and *DependsOn*. Given a TOSCA representation of a cloud application, the orchestration engine instantiates the application over a Kubernetes cluster. In this system, a container is the minimal entity that can be orchestrated. This rigid coupling of software components and their hosting containers is relaxed in their follow-up work [23], so that the components can be managed independently from their hosting containers. More specifically, the lifecycle of software components can be managed separately from that of their hosting containers. It is even made possible to have multiple components running in a single container for better inter-component communication latency and resource utilization. It is also shown that the orchestration process is further automated to allow for a partial topology description [28]. Given an incomplete specification of TOSCA applications, where only application-specific components are defined and other aspects are left unspecified, the proposed scheme can complete the TOSCA representation by automatically filling in unspecified parts. These research efforts focus on TOSCA orchestration systems tailored for container-based runtime environments. According to these proposals, TOSCA application components are mapped to Docker containers, and container orchestration technologies like Kubernetes and Docker Swarm are exploited as a base to build up a full-fledged orchestration system. Our proposal differs from these efforts in that TOSCA applications are mapped to underlying OAM entities. As a result, it can benefit from role-awareness and higher-level orchestration features enabled by OAM technology. To the best knowledge of the authors, we are the first to explore the possibility of the integration of two prominent orchestration technologies of TOSCA and OAM.

Another noteworthy effort is a layering approach where cloud management solutions at the bottom like CAMP and OCCI are exploited to deploy and orchestrate TOSCA application models at the top [22,29–31]. CAMP is a standardization effort at Platform-as-a-Service(PaaS) layer [32], while OCCI can be viewed as an IaaS-layer equivalent. A framework proposed by Fabian et al. [30] introduced a model-driven cloud orchestration scheme based on TOSCA and OCCI. OCCI provides the standardization of common APIs for Infrastructure-as-a-Service (IaaS) providers, while TOSCA focuses on defining application topology and their orchestrations. TOSCAMP architecture was also proposed by Alexander et al. [31], which explored the possibility for orchestrating cloud applications through an integration of TOSCA and CAMP. Trans-cloud system [22] is similar to TOSCAMP in that it supports cross-cloud management and orchestration by combining two OASIS standards of TOSCA and CAMP. However, a difference lies in the fact that Trans-cloud proposal provides unified orchestration support for both IaaS and PaaS services. It is also noted that the Trans-cloud system is extended to a fault-aware orchestration scheme [29] by which failures of application components can be automatically recovered thanks to the knowledge of the application topology and inter-dependencies among the components. After transforming TOSCA Service Templates into either OCCI or CAMP deployment configurations by a model translator, OCCI- or CAMP-compatible runtime fulfills the deployment and management of the applications. However, without support for orchestration modeling at a proper abstraction level, it would be too complicated and time-consuming for application builders to perform their orchestration tasks. Our proposed method introduces a capability layer over IaaS or PaaS resources to enable a role-aware orchestration framework that hides irrelevant details of capability configurations.

As an open-source cloud orchestration framework, Cloudify introduces a domain-specific language based on TOSCA [33]. Cloudify aims to define and deploy application

templates that declare the configuration and interaction of cloud services, regardless of cloud infrastructure providers. In addition, Cloudify provides a Kubernetes plugin to interact with the Kubernetes cluster. It describes the configuration and the entire life cycle of containerized applications in the form of Kubernetes API at a low-level infrastructure layer [34]. By contrast, to save users from a burden of repetitive configuration specifics, our approach allows for model applications based on platform capabilities instead of infrastructure or underlying platform resources.

KubeVela is an emerging cloud platform engine based on Kubernetes and Open Application Model [35]. As a runtime implementation conforming to the OAM specification, it is designed to pursue consistent application delivery across clouds and on-premise infrastructures using Kubernetes as a common abstraction layer. KubeVela serves as an application platform that assists platform builders in constructing DevOps platforms rather than a cloud application orchestration system. It is different from our approach in that we leverage the capability of TOSCA to build an orchestration framework equipped with several features that are not supported by KubeVela, such as application topology, component interface, and workflow definition.

Different container platforms are also considered for TOSCA-based orchestration of cloud applications. For instance, a TOSCA orchestration system is built over container clusters running on Mesos [36]. An orchestration system can be built up by augmenting native container runtimes and/or by exploiting accompanying container orchestration features. This work itself builds up a container orchestration on top of Mesos, while a native container orchestration layer such as Kubernetes and Docker Swarm could be exploited as much as possible to deliver a full-fledged orchestration systems as reported in [23].

6. Conclusions

This study proposes and explores a methodology for orchestrating cloud applications built on DevOps platforms by combining two prominent orchestration standards of TOSCA and OAM. Our orchestration system design does not only support cloud application modeling, but it also provides a viable solution for deploying and managing the applications on the target runtime platform. By defining a set of capabilities as modeling constructs, our scheme allows TOSCA-based application topology itself and its orchestration needs to be specified in a way to provide a more targeted support for different needs of involved individuals, e.g., application developers and operators. With support for advanced orchestration features such as application topology model, platform capability abstraction, and role-awareness for cloud application orchestration, our approach significantly reduces the complexity of application orchestration tasks on diverse cloud platforms.

We have demonstrated the feasibility and effectiveness of the proposed approach by using a Kubernetes-based prototype implementation. More importantly, our evaluation study shows the promising results that an ever-increasing complexity of cloud application orchestration can be effectively tamed by our scheme.

Author Contributions: Conceptualization, C.L. and E.K.; Funding acquisition, C.L. and S.C.; Investigation, Y.W. and S.R.; Methodology, Y.W. and C.L.; Project administration, C.L.; Software, Y.W. and S.R.; Supervision, E.K. and S.C.; Validation, Y.W., S.R., and S.C.; Visualization, Y.W. and S.R.; Writing—original draft, Y.W.; Writing—review and editing, C.L., E.K., and S.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Research Foundation of Korea (NRF) funded by the Korea government (MSIT) (No. 2020R1A2B5B01001758) and by the Korea Meteorological Administration Research and Development Program under Grant KMI 2021-01310.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

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

References

1. Kratzke, N.; Quint, P. Understanding Cloud-native Applications after 10 Years of Cloud Computing—A systematic Mapping Study. *J. Syst. Softw.* **2017**, *126*, 1–16. [CrossRef]
2. Pahl, C. Containerization and the PaaS Cloud. *IEEE Cloud Comput.* **2015**, *2*, 24–31. [CrossRef]
3. Ebert, C.; Gallardo, G.; Hernantes, J.; Serrano, N. DevOps. *IEEE Softw.* **2016**, *33*, 93–100. [CrossRef]
4. Callanan, M.; Spillane, A. DevOps: Making It Easy to Do the Right Thing. *IEEE Softw.* **2016**, *33*, 53–59. [CrossRef]
5. Lwakatare, L.; Kuvaja, P.; Oivo, M. Dimensions of devops. In Proceedings of the International Conference on Agile Software Development, Helsinki, Finland, 25–29 May 2015; pp. 212–217.
6. Baur, D.; Seybold, D.; Griesinger, F.; Tsitsipas, A.; Hauser, C.; Domaschka, J. Cloud Orchestration Features: Are Tools Fit for Purpose? In Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing, Limassol, Cyprus, 7–10 December 2015; pp. 95–101.
7. OASIS Standard. Topology and Orchestration Specification for Cloud Applications Version 1.0. 2013. Available online: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (accessed on 14 March 2021).
8. Binz, T.; Breiter, G.; Leymann, F.; Spatzier, T. Portable Cloud Services Using TOSCA. *IEEE Internet Comput.* **2012**, *16*, 80–85. [CrossRef]
9. Képes, K.; Breitenbücher, U.; Fischer, M.; Leymann, F.; Zimmermann, M. Policy-Aware Provisioning Plan Generation for TOSCA-based Applications. In Proceedings of the 11th International Conference on Emerging Security Information, Rome, Italy, 10–14 September 2017; pp. 142–149.
10. Open Application Model. Available online: https://github.com/oam-dev/spec (accessed on 14 March 2021).
11. Kubernetes. Available online: https://kubernetes.io (accessed on 14 March 2021).
12. Li, Z.; Zhang, Y.; Liu, Y. Towards a Full-stack Devops Environment (Platform-as-a-Service) for Cloud-hosted Applications. *Tsinghua Sci. Technol.* **2017**, *22*, 1–9. [CrossRef]
13. Casalicchio, E. A study on performance measures for auto-scaling CPU-intensive containerized applications. *Clust. Comput.* **2019**, *22*, 995–1006. [CrossRef]
14. Nguyen, T.; Yeom, Y.; Kim, T.; Park, D.; Kim, S. Horizontal pod autoscaling in Kubernetes for elastic container orchestration. *Sensors* **2020**, *20*, 4621. [CrossRef] [PubMed]
15. Adams, B.; Bellomo, S.; Bird, C.; Marshall-Keim, T.; Khomh, F.; Moir, K. The practice and future of release engineering: A roundtable with three release engineers. *IEEE Softw.* **2015**, *32*, 42–49. [CrossRef]
16. Ferry, N.; Rossini, A.; Chauvel, F.; Morin, B.; Solberg, A. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In Proceedings of the 6th IEEE International Conference on Cloud Computing, Santa Clara, CA, USA, 28 June–3 July 2013 ; pp. 887–894.
17. Puccini. Available online: https://github.com/tliron/puccini (accessed on 14 March 2021).
18. OAM Kubernetes Runtime. Available online: https://github.com/crossplane/oam-kubernetes-runtime (accessed on 14 March 2021).
19. Kubernetes Operator Pattern. Available online: https://kubernetes.io/docs/concepts/extend-kubernetes/operator (accessed on 14 March 2021).
20. Prometheus. Available online: https://prometheus.io (accessed on 14 March 2021).
21. KEDA. Available online: https://keda.sh (accessed on 14 March 2021).
22. Carrasco, J.; Durán, F.; Pimentel, E. Trans-cloud: CAMP/TOSCA-based Bidimensional Cross-cloud. *Comput. Stand. Interfaces* **2018**, *58*, 167–179. [CrossRef]
23. Bogo, M.; Soldani, J.; Neri, D.; Brogi, A. Component-aware Orchestration of Cloud-based Enterprise Applications, from TOSCA to Docker and Kubernetes. *Softw. Pract. Exp.* **2020**, *50*, 1793–1821. [CrossRef]
24. Bellendorf, J.; Mann, Z. Cloud Topology and Orchestration Using TOSCA: A Systematic Literature Review. In *European Conference on Service-Oriented and Cloud Computing*; Springer: Cham, Switzerland, 2018; Volume 11116, pp. 207–215.
25. Tomarchio, O.; Calcaterra, D.; Modica, G. Cloud Resource Orchestration in the Multi-cloud Landscape: A Systematic Review of Existing Frameworks. *J. Cloud Comput.* **2020**, *9*, 1–24. [CrossRef]
26. Wang, Y.; Lee, C. A Role-Based Orchestration Approach for Cloud Applications. In Proceedings of the International Conference on Information Networking (ICOIN), Jeju Island, Korea, 13–16 January 2021; pp. 693–698.
27. Brogi, A.; Rinaldi, L.; Soldani, J. TosKer: Orchestrating Applications with TOSCA and Docker. *Commun. Comput. Inf. Sci.* **2020**, *824*, 130–144.
28. Brogi, A.; Neri, D.; Rinaldi, L.; Soldani, J. Orchestrating incomplete TOSCA applications with Docker. *Sci. Comput. Program.* **2018**, *166*, 194–213. [CrossRef]
29. Brogi, A.; Carrasco, J.; Durán, F.; Pimentel, E.; Soldani, J. Robust Management of Trans-Cloud Applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 219–223.
30. Glaser, F.; Erbel, J.; Grabowski, J. Model Driven Cloud Orchestration by Combining TOSCA and OCCI. In Proceedings of the 7th International Conference on Cloud Computing and Services Science, Porto, Portugal, 24–26 April 2017; pp. 644–650.
31. Alexander, K.; Lee, C.; Kim, E.; Helal, S. Enabling End-to-End Orchestration of Multi-Cloud Applications. *IEEE Access* **2017**, *5*, 18862–18875. [CrossRef]

32. OASIS Standard. Cloud Application Management for Platforms Version 1.1. 2014. Available online: http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html (accessed on 14 March 2021).
33. Cloudify. Available online: https://cloudify.co (accessed on 14 March 2021).
34. Cloudify Kubernetes Plugin. Available online: https://cloudify.co/kubernetes (accessed on 14 March 2021).
35. KubeVela. Available online: https://kubevela.io (accessed on 14 March 2021).
36. Kehrer S.; Blochinger, W. TOSCA-based Container Orchestration on Mesos. *Comput. Sci. Res. Dev.* **2018**, *33*, 305–316. [CrossRef]