A Role-Based Orchestration Approach for Cloud Applications

Yue WANG and Choonhwa LEE
Department of Computer Science, Hanyang University,
222 Wangsimni-ro Seongdong-gu Seoul, 04763 Korea
{**wylk2019,lee**}**@hanyang.ac.kr**

2021 International Conference on Information Networking (ICOIN) | 978-1-7281-9101-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICOIN50884.2021.9333919

***Abstract*—With a rapid growth of cloud service offerings, application developers face a critical challenge to adequately orchestrate their applications on diverse cloud platforms. A desirable orchestration solution should ensure that the applications are deployed, configured, and operated in an appropriate way to the given situation. This paper proposes a model-driven orchestration approach that specifies particular tasks depending on the roles of involved individuals. By keeping the concerns of applications developers and operators separated, our proposed scheme allows them to better focus on what are expected from themselves to achieve the overall orchestration goal. To show the feasibility and effectiveness of the proposal, we present a proof-of-concept system implementation and demonstrate its orchestration operations on Kubernetes platform.**

***Index Terms*—Kubernetes, DevOps, TOSCA, Orchestration, OAM**

I. Introduction

Cloud-native applications rely on increasingly complicated technologies, which poses significant challenges to the design, development, and delivery of them [1]. To help application builders leverage diverse technologies, PaaS(Platform-as-a-Service) is introduced as a cloud service model that provides a configurable application 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]. With a combination of well-defined APIs, clear abstractions, and comprehensive extension, Kubernetes[1] has been considered as a solid foundation to build a DevOps platform [5]. Accompanied by the industry's wide acceptance, the Kubernetes ecosystem has already provided DevOps platforms with a wealth of essential functions, such as CI/CD pipeline, release strategy, image packaging, etc.

To keep platform resources or capabilities configurable, various numbers of configuration fields or parameters should be provided. Rich fields or parameters should bring higher flexibility and maintainability. However, not all the fields or parameters may be of interest to application builders. In contrast, most configurable fields are to be acknowledged and consumed by platform builders instead of application builders. A single application or component is supported by several platform capabilities, including workloads and requisite operational characteristics, which entails that application builders are expected to handle all configuration fields or parameters of those underlying platform resources. Obviously, it is a time-consuming and error-prone task for application builders. In other words, the application builders are likely overwhelmed with the configuration task. For cloud-native application orchestrations, the problem becomes even worse, because orchestrators must deal with all configurable content of resources [6].

In general, DevOps tasks involve two roles of developers and application operators. Developers are in charge of implementing real-world business logic, while application operators are responsible for keeping applications up and running. Both of them utilize platform capabilities to fulfill their goal. Both of them are also the main members of application builders. However, it is likely for them to find it challenging to identify and deal with particular orchestration fields or parameters that are relevant to them. If the distinct roles are left to perform their orchestration tasks solely on their own without any coordination, they may end up with making conflicting decisions. Therefore, it is crucial to make a clear distinction among different roles and tasks for orchestration, so that they can exist in a harmonious way to achieve the ultimate, common goal of cloud-native applications orchestration.

The remainder of this paper is organized as follows. Section 2 presents a sample scenario to motivate our work. Section 3 presents the architectural design of our proposed system and its prototype implementation. Our evaluation efforts and results are provided in Section 4. After discussing related work in Section 5, the paper concludes in Section 6.

II. Motivational Scenario

To motivate our work, we consider a flight tracker application being deployed on Kubernetes. The flight tracker queries and returns flight information according to user requests. To deploy the flight tracker on Kubernetes, we need a Kubernetes Deployment resource to provide workloads within which the containerized application runs. To expose the service of Deployment, a Kubernetes Service resource associated to the Deployment is also created. In this scenario, Deployment and its associated Service are a main concern for application developers who are responsible to choose appropriate workloads to run the applications.

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No.2020R1A2B5B01001758).

[1]https://kubernetes.io/

To make the application elastic to dynamically changing demands, the infrastructure is empowered with auto-scaling support. In our scenario, we create an operational capability, metric auto-scaling, which belongs to the application operator's domain. It is a component to enable auto-scaling based on custom monitoring metrics such as requests per second. For our use-case study presented in the paper, two popular tools, Prometheus[2] and KEDA(Kubernetes-based Event Driven Autoscaling)[3], are employed to fulfill this scaling capability on Kubernetes platform. Prometheus captures monitoring data from target Kubernetes Services to calculate metric results according to specified rules. KEDA fetches the metrics from Prometheus and sets up a metric server in the Kubernetes cluster. Finally, we can use a Kubernetes HPA (Horizontal Pod Autoscaler) associated with target Kubernetes Deployment to scale out or in based on the metrics from the metric server. The overall system structure of the case study is illustrated in Fig. 1.

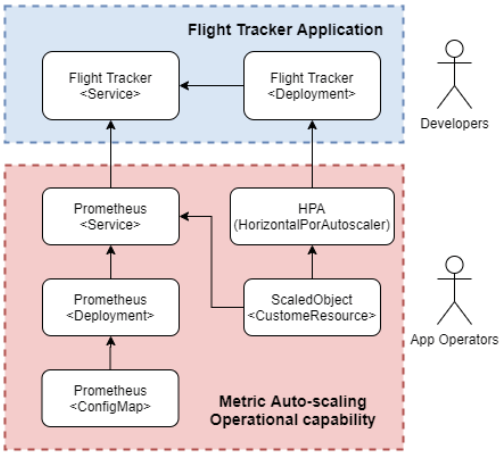


Fig. 1: Composition of flight tracker application.

Each time metric auto-scaling is applied to an application, a set of Kubernetes resources are going to be created as summarized in Table I. While many configuration fields have to be authored repetitively, only a small portion of them are typically required to be set, such as threshold value and scaling metrics rules. Other fields, like association relationships between Deployment and Service of Prometheus, remain unchanged. They are not concerned about by application operators. Therefore, the main idea of our proposed approach is to separate the fields or parameters of interest to application operators and encapsulate them into well-designed model entities. Consequently, from the viewpoint of application operators, the metric auto-scaling component only consists of a few but crucial configuration fields.

III. Role-Based Orchestration Approach

In order to support the separation of concerns during cloud orchestration tasks, we propose a novel model-driven orchestration system, which is based on OAM[4] application model and TOSCA orchestration model [8]. Through an intermediate abstraction layer between platform resources and applications built upon them, our approach shields a large portion of resources composition capabilities from applications builders. Furthermore, the extracted concerns are categorized according to two roles: developers and application operators. It enables independent management and precise coordination on application orchestration regarding the different concerns. Consequently, it is made possible to build an orchestration-friendly DevOps platform as depicted in Fig. 2.

TABLE I: Kubernetes Resources of metric auto-scaling

Kubernetes resources	Owner
Deployments.apps.k8s.io/v1	Prometheus
Services.core.k8s.io/v1	Prometheus
ConfigMaps.core.k8s.io/v1	Prometheus
ScaledObjects.keda.core/v1alpha1	KEDA
HorizontalPodAutoscalers.autoscaling.k8s.io/v1	KEDA

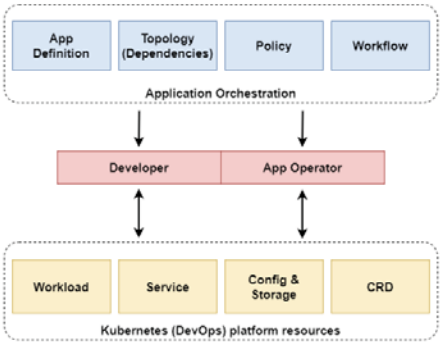


Fig. 2: Role-based orchestration approach.

After reviewing the state of the art for cloud orchestration technologies, this section details our approach that allows application developers and operators to focus on performing their expected tasks.

A. Technology for Cloud Application Orchestration

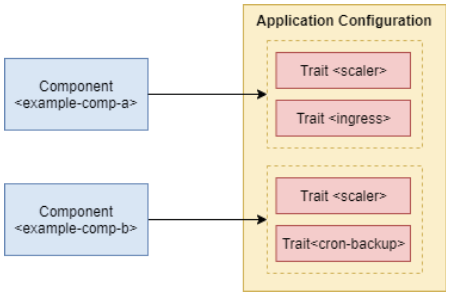
Our orchestration approach leverages existing standards, the most relevant ones of which are TOSCA and OAM.

First, OASIS TOSCA (OASIS Topology and Orchestration Specification for Cloud Applications) is an OASIS standard aiming to enable portable specification of cloud applications. It allows application builders to describe the topology and orchestration behavior of cloud applications within a declarative Service Template specification. The Service Template is composed of the application's components which are modeled as Node Templates that are interconnected via Relationship Templates.

OAM(Open Application Model) is a platform-agnostic specification for building cloud-native applications. Distinguishing between the parts that developers are responsible for

[2]https://prometheus.io/
[3]https://keda.sh/
[4]https://github.com/oam-dev/spec

and the parts that application operators are responsible for, OAM provides a standard and extensible framework to create application-centric platforms on top of underlying runtime systems such as Kubernetes. In the domain of OAM, an application is composed of two parts, one is the workload carrying running programs, the other one is the specification of operational requirements for the workloads. OAM Components are introduced to permit developers to declare the workloads and its properties. OAM Trait is a discretionary runtime overlay that augments a component workload type with additional operational features. Application Configuration is a resource that declares how all parts of an application are to be instantiated and configured. Fig. 3 shows the structure of an OAM Application Configuration composed of two components and its corresponding OAM YAML file snippet.



(a) Structure of OAM Application Configuration

```yaml
apiVersion: core.oam.dev/v1alpha2
kind: ApplicationConfiguration
metadata:
  name: example-app
spec:
  components:
    - componentName: example-comp-a
      traits:
        - name: scaler
          replicas: 2
        - name: ingress
          # properties of ingress trait
    - componentName: example-comp-b
      traits:
        - name: scaler
          replicas: 3
        - name: cron-backup
          # properties of cron-backup trait
```

(b) YAML snippet of OAM Application Configuration

Fig. 3: Open Application Model Specifications

B. Role-Based Orchestration System

First, we present the architectural design of our role-based orchestration system. Our approach is to enable role-awareness in both TOSCA and OAM domain models, capturing distinct concerns in their respective model domain. More specifically, TOSCA model specification can be enhanced to make it role-aware, while OAM takes care of how to deploy and manage applications on compatible runtime platforms in accordance with the orchestration model comprising of topology, policies, and workflows defined in TOSCA Service Template.

As illustrated in Fig. 4, our architectural design consists of a TOSCA compiler based on Puccini[5] and an extended OAM Kubernetes runtime[6] linked together by a conversion engine capable of translating TOSCA Service Templates into deployable OAM Application Configurations.

Here is a brief on the system workflow. As the initial input, TOSCA Service Templates carrying a specific application orchestration will be processed by TOSCA compiler. TOSCA YAML files are parsed into typed Go structs if basic validation based on TOSCA meta-model is passed. Afterward, the model conversion engine will consume these typed Go structs as input to convert and generate OAM compatible objects. Following the pre-defined conversion strategies, TOSCA Service Templates will be converted into a bundle of OAM configuration YAML files. Finally, OAM Kubernetes runtime will deploy and orchestrate the application in a Kubernetes cluster according to these configuration files.

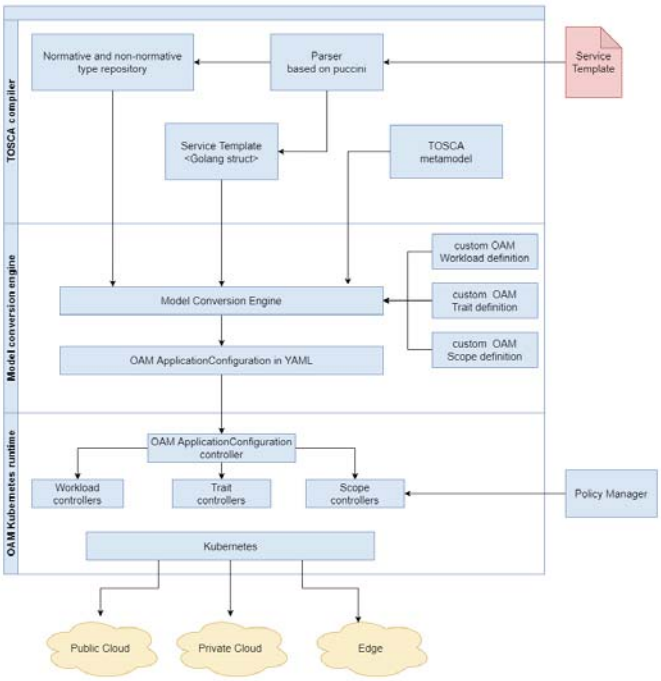


Fig. 4: Role-based orchestration system architecture.

1) TOSCA Compiler: To infuse different concerns into an orchestration model, we define a set of custom TOSCA types, including Nodes, Capabilities, Relationships, Policies, etc. With these extensions, orchestration platform capabilities can be modeled as TOSCA Nodes. They are informative and extensible to describe capabilities. Some of them are runnable workloads concerned about by developers and operational characteristics of interest to application operators. According to our approach, developers are responsible for choosing appropriate Node types referring to runnable workloads and declaring dependencies among them. Besides custom Node types relating to operational capabilities, we also provide application operators with custom Relationship types that enable portable and flexible descriptions of relationships between

[5]https://github.com/tliron/puccini

[6]https://github.com/crossplane/oam-kubernetes-runtime

components and their operational characteristics. Additionally, application builders also can apply policies according to non-functional requirements and define workflows based on the topology building blocks. Fig. 5 shows the TOSCA Service Template of our use-case scenario.

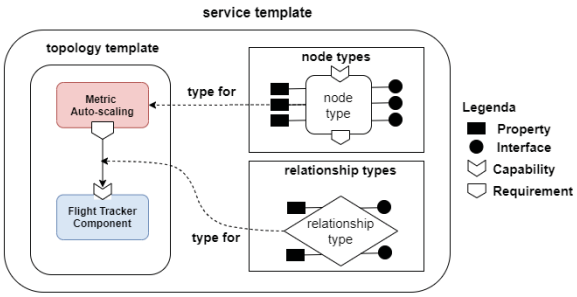


Fig. 5: TOSCA topology of flight tracker application.

In order for our extended TOSCA model to be fed to the model conversion engine as diagrammed in Fig. 4 and be deployed on the OAM runtime, we have developed an in-house TOSCA parser based on open-source project of Puccini. Besides the functions supported by Puccini, our TOSCA compiler also can recognize different roles' concerns embedded in TOSCA Service Templates and distinguish them with custom labels. Labeled TOSCA types or templates will be consumed by the model conversion engine to construct OAM Application Configurations. The parser consists of three primary parts.

- Parser module based on Puccini is used to process the TOSCA Service Template.
- Role Label module recognizes and labels TOSCA entities according to roles' concerns.
- A Web server module allows for access to the parser through RESTful APIs.

2) Extension to OAM Kubernetes Runtime: To deploy and manage applications on Kubernetes according to an orchestration model, we designed an orchestration runtime platform based on the OAM Kubernetes runtime. It aims to quickly build a customized OAM platform rather than starting from scratch. It consists of a set of Kubernetes controllers capable of creating OAM resources such as workloads, traits, scopes, etc., and reconciling them into the desired state declared in OAM Application Configurations. For each TOSCA custom Node types referring to platform capabilities, we developed corresponding Kubernetes custom resources and controllers to fulfill the capabilities, e.g., metric auto-scaling capabilities in our use-case study, on Kubernetes. Additionally, we added two significant orchestration features: policy and workflow that OAM runtime does not support.

Application builders can have non-functional requirements and policies that should hold for any of their applications or components [9]. TOSCA Policy types represent a logical grouping of TOSCA Nodes with an implied relationship and need to be orchestrated or managed together to achieve some results. We create an independent Policy Manager module to deal with TOSCA Policy based on OAM Scope mechanism. Generally, TOSCA Policy indicates a group of Nodes and constraints assigned to them. Correspondingly, the OAM runtime manages Components in the form of a group with Scopes and controls Component's behavior with Traits. Therefore, the policy manager will convert TOSCA Policy types into OAM Scopes capable of injecting data into Components or attaching specific Traits to Components at runtime.

TOSCA provides two different approaches of workflows to deploy and manage applications at runtime: declarative workflows and imperative workflows [10]. Since OAM argues for a purely declarative approach to specify applications, it provides no support for imperative workflows. We create a workflow engine in our prototype implementation to resolve and control imperative workflow behaviors defined in TOSCA Service Templates. Specifically, the workflow engine is able to consume parsed TOSCA workflow entities and delegates operations being applied to OAM Components or Traits to an OAM-Kubernetes runtime [7].

3) Model Conversion Engine: To convert TOSCA Service Templates to OAM Application Configurations, the model conversion engine uses a set of mapping rules that define mapping relationships between two models. Table II summarizes mapping relationships between TOSCA and OAM entities. In TOSCA Service Template, Node types concerned about by developers are converted into OAM Components. As the instance of Node types, Node templates are converted into OAM Component instances in Application Configuration. As for Node types and Node templates of interest to application operators, the conversion engine will translate them into OAM Trait Definitions and Trait instances attached to Components.

TABLE II: TOSCA-to-OAM Entity Mapping

TOSCA Types/Templates	OAM Entities
Node Type (developer)	Component
Node Template (developer)	Component
Node Type (operator)	Trait Definition
Node Template (operator)	Trait
Relationship Type(operator)	Trait Definition
Relationship Template(operator)	Trait
Group Type and Policy Type	Scope and Trait

One of TOSCA's significant features is to provide a portable and flexible model on relationships between application components. Whereas OAM specification has no support for modeling relationships, our orchestration system handles relationships in two different ways: relationship between two components and relationship between a component and its operational characteristic. It is noted that these two are modeled in TOSCA Service Template as specific Relationship types connecting Nodes.

For relationships between a component and its operational characteristics, we design a custom TOSCA Relationship type derived from a TOSCA Relationship type, "tosca.relationships.AttachesTo". When this Relationship type

[7]https://github.com/oam-dev/kubevela

is found, the conversion engine adds the Trait that represents the source Node to the Component for the target Node. For relationships between two components, usually depending on each other, we employed several OAM Traits to implement these relationships. Specifically, OAM Traits can retrieve and inject data into the Component to which they attach. Fig. 6 shows the two cases of relationship conversion.

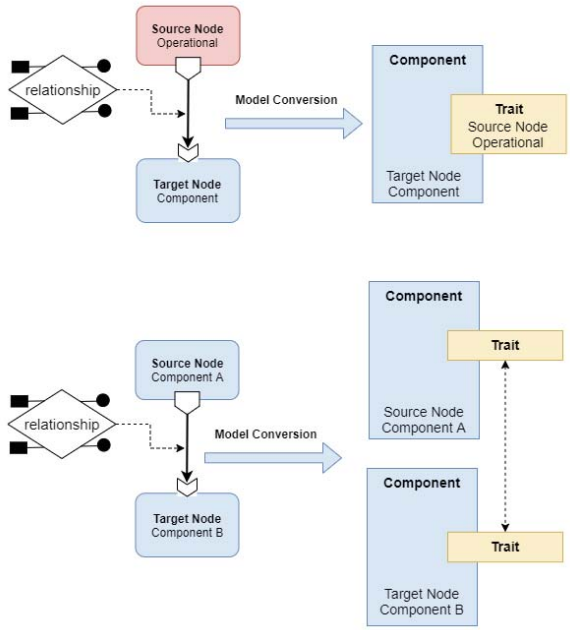


Fig. 6: Relationship Conversion from TOSCA to OAM.

```
topology_template:
  node_templates:
    flight-service-comp:
        type: dcc.workload.Containerized
        properties:
            containers:
            - name: flight-service-app
              image: dcc-dev/flight-service-app
              imagePullPolicy: Always
              ports:
              - containerPort: 8080
                name: flightsvc
    metric-hpa-trait:
        type: dcc.operational.capability.MetricHPA
        properties:
            promQuery: sum(rate(http_requests[2m]))
            promThreshold: 3
        requirements:
        - flight-service-comp:
          relationship: attachTo
```

Fig. 7: TOSCA specification for flight tracker application.

```
apiVersion: core.oam.dev/v1alpha2
kind: ApplicationConfiguration
metadata:
  name: flight-service-app
spec:
  components:
    - componentName: flight-service-comp
      traits:
        - trait:
            apiVersion: extend.hanyang.ac.kr/v1alpha1
            kind: MetricHPATrait
            spec:
              promQuery: sum(rate(http_requests[2m]))
              promThreshold: 3
---
apiVersion: core.oam.dev/v1alpha2
kind: Component
metadata:
  name: flight-service-comp
spec:
  workload:
    apiVersion: core.oam.dev/v1alpha2
    kind: ContainerizedWorkload
    spec:
      containers:
        - name: flight-service
          image: dcc-dev/flight-service-app
          imagePullPolicy: Always
          ports:
              - containerPort: 8080
                name: flightsvc
```

Fig. 8: Converted OAM Application Configuration.

IV. Evaluation

We performed a set of experiments using our prototype implementation to deploy a sample application. In order to show that our approach is capable of supporting different needs and concerns, we choose to demonstrate the metric auto-scaling capability.

A. Model Conversion

Fig. 7 is the TOSCA Topology Template of our motivational scenario application. It consists of three main parts: flight service Node template referring to the component, metric auto-scaling Node template referring to the operational capability, and Relationship template connecting these two Nodes.

The Service Template YAML file is processed by TOSCA compiler and converted into OAM Application Configurations in Fig. 8 by our model conversion engine.

B. Deployment and metric auto-scaling

According to the OAM Application Configuration, our orchestration runtime deploys the application that comprises of flight service component and a metric auto-scaling capability on a Kubernetes cluster. As shown in Fig. 7, we have defined rules for metric auto-scaling capability: *"promQuery: sum(rate(http_requests[2m]))"* and *"promThreshold: 3"*. It specifies an auto-scaling behavior that monitors the number of requests per second in two minutes of the service and increases the replicas, when the fluctuation threshold exceeds three. For this experiment, we gradually increased the number of requests to the flight service application and recorded the number of replicas of the component. The results are plotted in Fig 9. As the number of requests increases, the requests per second measured by Prometheus increase as well. When the threshold of 3 requests per second is crossed, a new replica is created. The experiment was run for an interval of 240 seconds. With an increasing number of arriving requests, more replicas are added by the rule defined in the model.

V. Related work

Several model-driven orchestration systems exist [7], including OpenTOSCA [11], TosKer [12], [13], Cloudify [14], and TOSCAMP [15]. While these systems adequately use TOSCA for orchestration on the Kubernetes platform, they construct orchestration models composed of entities representing Kubernetes basic resources. Therefore, these systems still have to deal with a large number of configuration content of Kubernetes resources. By contrast, to get users rid of enormous and repetitive configuration content, our approach allows to

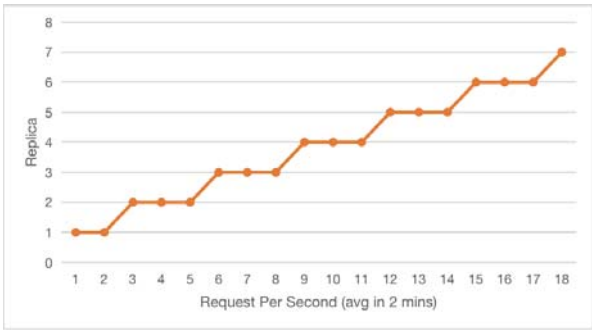


Fig. 9: Scaling of replicas.

model applications based on platform capabilities instead of underlying basic resources. Our orchestration system design aims to support application modelling as well as provide a means of deploying and managing the application on the target runtime platform such as Kubernetes.

VI. Conclusion

This study has proposed and explored a methodology for orchestrating applications built on DevOps platforms through a combination of two prominent orchestration standards: TOSCA and OAM. Owing to our role-based orchestration approach, DevOps platforms can be now enhanced to provide a more targeted support for different needs and concerns of involved players, e.g., application developers and operators. Through encapsulating miscellaneous platform resources into capability entities, our approach retains a small number of configuration content that is most concerned by users during orchestration. As a result, an increasing complexity of cloud application orchestration can be curtailed significantly. We have also demonstrated the feasibility and effectiveness of our proposed approach using a Kubernetes-based prototype implementation.

References

[1] Kratzke, N., Quint, P. C. (2017). Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. Journal of Systems and Software, 126, 1–16.
[2] Pahl, C. (2015). Containerization and the PaaS Cloud. IEEE Cloud Computing, 2(3), 24–31.
[3] C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, ”DevOps,” in IEEE Software, vol. 33, no. 3, pp. 94-100, May-June 2016
[4] Zhu, L., Bass, L., DevOps and Its Practices. IEEE Software, 33(3), 32–34.
[5] Li, Zhenhua, Yun Zhang, and Yunhao Liu. ”Towards a full-stack devops environment (platform-as-a-service) for cloud-hosted applications.” Tsinghua Science and Technology 22.01 (2017): 1-9.
[6] 9.Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C. B., Domaschka, J. (2015). Cloud Orchestration Features: Are Tools Fit for Purpose? Proceedings - 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing, UCC 2015, 95–101.
[7] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, ”Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems,” in Proc. 6th Int. Conf. Cloud Comput., Santa Clara, CA, USA, 2013, pp. 887–894.
[8] TOSCA Simple Profile in YAML Version 1.3, OASIS Standard TOSCA-Simple-Profile-YAML-v1.3, 2020.
[9] C. Dimoulas, S. Moore, A. Askarov, and S. Chong, ”Declarative policies for capability control,” in Proc. Comput. Secur. Found. Symp., Vienna, Austria, 2014, pp. 3–17.
[10] Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F., Wettinger, J. (2017). Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. Proceedings of the 9th International Conferences on Pervasive Patterns and Applications, 22–27.
[11] Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S. (2013). OpenTOSCA - A runtime for TOSCA-based cloud applications. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8274 LNCS, 692–695.
[12] Brogi, A., Rinaldi, L., Soldani, J. (2018). TosKer: Orchestrating Applications with TOSCA and Docker. In Communications in Computer and Information Science (Vol. 824).
[13] Bogo, M., Soldani, J., Neri, D., Brogi, A. (2020). Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes. Software - Practice and Experience, (January), 1793–1821. https://doi.org/10.1002/spe.2848
[14] Kim, D., Muhammad, H., Kim, E., Helal, S. and Lee, C., 2019. TOSCA-based and federation-aware cloud orchestration for Kubernetes container platform. Applied Sciences, 9(1), p.191.
[15] Alexander, K., Lee, C., Kim, E., Helal, S. (2017). Enabling End-to-End Orchestration of Multi-Cloud Applications. 5.