Received July 11, 2017, accepted July 30, 2017, date of publication August 11, 2017, date of current version October 12, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2738658

Enabling End-to-End Orchestration of Multi-Cloud Applications

KENA ALEXANDER[1], CHOONHWA LEE[1], EUNSAM KIM[2], AND SUMI HELAL[3]
[1]Department of Computer Science, Hanyang University, Seoul 04763, South Korea
[2]Department of Computer Engineering, Hongik University, Seoul 121-791, South Korea
[3]CISE Department, University of Florida, Gainesville, FL 32611 USA

Corresponding author: Choonhwa Lee (lee@hanyang.ac.kr)

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea, Ministry of Science, ICT & Future Planning under Grant 2017R1A2B4010395 and Grant 2016R1D1A1A09917396.

ABSTRACT The orchestration of application components across heterogeneous cloud providers is a problem that has been tackled using various approaches, some of which led to the creation of cloud orchestration and management standards, such as TOSCA and CAMP. Standardization is a definitive method of providing an end-to-end solution capable of defining, deploying, and managing applications and their components across heterogeneous cloud providers. TOSCA and CAMP, however, perform different functions with regard to cloud applications. TOSCA is focused primarily on topology modeling and orchestration, whereas CAMP is focused on deployment and management of applications. This paper presents a novel solution that not only involves the combination of the emerging standards TOSCA and CAMP, but also introduces extensions to CAMP to allow for multi-cloud application orchestration through the use of declarative policies. Extensions to the CAMP platform are also made, which brings the standards closer together to enable a seamless integration. Our proposal provides an end-to-end cloud orchestration solution that supports a cloud application modeling and deployment process, allowing a cloud application to span and be deployed over multiple clouds. The feasibility and the benefit of our approach are demonstrated in our validation study.

INDEX TERMS Distributed computing, middleware, software architecture, model-driven development.

I. INTRODUCTION

Cloud computing is an on-going area of distributed computing that enables the delivery of applications as services over the Internet, as well as platform- and infrastructure-level computing resources. The advent of cloud computing promises to provide ‘‘users‘‘ the benefits of, among many, availability of on-demand services, elimination of up-front commitment, and pay-per use model. These benefits, however, come with the addition of inherent issues such as availability of service, performance unpredictability, resource scaling, and vendor lock-in [1], [2].

The orchestration of applications and components across cloud providers is capable of addressing some of these inherent issues regarding cloud computing. However, the orchestration of applications and components itself is not an easy task to accomplish [3]. The orchestration of applications and resources in the cloud involves dynamically deploying, managing, and maintaining those aforementioned components in and across multiple heterogeneous cloud platforms. As it is possible that cloud providers platforms may be built using varying technologies and APIs [4], it is clear that standardization can provide the answer to orchestration across these heterogeneous cloud platforms [5]–[7].

Currently, the de-facto standard for cloud application modeling and orchestration, OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) [8], provides a method of defining the topology of cloud applications through the use of an XML DSL coupled with the detailed plans for the management of the applications. More recently, the TOSCA simple profile in YAML was produced, providing a declarative method for defining cloud application topologies via TOSCA [9]. This declarative approach negates the need for specifying deployment and or management plans within a TOSCA Service Template, thus making TOSCA a fully declarative specification.

OASIS CAMP (Cloud Application Management for Platforms), is another specification whose primary purpose is to simplify cloud application deployment and management [6], [10]. It also uses a declarative deployment plan defined in YAML in order to specify the artifacts that should be deployed as well as the services that should be used to fulfill those artifact deployments. CAMP serves as an API

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

between the developers and cloud providers and provides a standard way for deploying and managing cloud applications. However, the orchestration of applications across multiple providers was not one of its deliverables.

In this paper, we present TOSCAMP (TOSCA + CAMP) which is our proposed solution for providing a standards-based, end-to-end cloud orchestration solution by combining the standards of TOSCA and CAMP. By building upon mainstream standards relevant to cloud application deployment and management and orchestration, we can simplify the work required to deploy and orchestrate applications across multiple heterogeneous cloud providers.

Therefore, the major contributions presented in this paper are as follows. (1) We present a method of converting TOSCA service templates into CAMP deployment plans and consequently converting the components of a TOSCA service template into appropriate deployment and management components of a CAMP deployment plan. (2) The paper introduces our architectural design of TOSCAMP platform used to convert TOSCA Service Templates to CAMP deployment plans, and presents a prototype implementation to demonstrate our approach. (3) We validate our proposed approach using our TOSCAMP platform and analyse the performance of our orchestration solution. (4) Finally, we discuss what differentiates our approach from the state-of-the-art approaches.

The remainder of this paper is structured as follows. Section II describes our motivation and challenges. This section also contains our motivational scenario used to validate our approach. Section III presents our TOSCAMP architecture whose performance evaluation results are presented in the following section IV. Finally, we discuss related and future works in Section V, after which we conclude the paper.

II. MOTIVATION AND CHALLENGES

To illustrate our approach, we consider deploying a modified version of the WebServer-DBMS WordPress case study example [9]. The application used in this scenario comprises a Web application front-end which, in this case, is the WordPress application. The WordPress application is installed onto the server via a supplied installation script. Apart from the installation script, configuration scripts are also used to configure the WordPress application which is deployed in a clustered configuration. In this configuration, the front-end cluster may consist of one or more WordPress servers. Each node of the web cluster is in turn provisioned on a compute node provided by an IaaS cloud provider. The WordPress Web Application must connect to a database that is hosted on a SQL DBMS server. These data management components are also provisioned on a compute node on a cloud provider. Fig. 1 depicts the topology of the web application.

To leverage the features of orchestration, our example application must be deployed across two heterogeneous cloud providers. It is noted that the deployment scenario entails a more advanced form of orchestration support beyond current orchestration technologies in that the application spans multiple clouds. Moreover, there are further constraints that should be adhered to:

- The Web application and the database should not be deployed on the same provider's cloud platform.
- The web application front-end should be deployed in a load-balanced configuration.

Our TOSCAMP approach must allow for such an application to be deployed to the cloud as well as provide a means of declaratively specifying the management criteria for maintaining the application within the constraints given. There are, however, some challenges that have to be overcome in order to realize this solution.

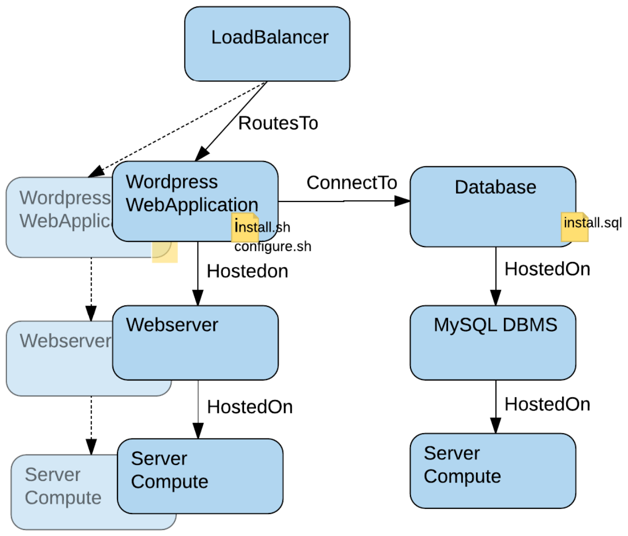


FIGURE 1. Topology of WordPress web application.

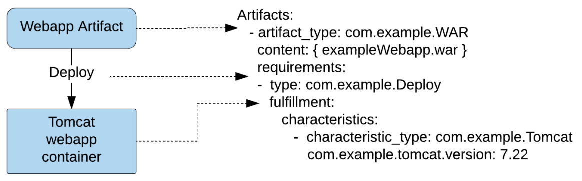
A. PORTABLE APPLICATION TOPOLOGY SPECIFICATION

A standardized approach to application orchestration must adhere to the caveat that the topology of the application should be described in a form that is portable and interoperable across compliant providers. To address this challenge, we utilize TOSCA as the standard Domain Specific Language (DSL) for specifying the topology. TOSCA provides a definition for modeling the topology of applications that may be deployed to heterogeneous cloud service providers. In TOSCA, cloud application designers may define the topology of an application, also known as the application's Topology Template, within a declarative Service Template document. The topology template is composed of the application's components which are modeled as typed Nodes that are interconnected via typed Relationships. The modeled application's topology may be used to deploy the components of the application via a TOSCA orchestrator capable of interpreting the nodes and relationship types that have been used. Another salient feature of TOSCA is its ability to declare user-defined types which can be used to fulfill components within the Topology Template. While the specification provides a meta-model for describing the topology of an application, it does not define how a TOSCA compliant orchestrator may fulfill an application topology across heterogeneous cloud providers. That is, a TOSCA compliant orchestrator may be capable of deploying the components on its own or leveraging the deployment capabilities of another suitable deployment platform. More importantly, though, TOSCA documents are not embellished with provider-specific information. As a result, a TOSCA document remains portable across compliant TOSCA platforms.

B. STANDARD API FOR CLOUD PROVIDERS

Cloud providers are free to utilize any platform or API for providing their services. For example, Amazon's AWS is powered by its own private platform, whereas providers such as Rackspace makes use of OpenStack for their cloud platform. From this, it can be seen that while some of these may be open, there may be proprietary platforms as well, which present proprietary APIs for connecting and performing management tasks. Standardization is a means of bringing these disparate platforms together through a unified API. In our approach, we use OASIS CAMP in order to provide a standard means of interfacing with cloud providers.

OASIS CAMP defines the models, mechanisms, and protocols for the management of applications in and their use of a Platform as a Service (PaaS) environment [6], [10]. Unlike TOSCA, the OASIS CAMP specification describes the format of an application as well as how that application's components should be deployed to a CAMP compliant provider. The CAMP specification therefore makes use of declarative plan files written in YAML as well as a CAMP platform consisting of platform components. A CAMP deployment plan is constructed by creating a typed graph connecting artifacts and services via requirements, as seen in Fig 2.



```
Artifacts:
  - artifact_type: com.example.WAR
    content: { exampleWebapp.war }
    requirements:
    -  type: com.example.Deploy
       fulfillment:
         characteristics:
           -  characteristic_type: com.example.Tomcat
              com.example.tomcat.version: 7.22
```

FIGURE 2. Sample structure of OASIS CAMP plan.

Artifacts used in a CAMP plans are the functional aspects of an application to be deployed. For example, if we consider a simple Web application packaged as a WAR file, then the functionality of the application is contained in the WAR file and must be deployed in order for the application to be useful. This WAR file is considered to be the application Artifact and may be deployed to a provider via a CAMP platform.

C. ORCHESTRATION-AWARE DEPLOYMENT AND MANAGEMENT

Apart from being able to interface with heterogeneous providers, it is necessary to be able to deploy and manage the components of an application across those providers. The application, "as a unit", should be maintained, even if its components are distributed across various providers. In other words, components are not to be managed in silos. While OASIS CAMP is capable of deploying, and managing an application on specified cloud providers, it is incapable of orchestrating the components of the application across multiple providers. To overcome this challenge, we propose an extension to the CAMP specification through the addition of policies that will allow for the components of CAMP to be deployed and managed across heterogeneous cloud providers.

Policies provide a means of management and orchestration of complex applications over heterogeneous clouds [11], [12]. Cloud computing relies on the delivery and orchestration of decoupled, distributed services across disparate providers in order to meet consumers requirements. Policies, therefore, should be used to control the state of services in an application topology and the services used to fulfill the components of that topology [13]. Tosca's YAML specification consists of provisions for declaratively specifying policies within an application topology. In contrast, CAMP's specification does not contain provisions for the specification of policies whatsoever. To address this, we propose to extend CAMP by adding a declarative policy format, based on CAMPs YAML specification. Our proposed policy extensions for CAMP should provide declarative policies that may be associated with components of a CAMP plan. One key requirement of our extension is the policies must be declarative. Through the use of declarative policies, we can maintain the declarative structure of CAMP plans and reduce the complexity associated with orchestrating application components. We discuss our declarative policy approach in detail within Section III.

D. CONVERSION METHODOLOGY FROM TOSCA TO CAMP

While both TOSCA and CAMP were derived from the same standards body, OASIS, they serve quite different purposes due to the fact that they have different targets. Apart from having different targets, the models which their YAML documents are based on are also not identical matches. TOSCAs model consists of normative types that can be used to compose applications or extended to form new types. CAMPs model, however, do not contain a static set of normative types. Instead, a cloud provider can supply types compatible to its namespace. Therefore, trying to provide a direct translation from a TOSCA model to a CAMP model would prove to be a complicated endeavor. Thus, in order to bridge the gap between the different models of TOSCA and CAMP, we made use of ATL (Atlas Transformation Language) to perform a model-to-model translation. Our model conversion strategy is elaborated in Section III.

III. TOSCAMP ARCHITECTURE

In this section, we present the overall strategy of our approach, TOSCAMP, as well as the architecture behind our approach. The overall approach of TOSCAMP focuses on the idea that both the TOSCA and CAMP specifications make use of typed components to describe an application topology or deployment depending on the specification. TOSCA specification contains a collection of predefined, normative types that may be used or extended, in order to define the topology of an application. CAMP, on the other hand, expects that CAMP platforms will be aware of and capable of interpreting specific defined types of CAMP components. Through this knowledge of the existence of "known" types in both specifications, we were able to translate a TOSCA topology into a CAMP deployment plan, so that it may be deployed and managed in a standard manner on a cloud provider's platform.

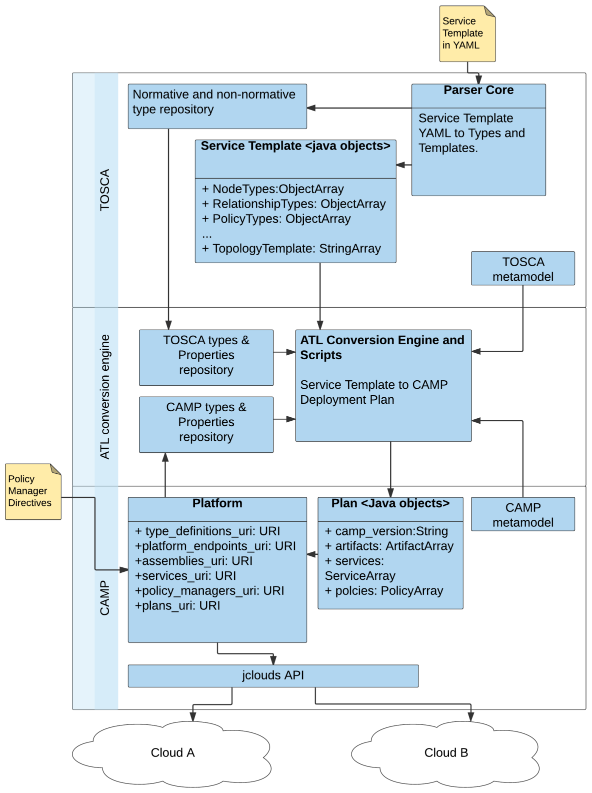

Service Template in YAML
TOSCA
Normative and non-normative type repository
Parser Core
Service Template YAML to Types and Templates.
Service Template <java objects>
+ NodeTypes:ObjectArray
+ RelationshipTypes: ObjectArray
+ PolicyTypes: ObjectArray
...
+ TopologyTemplate: StringArray
TOSCA metamodel
ATL conversion engine
TOSCA types & Properties repository
ATL Conversion Engine and Scripts
Service Template to CAMP Deployment Plan
CAMP types & Properties repository
Policy Manager Directives
CAMP
Platform
+ type_definitions_uri: URI
+platform_endpoints_uri: URI
+assemblies_uri: URI
+services_uri: URI
+policy_managers_uri: URI
+plans_uri: URI
Plan <Java objects>
+ camp_version:String
+ artifacts: ArtifactArray
+ services: ServiceArray
+ polcies: PolicyArray
CAMP metamodel
jclouds API
Cloud A
Cloud B

FIGURE 3. **TOSCAMP architecture.**

A. ARCHITECTURAL DESIGN

As illustrated in Fig. 3, the TOSCAMP architecture consists of a TOSCA parser and an extended CAMP platform joined by a conversion engine capable of translating the TOSCA topologies supplied to it into CAMP plans.

1) TOSCA PARSER

There exists quite a number of platforms that are capable of reading a TOSCA service template documents and interpreting the types contained. As of this writing, we are aware of Cloudify [14], Alien4cloud [15], Ubicity [16], SeaClouds [17] and OpenTosca [18]. However, in order to ensure that our TOSCA model can connect directly to our model converter and consequently be coupled closer with

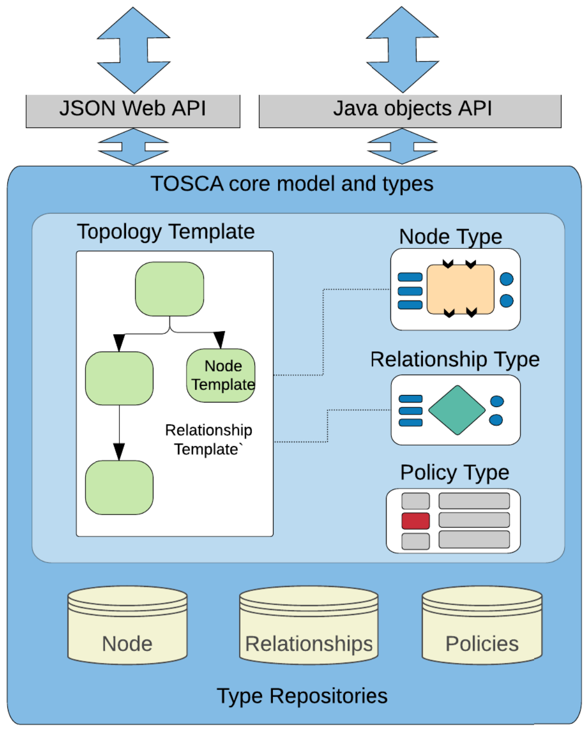


FIGURE 4. **TOSCA parser architecture.**

CAMP, we have developed an in-house TOSCA parser capable of parsing TOSCA normative types in YAML as well as user-defined types written as extensions to TOSCA normative types. Our in-house TOSCA parser is capable of parsing a TOSCA YAML document and storing the components as objects which can be later deployed to a provider via a compatible platform. The parser as seen in Fig. 4 consists of three main parts.

- *The parser core* contains the representations of the TOSCA normative types and is used to process the TOSCA service template.
- *A programming API model* allows for access to the parser core through Java objects.
- *Web API model* allows for access to the parser core through a Web API that consumes and produces JSON.

2) EXTENDED CAMP

Another major component is the extended CAMP platform. The OASIS CAMP standard is defined to allow the deployment and management of cloud applications, defined as YAML plans, onto cloud provider platforms. Applications are packaged as Platform Deployment Packages, otherwise known as PDPs, and delivered to a compliant CAMP platform. Upon arriving at the platform, the PDP is parsed into Artifacts, Requirements and Services, all of which are used to deploy the application to an appropriate cloud provider's platform. The specification not only defines the format for supplying an application to be deployed, but it also defines a method for managing the application as well.

To do so, CAMP makes use of an infrastructure composed of resources. Resources represent elements of the underlying system that can be interacted with through the CAMP protocol exposed by the platform.

Our extended CAMP platform was created, so that it can process policies that may be used to manage a deployed application or components. In a previous work, we demonstrated the use of policies for application orchestration across heterogeneous clouds [19]. With that in mind, we also made special considerations, when addressing the use of policies within our work. TOSCAs previous, XML-based specification made use of policies defined as work-flows in BPEL or BPMN [8]. The state of the art has since moved from the imperative specification of policies to a declarative specification. As TOSCAs policies are now defined declaratively, it is necessary for us to provide a declarative policy specification in CAMP that will allow for the interpretation of translated TOSCA policies. Our approach introduces declaratively defined, typed policies within our extended CAMP document. Our policy specification makes use of two main components, typed policies and typed constraints.

```
policies:
- type: kr.ac.hanyang.oCamp.entities.policies.Placement
  constraints:
  - property: SERVICE_UP
    type: kr.ac.hanyang.oCamp.entities.constraints.EqualTo
    value: true
  - property: PROVISIONING_LOCATION
    type: kr.ac.hanyang.oCamp.entities.constraints.Within
    value: [Rackspace Hong Kong (hkg), Rackspace Sydney (syd)]
  targets: [webcluster]
```

FIGURE 5. **Sample policy specified in our extended CAMP format.**

Declarative policies in our extended CAMP platform in Fig. 5 specify state or behavior that an entity should adhere to and do not imperatively specify actions that must be taken. Typed policies in our extension represent a directive that may be associated with an entity, but must be interpreted by a policy enforcement agent and not the entity itself. Typed constraints capture the state that an entity should ideally be in, if it is to conform to the policy. For example, a Placement policy may be associated with an entity stating that the entity should be started (e.g., `SERVICE_UP` = true) and its location should ideally be chosen from one of a supplied set of locations (e.g., `PROVISIONING_LOCATION` is within [loc 1, loc 2, loc n]). If this policy were to be enforced, it would mean that the target must be maintained in a started state in one of the defined locations. In the event that the target's state stops or fails for some reason, the through the policy their target should be attempted to be placed back into a started state.

As declaratively defined policies define criteria that an entity should adhere to, whether or not the entity adheres depends not on the policy but the unit used to enforce the policy [20]–[24]. To capture this concept, we introduced a component known as a Policy Manager into our extended CAMP platform. This component serves as a container for policies and must interpret and enforce those policies on an entity. In the case of a policy violation, the policy manager component should be aware of actions that can be taken on the entity in order to return that entity to a valid state. To further our work, it is necessary to be able to specify these actions declaratively as directives. The actions, in our approach, are thus taken by the policy manager in order to enforce a policy on a CAMP entity. With this approach, an application designer would be able to declaratively define an application to be deployed as well as define directives to policy managers that may influence how they manage the entities of the application.

```
type: kr.ac.hanyang.oCamp.entities.policies.Placement
  actiongroups:
  - id: START
    actions:
    - property: SERVICE_UP
      transitions:
      - type: Initial
        value: false
      - type: Set
        value: true
    - property: PROVISIONING_LOCATION
      transitions:
      - type: Initial
        value: null
      - type: Set
        value: ANYTHING
  - id: STOP
    actions:
    - property: SERVICE_UP
      transitions:
      - type: Initial
        value: true
      - type: Set
        value: false
    - property: PROVISIONING_LOCATION
      transitions:
      - type: Initial
        value: ANYTHING
      - type: Set
        value: null
```

FIGURE 6. **Sample policy management directive in our extended CAMP format.**

The policy management directives as seen in Fig. 6, are defined as a group of actions the policy manager can choose from, that are applicable to the type of entity that the policy is associated with. Each action identifies the property of the entity that this action influences and the transition the property will undergo in the event of the action being performed. For example, the policy manager may be given the placement directives that specify two possible courses of actions it may take, i.e., START or STOP. The START action operating on an entity results in its `SERVICE_UP` property being set to true from its initial value of false. The same action may also affect the entitys provisioned location by setting its `PROVISIONING_LOCATION` value to "any" location from its initial value of null. The initial value is null, because a START action can only be performed, if the entity is in a STOPPED stated. In a stopped state, the location of the entity would not be defined and would be null. As these actions represent known actions that can be invoked on entities within our extended platform, the policy manager needs only to decide on the best course of action to be performed and invoke that action on the entity. This decision is made by analyzing the transitions an entity will undergo in order to be returned to a valid state. The policy manager first gathers the actions that contain properties that will be affected. The transitions are analyzed to determine if their start and end states coincide with the desired state of the entity. Transitions are weighted by the policy manager, so that actions with the lowest transition weight are selected first. Once an action is selected, the entity should already be aware of how to carry out that action, since actions are represented by CAMP operations. Thus, imperative instructions are not passed to the entity.

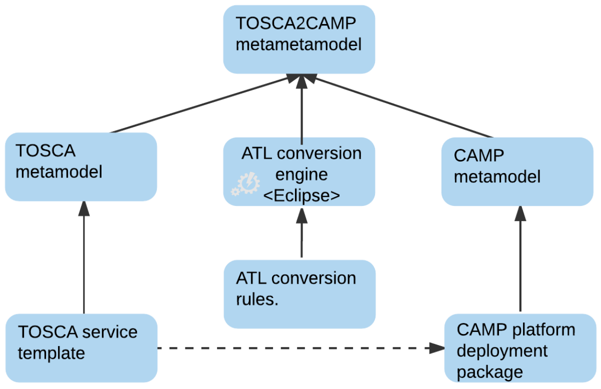


FIGURE 7. Model conversion via atlas transformation language.

B. MODEL CONVERSION STRATEGY

Our approach to combining TOSCA and CAMP involves the conversion of a TOSCA Service Template into a CAMP platform deployment package using ATL (Atlas Transformation Language) [25], [26]. The Atlas Transformation Language is a hybrid, imperative and declarative language that can allow the production of a number of target models from a set of source models through the use of model elements and rules. The model transformation language ATL works by using a meta-model of the source model and a meta-model of the destination model which must each in turn conform to a common meta-meta-model as depicted in Fig. 7.

To illustrate the model conversion process, we use a typical scenario depicting the deployment of an application using TOSCAMP, the details of which can also be seen through Fig. 3. The process begins with a YAML file representing the Service Template of the application to be deployed. The Service Template is first loaded into the parser where it is parsed into the relative components, for example, Node Templates, Relationship Templates, and Policy Definitions. Any user-defined types are also parsed and stored in repositories within the parser.

The parsed Service Template object and components are stored as Java classes within the parser. These classes can be directly accessed by the ATL conversion engine and serves as the TOSCA model depicted in Fig. 7. The meta-model of the TOSCA model is also stored within the TOSCA parser as an Eclipse ecore object. The model is used by the ATL conversion engine along with the CAMP meta-model which is stored in the extended CAMP platform as an ecore object.

The actual conversion from TOSCA to CAMP takes place in the ATL conversion engine using ATL conversion rules. To enable this process, we must deliver to the engine the service template model, the TOSCA meta-model, and the CAMP meta-model. The properties of the nodes and components of TOSCA and CAMP are stored in lookup tables and are directly accessed by the ATL rules, when converting one model to another. For example, the lookup table may have an entry for ‘‘textttroot_password’’ property of the ‘‘tosca.nodes.DBMS’’ node template that maps to ‘‘password’’ property of the ‘‘services.database.mysql.MySQL’’ in CAMP.

Once the conversion is complete, the ATL conversion creates Java objects that represent the Platform Deployment Plan of a CAMP application. This PDP object can then be directly deployed by the CAMP platform. The declarative policies introduced into our CAMP PDP are directly translated from the declarative policies of the TOSCA topology. However, to ensure our policies are enforced, we must side-load policy directives into the CAMP platform.

During the conversion process, there are also other considerations that must be made, for example, CAMP plans are typically ‘‘flat’’ documents of about one level in depth, whereas a typical TOSCA document may be a few service levels deep. It is not possible to directly convert such a structure into CAMP, as CAMP only captures the relationship between Artifacts and the Services that fulfill them and not Service to Service relationships. To mitigate this, it is necessary to devise an approach to flatten the structure of the TOSCA Topology prior to the conversion. Our method relies on identifying particular patterns in the TOSCA topology. For example, a TOSCA topology may comprise a LoadBalancer typed node that ‘‘RoutesTo’’ an application hosted on a WebServer typed node as seen in Fig. 8. Our approach identifies this pattern as a ‘‘member pattern’’ and compresses the pattern into a LoadBalancer with a WebServer as its member. In another example also seen in Fig. 8, if we consider a database node that is ‘‘HostedOn’’ a SQL DBMS typed node. This SQLDBMS typed node may, in turn, be ‘‘HostedOn’’ a Computer typed node with specific specifications. Our approach identifies this ‘‘service to service’’ pattern and compresses the services up to the highest service, while preserving the properties of lower services for translation. Fig. 8 gives a depiction of these strategies.

IV. IMPLEMENTATION DETAILS AND EVALUATION

In order to evaluate our proposal of the TOSCAMP architecture, we have taken a two-pronged approach. Firstly, we present a case study of the implementation of TOSCAMP as well as a look at the manner in which our solution handles the deployment and management of our sample application

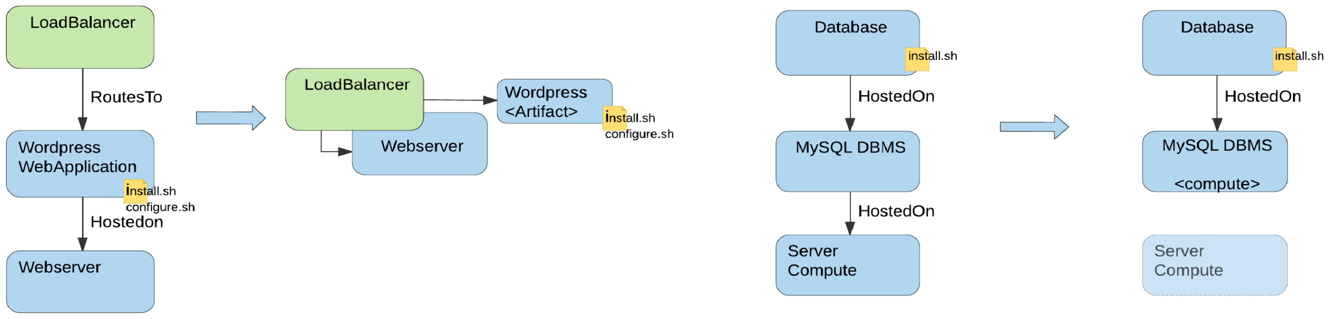
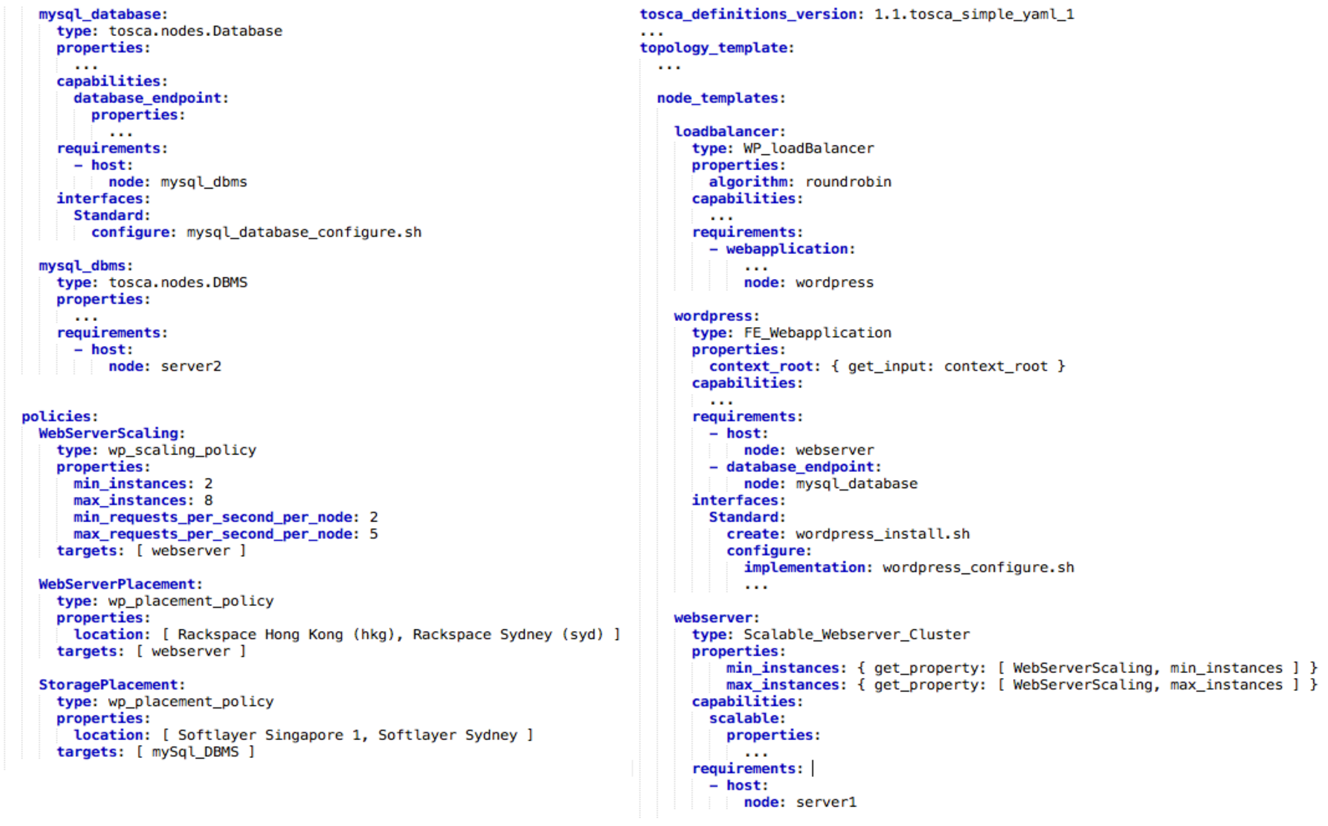


FIGURE 8. Member and service-service pattern conversion strategies.



```
  mysql_database:
    type: tosca.nodes.Database
    properties:
      ...
    capabilities:
      database_endpoint:
        properties:
          ...
    requirements:
      - host:
          node: mysql_dbms
    interfaces:
      Standard:
        configure: mysql_database_configure.sh

  mysql_dbms:
    type: tosca.nodes.DBMS
    properties:
      ...
    requirements:
      - host:
          node: server2


policies:
  WebServerScaling:
    type: wp_scaling_policy
    properties:
      min_instances: 2
      max_instances: 8
      min_requests_per_second_per_node: 2
      max_requests_per_second_per_node: 5
    targets: [ webserver ]

  WebServerPlacement:
    type: wp_placement_policy
    properties:
      location: [ Rackspace Hong Kong (hkg), Rackspace Sydney (syd) ]
    targets: [ webserver ]

  StoragePlacement:
    type: wp_placement_policy
    properties:
      location: [ Softlayer Singapore 1, Softlayer Sydney ]
    targets: [ mySql_DBMS ]
```

```
tosca_definitions_version: 1.1.tosca_simple_yaml_1
...
topology_template:
  ...

  node_templates:

    loadbalancer:
      type: WP_loadBalancer
      properties:
        algorithm: roundrobin
      capabilities:
        ...
      requirements:
        - webapplication:
            ...
            node: wordpress

    wordpress:
      type: FE_Webapplication
      properties:
        context_root: { get_input: context_root }
      capabilities:
        ...
      requirements:
        - host:
            node: webserver
        - database_endpoint:
            node: mysql_database
      interfaces:
        Standard:
          create: wordpress_install.sh
          configure:
            implementation: wordpress_configure.sh
            ...

    webserver:
      type: Scalable_Webserver_Cluster
      properties:
          min_instances: { get_property: [ WebServerScaling, min_instances ] }
          max_instances: { get_property: [ WebServerScaling, max_instances ] }
      capabilities:
        scalable:
          properties:
            ...
      requirements:
        - host:
            node: server1
```

FIGURE 9. TOSCA specification for Wordpress application.

across two cloud providers. Secondly, we performed a standalone evaluation of our solution to determine how it would handle different scaling situations, when varying the workload of the application. For our case study, we have chosen to use WordPress as the application, which is a popular, scalable Web blogging and publishing platform. Our WordPress topology consists of a load balancer which routes requests to WordPress front-end Web servers. Each Web server connects to a MySQL database on a MySQL back-end DBMS. The architecture is designed, so that the front-end servers can be scaled up or down depending on the workload being experienced. The case study analyses how our solution handles difference scenarios.

- Placement of application components
- Decision making with regards to scaling

The TOSCA service template used to represent the application is given in Fig. 9 and corresponds to the application topology given in Fig. 1.

A. MODEL CONVERSION

Our approach emphasizes application portability and interoperability by adhering to the standards TOSCA and CAMP. By maintaining two separate models, each model can be tailored to perform best at what they were designed to do. This adheres to the basics of MDE (Model-Driven

Engineering) [27] of which the main objective is to "factorize the complexity into different levels of abstraction and concern from high level, conceptual models down to individual aspects of the target platform."

One of the core tenets of MDE is the separation into Platform-Independent and Platform-Specific Model, PIM and PSM, respectively. PIM represents a model that is independent of the platform that is used to implement it. A PIM would thus represent a truly detached and portable model, abstracted from the platform-specific details of its implementation [27]. A PSM however is related in some form to the platform implementing it. Hence, the functionality that is described in a PIM should be realized through a PSM [27].

Models, if interrogated, should be able to provide up-to-date information about the solution they represent and thus drive adaptation decisions. Our approach maintains two independent models: a TOSCA model which represent the topology of the application independent of any providers or provider-specific resources and a CAMP model which is a provider-specific model that incorporates the types capable of deployment by specific providers. The TOSCA plan is converted using our ATL model conversion rules into a CAMP platform deployment plan representation of the WordPress application. The original TOSCA plan remains unchanged and there is no information injected into this model to facilitate the conversion. By keeping the model unmodified, the original TOSCA plan can be used in other TOSCA platforms, Hence, the model remains portable and reusable as provider-specification information is omitted from this model.

To govern placement of the application components, TOSCA placement policies are to be created and embedded into the application's service template. Our placement policies definition in TOSCA are also typed components that associate the property to be managed, in this case, the location of the application component, and the target node the policy should be associated with. This approach allows for policies to be defined separate of the components they will govern. It also means that the application topology remains portable as specific-information related to the policy is never embedded into the component but associated, once the policy is processed. TOSCA orchestrators capable of processing policies will process and enforce them. However, orchestrators that are unable to process policies will not be affected by their presence, as the policies are not embedded into the nodes.

The conversion process makes use of an Atlas Transformation Language (ATL) conversion engine to transform a model of the TOSCA application's service template into a CAMP deployment plan. The conversion was done by creating and maintaining a look-up table of TOSCA nodes and relationships as well as CAMP components. During the conversion, properties registered to TOSCA types are converted to CAMP components. Fig. 10 presents the high-level look at our model conversion algorithm and, in Fig. 11, we can see the converted CAMP deployment plan.

To evaluate this algorithm, we performed a series of conversions of different TOSCA plans into CAMP. First, we ran

```
for each item in ServiceTemplate
    if isArtifact(item.getType()){
        convertToArtifact(item);
    }else
    if isService(item.getType()){
        convertToService(item);
    }else
    if isPolicy(item.getType()){
        convertToPolicy(item);
    }else{
        /**
        * The item's type could not be classified
        */
    }
}

converToArtifact(item){
    return campArtifact(
        artifactType = convertNodeType(item.type);
        content = getContent(item);
        requirements = getRequirements(item.Requirements);
    );
}

convertToService(item){
    return campService(
        characteristicType = convertServiceCharacteristicType(item.type);
        customAttributes = getProperties(item);
    );
}

convertToPolicy(item){
    return extendedCampPolicy(
        policyType = convertPolicyType(item.type);
        policyConstraints = getConstraints(item);
        targets = getTargets(item.targets);
    );
}
```

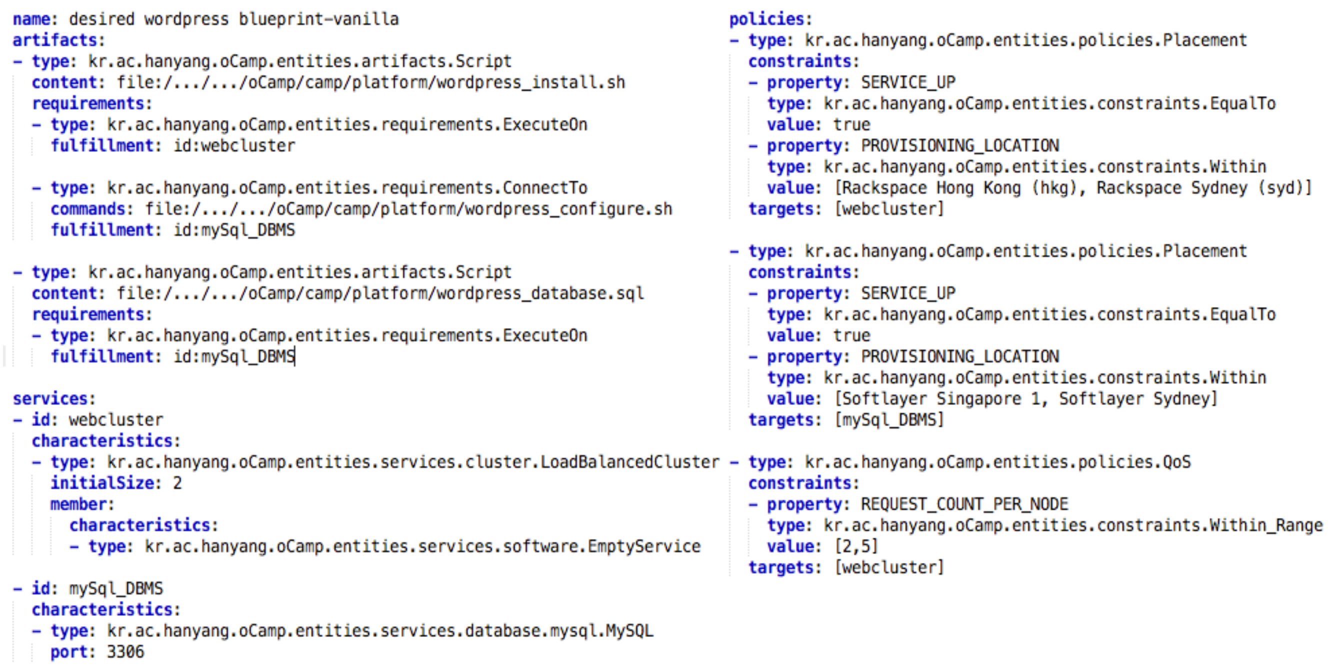
FIGURE 10. Model conversion algorithm.

a baseline conversion of a TOSCA document containing only a WordPress Front-End (WP_FE) which refers to a WordPress webapplication TOSCA node "HostedOn" a webserver node. Using the values of ten conversions runs, the average time to convert a single application stack was found.

After our baseline was established, we then ran a series of conversions by increasing the number of WordPress Front-Ends. We then channeled this data into a graph of conversion time against conversion run as can be seen in Fig. 12. By looking at the data plotted, we observed that the conversion times for each run remained consistently small. The times for each conversion, although it fluctuated, did not change drastically but instead remained between 0.12s and 0.18s.

B. MULTICLOUD DEPLOYMENT AND SCALING

The placement of application components is handled in our approach by using placement policies. Placement policies identify the property and constraint to be enforced as well as the targets they should be enforced on. To test the placement of components using placement policies, we devised three scenarios in which we used the application topology specified in Fig. 1. To deploy the application to various cloud providers. In scenario 1, a single policy is used to place all components on the same provider. This scenario is used as our baseline and is run for each provider. The time taken to deploy the topology to each provider was recorded and can be seen in Fig. 13. In scenario 2, we separated the components by using two policies to place separate components on separate providers. In this scenario, we tested the separation of services by using



```
name: desired wordpress blueprint-vanilla
artifacts:
- type: kr.ac.hanyang.oCamp.entities.artifacts.Script
  content: file:/.../.../oCamp/camp/platform/wordpress_install.sh
  requirements:
  - type: kr.ac.hanyang.oCamp.entities.requirements.ExecuteOn
    fulfillment: id:webcluster

  - type: kr.ac.hanyang.oCamp.entities.requirements.ConnectTo
    commands: file:/.../.../oCamp/camp/platform/wordpress_configure.sh
    fulfillment: id:mySql_DBMS

- type: kr.ac.hanyang.oCamp.entities.artifacts.Script
  content: file:/.../.../oCamp/camp/platform/wordpress_database.sql
  requirements:
  - type: kr.ac.hanyang.oCamp.entities.requirements.ExecuteOn
    fulfillment: id:mySql_DBMS

services:
- id: webcluster
  characteristics:
  - type: kr.ac.hanyang.oCamp.entities.services.cluster.LoadBalancedCluster
    initialSize: 2
    member:
      characteristics:
      - type: kr.ac.hanyang.oCamp.entities.services.software.EmptyService

- id: mySql_DBMS
  characteristics:
  - type: kr.ac.hanyang.oCamp.entities.services.database.mysql.MySQL
    port: 3306
```

```
policies:
- type: kr.ac.hanyang.oCamp.entities.policies.Placement
  constraints:
  - property: SERVICE_UP
    type: kr.ac.hanyang.oCamp.entities.constraints.EqualTo
    value: true
  - property: PROVISIONING_LOCATION
    type: kr.ac.hanyang.oCamp.entities.constraints.Within
    value: [Rackspace Hong Kong (hkg), Rackspace Sydney (syd)]
  targets: [webcluster]

- type: kr.ac.hanyang.oCamp.entities.policies.Placement
  constraints:
  - property: SERVICE_UP
    type: kr.ac.hanyang.oCamp.entities.constraints.EqualTo
    value: true
  - property: PROVISIONING_LOCATION
    type: kr.ac.hanyang.oCamp.entities.constraints.Within
    value: [Softlayer Singapore 1, Softlayer Sydney]
  targets: [mySql_DBMS]

- type: kr.ac.hanyang.oCamp.entities.policies.QoS
  constraints:
  - property: REQUEST_COUNT_PER_NODE
    type: kr.ac.hanyang.oCamp.entities.constraints.Within_Range
    value: [2,5]
  targets: [webcluster]
```

FIGURE 11. Converted CAMP Wordpress PDP.

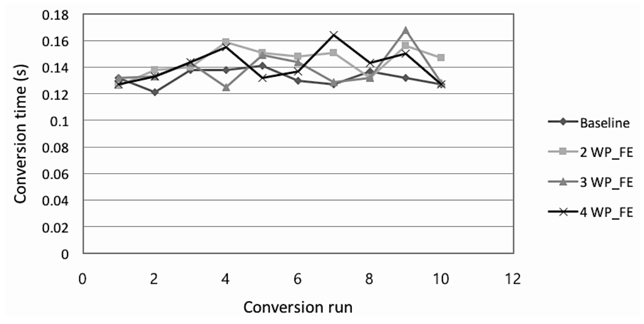


FIGURE 12. Conversion times of TOSCA document to CAMP plan.

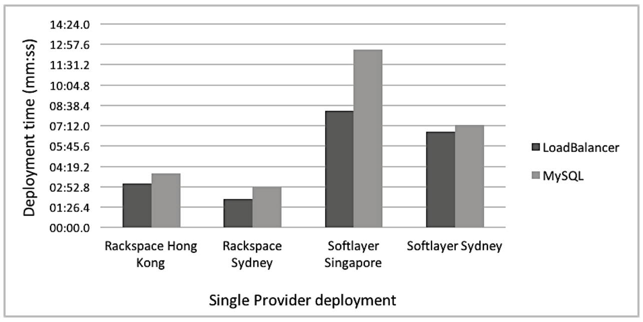


FIGURE 13. Deployment times to a single provider.

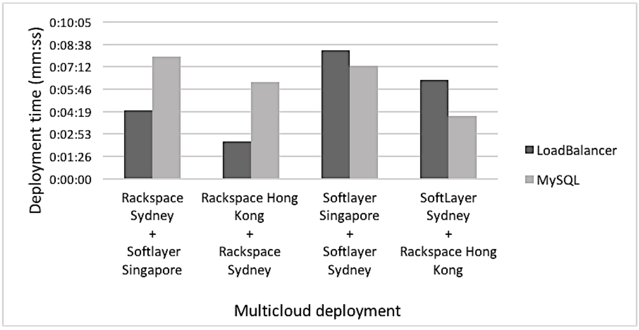


FIGURE 14. Deployment times over multiple providers.

policies to place each service on a different provider. The times taken to deploy each service on their particular provider as well as connect the complete application was recorded. In scenario 3, we used a single policy to place all component across multiple providers. This scenario tested whether it was possible for a single policy to be used to distribute the components of an application across multiple heterogeneous providers. We choose to conduct these experiments using live cloud services: Rackspace and IBM Softlayer cloud services. For each provider, we chose two locations. It was possible to have chosen more locations. However, in an effort to keep the tests as simple as possible, two locations were decided to be appropriate.

The placement policy used for scenario 3 simply chooses randomly from the supplied providers (without affinity) and deploys the component to the provider selected. In order to obtain sufficient data, we therefore needed to run as many deployments as possible. Therefore, during the experiment, we ran the deployments and selected the combinations that were chosen 3 or more times. Using these combinations, we then ran the experiments of scenario 2, obtaining a further set of data. We found the average deployment times of these combinations and presented the information in Fig. 14. Through the results of Fig. 13 and Fig. 14, we can deduce that our strategy is capable of deploying and connecting application components on the same provider as well as across heterogeneous providers. Differences in deployment time may be attributed to different provider APIs. However, this analysis may require further research.

In order to observe how our approach would handle scaling of application components, we performed experiments using a scaling policy to govern how the application components will scale based on varying workloads. These experiments also made use of the application topology define in Fig. 1. The scaling policy, as with placement policies, were defined within the TOSCA topology and then converted to a CAMP equivalent. After translating the topology, we obtained a scaling policy in CAMP which was deployed along with the application via our CAMP platform. For this deployment, a multi-cloud approach was also chosen. Hence, each component of the application was deployed on a separate provider. The providers used were kept as Rackspace and IBM Softlayer.

Our approach to policies requires the policy manager to be equipped with the proper directives in order to process the policy constraints and choose an appropriate action in case of a violated constraint. We created and loaded a declarative policy manager directive as can be seen in Fig 15, into out platform prior to loading our application topology.

```
type: kr.ac.hanyang.oCamp.entities.policies.QoS
actiongroups:
- id: SCALE_UP
  actions:
  - property: REQUESTS_PER_SECOND_PER_NODE
    transitions:
    - type: Initial
      value: ANYTHING
    - type: Decrease
      value: ANYTHING
  - property: NUM_CLUSTER_MEMBERS
    transitions:
    - type: Initial
      value: ANYTHING
    - type: Increase
      value: ANYTHING
- id: SCALE_DOWN
  actions:
  - property: REQUESTS_PER_SECOND_PER_NODE
    transitions:
    - type: Initial
      value: ANYTHING
    - type: Increase
      value: ANYTHING
  - property: NUM_CLUSTER_MEMBERS
    transitions:
    - type: Initial
      value: ANYTHING
    - type: Decrease
      value: ANYTHING
```

FIGURE 15. Scaling policy directive.

To initiate scaling, it was necessary to manipulate the workload of the WordPress front-end servers. To do this, we made use of the Apache JMeter (http://jmeter.apache.org) to generate a workload for the deployed applications in order to trigger a scaling scenario. Our application scaling rules were configured, so that the application would scale up the number of nodes in the WordPress front-end cluster, if the load, i.e., the requests per second of the servers within the cluster, is above a defined threshold.

Once the scaling policy and policy manager directives were deployed and enabled, our platform began responding to changes in the `REQUEST_PER_SECOND_PER_NODE` sensor which forms part of the component deployed via our extended CAMP. Once a violation was detected, the policy manager determined an appropriate action based on its directives and initiated that action. In our case, the policy manager decided that the application front-end should be scaled up. The scale-up process took place in stages, as the policy manager conservatively scaled and then checked that the violation was alleviated. A small "hold of" period was used to ensure that the policy manager does not trigger multiple actions in quick succession.

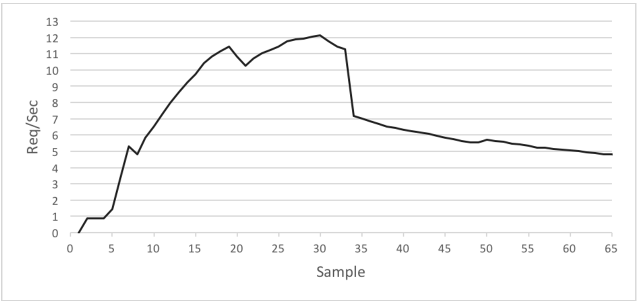


FIGURE 16. Requests per second per node measured at the load balancer.

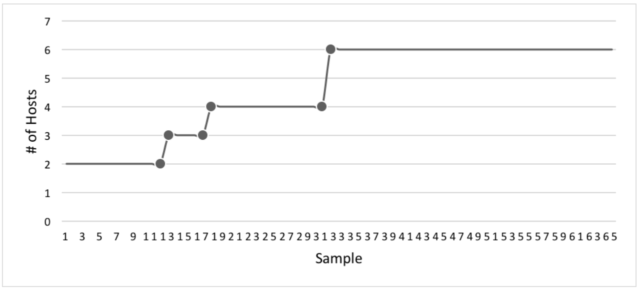


FIGURE 17. Scale up of nodes configured at the load balancer.

The results of our scaling experiment are shown in Fig. 16 and Fig 17. The experiment was run for 360 seconds during which the number of requests to the Web servers was gradually increased. Samples were taken of the average requests per second on each node via the LoadBalancer at 5 second intervals. As the number of requests increases, the requests per second measured at the server increase as well, until the value of 5 requests per second is crossed. At this point, the number of nodes is scaled up to 3 since this is detected as a policy violation. This scale up, however, fails to address the constraint violation and as a result another scale up is triggered at sample 17. As the constraint remains violated even after each consecutive scale up, another scale up is finally triggered at sample 31, bringing the number of nodes to 6. This scale up is enough to drop the constraint so that it is no longer in violation.

V. RELATED WORK

Cloud application orchestration has generated a considerable amount of popularity within the field of distributed computing. There have been numerous approaches to cloud application orchestration, each with varying advantages and limitations [7], [28], [29]. Some approaches focus on creation and use of standards, others on the use of libraries and intermediary layers, while others are based on the use of semantics of models. The authors of [28] also described a taxonomy that was used to compare cloud application orchestration techniques on two broad sets of criteria. From this taxonomy, cloud orchestration techniques were judged based on two broad areas.

- Cloud Feature: features specific to the cloud infrastructure.
- Application Feature: features specific to the application supported in the cloud.

To gain an appreciation for how our approach stacks up against other similar approaches, we produced a qualitative comparison of the techniques. Table 1 depicts the comparison of TOSCAMP, SeaClouds, MODAClouds, Brooklyn-TOSCA, and Cloud Provider Orchestration features.

Brooklyn-TOSCA [30] combined the facilities of TOSCA with the Apache Brooklyn platform. The SeaClouds project [17] is also known for its approach to multi-cloud deployment and management using a TOSCA DSL coupled with various deployment frameworks such as Apache Brooklyn and MODAClouds [31]. While these projects adequately combined the frameworks for orchestration and deployment and provide multi-cloud and cross-cloud support, there are some differences with regards to our TOSCAMP architecture. For example, Brooklyn-TOSCA makes use of an agnostic graph or intermediate graph that is used to bridge the gap between the TOSCA and Apache Brooklyn, and allows for resource selection and runtime adaptation. TOSCAMP approaches these through the use of placement policies which abstract the provider location from the application model. The SeaClouds approach also integrates TOSCA compliant plans and CAMP through the use of Apache Brooklyn. Our approach, however, does not rely on an intermediate graph in order to translate nor does it utilize Apache Brooklyn's CAMP DSL. Instead, it makes use of matchmaking, SLA and monitoring components coupled with continuous refinement to provide resource selection, lifecycle management, monitoring, and runtime adaptation. For these, our TOSCAMP approach makes use of the CAMP platform features as well as our policy extensions for CAMP.

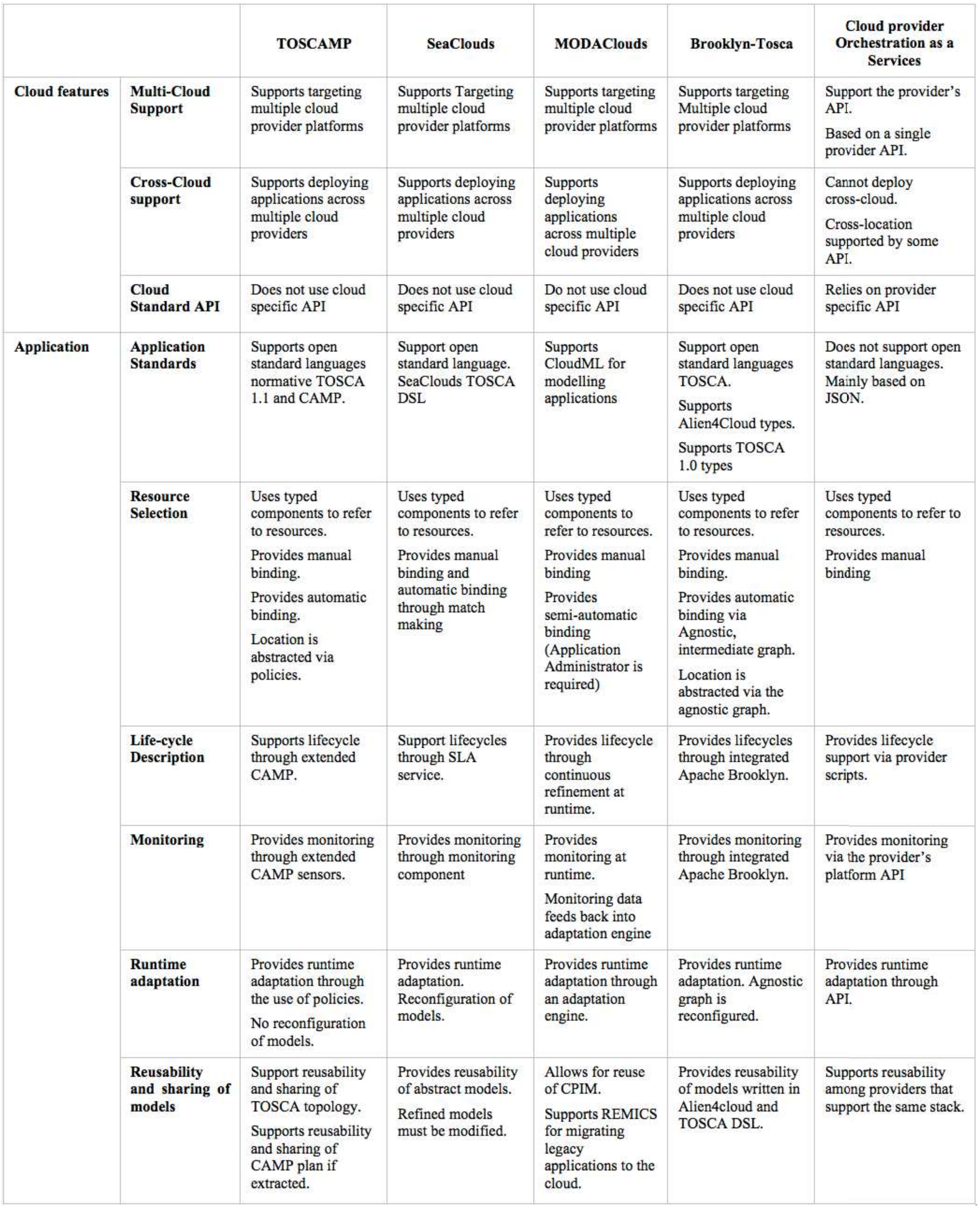
The concept of combining TOSCA and CAMP is not only limited to the SeaCloud and Brooklyn-TOSCA approaches. However, there are other studies that suggested the combination of the TOSCA and CAMP standards in order to provide cloud orchestration features [32], [33]. The authors of [32] proposed a strategy involving the use of ATL to convert from a TOSCA topology into an Apache Brooklyn plan in order to be deployed to a provider platforms. The proposal also suggests the use of an agnostic model to express the TOSCA model in order to generate orchestration plans for the application deployment and management. Similarly, the authors of [33] also proposed the combination of TOSCA and CAMP via Apache Brooklyn and suggested the use of an agnostic graph to bridge to automate the conversion process from TOSCA models to CAMP models. Their approach focused on abstracting all provider-specific information from TOSCA to provide a cross-cloud strategy that provides portability. Our proposal, however, incorporates an extended policy support system in order to provide cross- and multi-cloud support.

With regards to cloud application modeling, the CloudMF [34] and MODAClouds [31] approaches provide a model-driven approach to multi-cloud application deployment through the use of a Domain Specific Language. The CloudMF approach introduced a cloud modeling framework consisting of a modeling language CloudMF and a deployment and management component described as models@run-time. This approach like ours took the two-model approach of Model Driven Engineering. This approach, however, relies on comparison and reasoning engines that must make modifications to the run-time model, when changes occur in either the Cloud Provider Independent Model or the models@run-time. While there are similarities to our approach, this still differs, as it makes use of refinement to transform the Cloud Provider Independent Model into the Cloud Provider Specific Model (CPSM). Also, it does not make use of policies to manage the deployed model, but instead relies on a reasoning engine to determine difference in the actual and target CPSM. More so, CloudMF is a model-based approach not based on open standards. Hence, its DSL does not adopt open techniques such as TOSCA and CAMP.

The MODAClouds approach, which is a reference implementation of CloudMF, highlighted the use of models for both the deployment of the system and for monitoring the run-time environment and considered two general phases of the application lifecycle: design time and run-time. Our TOSCAMP approach does not perform continuous refinement of an intermediate model. With continuous refinement, the model is continually enriched with provider-specific information, as it is translated into the provider-specific model. By not performing continuous refinement, our model does not need to be reevaluated, when provider information changes or there is a need to target another provider.

Almost every approach relies on an interface to each cloud platform. The Apache Brooklyn project represents a platform most compatible with the tenets of OASIS CAMP [10]. Being built on the concepts of CAMP, the project provides a blueprint document that is somewhat compliant to the CAMP standard. While our project makes use of some of Brooklyn's core components, the Brooklyn blueprint only represents a subset of the CAMP plan document. In its current form,

TABLE 1. Qualitative analysis of orchestration approaches.



		TOSCAMP	**SeaClouds**	**MODAClouds**	**Brooklyn-Tosca**	**Cloud provider Orchestration as a Services**
Cloud features	**Multi-Cloud Support**	Supports targeting multiple cloud provider platforms	Supports Targeting multiple cloud provider platforms	Supports targeting multiple cloud provider platforms	Supports targeting Multiple cloud provider platforms	Support the provider's API. Based on a single provider API.
	Cross-Cloud support	Supports deploying applications across multiple cloud providers	Supports deploying applications across multiple cloud providers	Supports deploying applications across multiple cloud providers	Supports deploying applications across multiple cloud providers	Cannot deploy cross-cloud. Cross-location supported by some API.
	Cloud Standard API	Does not use cloud specific API	Does not use cloud specific API	Do not use cloud specific API	Does not use cloud specific API	Relies on provider specific API
Application	**Application Standards**	Supports open standard languages normative TOSCA 1.1 and CAMP.	Support open standard language. SeaClouds TOSCA DSL	Supports CloudML for modelling applications	Support open standard languages TOSCA. Supports Alien4Cloud types. Supports TOSCA 1.0 types	Does not support open standard languages. Mainly based on JSON.
	Resource Selection	Uses typed components to refer to resources. Provides manual binding. Provides automatic binding. Location is abstracted via policies.	Uses typed components to refer to resources. Provides manual binding and automatic binding through match making	Uses typed components to refer to resources. Provides manual binding Provides semi-automatic binding (Application Administrator is required)	Uses typed components to refer to resources. Provides manual binding. Provides automatic binding via Agnostic, intermediate graph. Location is abstracted via the agnostic graph.	Uses typed components to refer to resources. Provides manual binding
	Life-cycle Description	Supports lifecycle through extended CAMP.	Support lifecycles through SLA service.	Provides lifecycle through continuous refinement at runtime.	Provides lifecycles through integrated Apache Brooklyn.	Provides lifecycle support via provider scripts.
	Monitoring	Provides monitoring through extended CAMP sensors.	Provides monitoring through monitoring component	Provides monitoring at runtime. Monitoring data feeds back into adaptation engine	Provides monitoring through integrated Apache Brooklyn.	Provides monitoring via the provider's platform API
	Runtime adaptation	Provides runtime adaptation through the use of policies. No reconfiguration of models.	Provides runtime adaptation. Reconfiguration of models.	Provides runtime adaptation through an adaptation engine.	Provides runtime adaptation. Agnostic graph is reconfigured.	Provides runtime adaptation through API.
	Reusability and sharing of models	Support reusability and sharing of TOSCA topology. Supports reusability and sharing of CAMP plan if extracted.	Provides reusability of abstract models. Refined models must be modified.	Allows for reuse of CPIM. Supports REMICS for migrating legacy applications to the cloud.	Provides reusability of models written in Alien4cloud and TOSCA DSL.	Supports reusability among providers that support the same stack.

the Brooklyn blueprint diverges from core CAMP in the way policies are specified and processed. Apache Brooklyn is capable of handling and specifying policies declaratively within plans. In our approach, however, we decided to also explore the ability to define and tailor new policies and policy processing logic by providing the ability to define the policy processing components via directives.

Cloud providers rely on an array of services within their platforms in order to elevate their platform above others. There have been a number of cloud service providers providing orchestration services as part of their catalog of services. ''Orchestration as a Service'' solutions such as Amazon OpsWorks (https://aws.amazon.com/opsworks/), Amazon Cloud Formation (https://aws.amazon.com/cloudformation/), and Rackspace Cloud Orchestration (https://www.rackspace.com/cloud/orchestration) provide provisioning and scaling features for customers of these respective platforms through the use of DevOps recipes and/or Blueprints. While services such as these provide cloud orchestration features, by their nature, they still lock the client into a particular cloud provider. The DSL used by one provider may not necessarily be compatible with that of another provider. Furthermore, the orchestration actions performed on one provider cannot allow for cross-provider orchestration. Our work is set apart from others, as it provides a standards-based, complete orchestration solution for heterogeneous cloud platforms. Hence, a means of cross-provider orchestration that should keep users from being locked-in to any particular vendor.

VI. CONCLUDING REMARKS

Our work has explored a methodology for defining, deploying, and managing distributed cloud applications through the combination of two prominent standards: TOSCA and CAMP. We have also been successful in demonstrating that, through extensions to the current CAMP standard, it is possible to define declarative policies that can be used to orchestrate the components of a deployed application over heterogeneous cloud platforms.

We have also proposed, implemented, and tested a method of translating TOSCA documents into CAMP plans, while maintaining the separation of each model. We consider it to be beneficial to maintain this separation in order to promote model interoperability and mobility by ensuring that each model remains ''pure'' and not deviated from the standard. We would like to emphasize that our work was made possible through some proposed extensions to the CAMP platform. These extensions afford the platform the ability to process orchestration policies. It should also be noted that our approach to policies does not alter the established components of the standard but adds new components capable of performing the required orchestration tasks on the existing components. We view this as significant, since, with our approach, there will be no effect to platforms that are incapable of interpreting our policies.

There is still a significant amount of work that remains with regards to our approach and our prototype platform. Currently, our platform is capable of parsing a TOSCA version 1.1 document and converting the components into a CAMP document. The CAMP half of our TOSCAMP architecture, however, does not have a complete library of types that it is capable of deploying, and we plan to increase the number of types that can be handled by the CAMP portion. We also intend to explore the possibility of adding new types of policies and policy managers to our CAMP platform.

REFERENCES

[1] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective," *J. Cloud Comput., Adv., Syst. Appl.*, vol. 5, no. 1, p. 4, 2016.

[2] M. Armbrust *et al.*, "Above the clouds: A Berkeley view of cloud computing," Univ. California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009-28, Feb. 2009.

[3] C. Liu, B. T. Loo, and Y. Mao, "Declarative automated cloud resource orchestration," in *Proc. 2nd Symp. Cloud Comput.*, Cascais, Portugal, 2011, pp. 1–8.

[4] 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.

[5] A. Barros, M. Dumas, and P. Oaks, "Standards for Web service choreography and orchestration: Status and perspectives," *Lecture Notes Comput. Sci.*, vol. 3812, pp. 61–74, 2006.

[6] M. Hogan, F. Liu, A. Sokol, and J. Tong, "NIST cloud computing standards roadmap," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep., Special Pub. NIST-SP-500-291, Jul. 2011.

[7] D. Petcu and A. V. Vasilakos, "Portability in clouds: Approaches and research opportunities," *Scalable Comput.*, vol. 15, no. 3, pp. 251–271, 2014.

[8] *Topology and Orchestration Specification for Cloud Applications Version 1.0*, OASIS Standard TOSCA-v1.0, 2013.

[9] *TOSCA Simple Profile in YAML Version 1.0*, OASIS Standard TOSCA-Simple-Profile-YAML-v1.0, 2016.

[10] *Cloud Application Management for Platforms Version 1.1, OASIS CAMP TC Committee Specification 01*, 2014.

[11] D. Breitgand, A. Marashini, and J. Tordsson, "Policy-driven service placement optimization in federated clouds," IBM Res. Division, Haifa, Israel, Tech. Rep. H-0299, Feb. 2011, pp. 11–15.

[12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and M. Wieland, "Policy-aware provisioning of cloud applications," in *Proc. 7th Int. Conf. Emerg. Secur. Inf., Syst. Technol.*, Barcelona, Spain, 2013, pp. 86–95.

[13] A.-F. Antonescu, P. Robinson, and T. Braun, "Dynamic topology orchestration for distributed cloud-based applications," in *Proc. 2nd Symp. Netw. Cloud Comput. Appl.*, London, U.K., 2012, pp. 116–123.

[14] *Pure-Play Cloud Orchestration & Automation Based on TOSCA|Cloudify*. Accessed on Mar. 24, 2017. [Online]. Available: http://getcloudify.org

[15] *ALIEN 4 Cloud*. Accessed on Mar. 24, 2017. [Online]. Available: http://alien4cloud.github.io

[16] *UbicityTosca Tools and Orchestration*. Accessed on Mar. 24, 2017. [Online]. Available: https://ubicity.com

[17] A. Brogi *et al.*, "Adaptive management of applications across multiple clouds: The SeaClouds approach," *CLEI Electron. J.*, vol. 18, no. 1, pp. 1–14, 2015.

[18] T. Binz *et al.*, "OpenTOSCA—A runtime for TOSCA-based cloud applications," in *Proc. 11th Int. Conf. Service-Oriented Comput.*, Berlin, Germany, 2013, pp. 692–695.

[19] *Home—Apache Brooklyn*. Accessed on Apr. 29, 2016. [Online]. Available: https://brooklyn.apache.org/

[20] K. Alexander, C. Lee, and S. Chai, "Declarative policy support for cloud application orchestration," in *Proc. 19th Int. Conf. Adv. Commun. Technol.*, Bongpyeong, South Korea, 2017, pp. 102–104.

[21] L. Kagal, T. Finin, and A. Joshi, "Declarative policies for describing Web service capabilities and constraints," in *Proc. W3C Workshop Constraints Capabilities Web Services*, Redwood City, CA, USA, 2004, pp. 1–5.

[22] 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.
[23] N. C. Damianou, "A policy framework for management of distributed systems," Ph.D. dissertation, Dept. Comput., Univ. London, London, U.K., 2002.
[24] M. Sloman, "Policy driven management for distributed systems," *J. Netw. Syst. Manage.*, vol. 2, no. 4, pp. 333–360, 1994.
[25] ATLAS Group, INRIA and University of Nantes. (Feb. 2006). *ATL: Atlas Transformation Language—ATL User Manual Version 0.7*. [Online]. Available: http://www.eclipse.org/atl/documentation/old/ATL_User_Manual[v0.7].pdf
[26] F. Jouault and I. Kurtev, "Transforming models with ATL," *Lecture Notes Comput. Sci.*, vol. 3844, pp. 128–138, 2006.
[27] S. Kent, "Model driven engineering," in *Proc. 3rd Int. Conf. Integr. Formal Methods*, Turku, Finland, 2006, pp. 286–298.
[28] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and J. Domaschka, "Cloud orchestration features: Are tools fit for purpose?" in *Proc. 8th Int. Conf. Utility Cloud Comput.*, Limassol, Cyprus, 2015, pp. 95–101.
[29] K. Bousselmi, Z. Brahmi, and M. M. Gammoudi, "Cloud services orchestration: A comparative study of existing approaches," in *Proc. 28th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, Victoria, BC, Canada, 2014, pp. 410–416.
[30] J. Carrasco, J. Cubo, F. Durán, and E. Pimentel, "Bidimensional cross-cloud management with TOSCA and Brooklyn," in *Proc. 9th Int. Conf. Cloud Comput.*, San Francisco, CA, USA, 2016, pp. 951–955.
[31] D. Ardagna *et al.*, "MODAClouds: A model-driven approach for the design and execution of applications on multiple clouds," in *Proc. 4th Int. Workshop Modeling Softw. Eng.*, Zürich, Switzerland, 2012, pp. 50–56.
[32] J. Carrasco, J. Cubo, and E. Pimentel, "Towards a flexible deployment of multi-cloud applications based on TOSCA and CAMP," *Commun. Comput. Inform. Sci.*, vol. 508, pp. 278–286, 2015.
[33] J. Carrasco, J. Cubo, E. Pimentel, and F. Durán, "Deployment over heterogeneous clouds with TOSCA and CAMP," in *Proc. 6th Int. Conf. Cloud Comput. Services Sci.*, Rome, Italy, 2016, pp. 170–177.
[34] N. Ferry, H. Song, A. Rossini, F. Chauvel, and A. Solberg, "CloudMF: Applying MDE to tame the complexity of managing multi-cloud applications," in *Proc. 7th Int. Conf. Utility Cloud Comput.*, London, U.K., 2014, pp. 269–277.

KENA ALEXANDER received the B.S. degree in computer science from The University of the West Indies, St. Augustine, Trinidad and Tobago, in 2005. He is currently pursuing the M.S. degree at Hanyang University, Seoul, South Korea. His research areas are cloud application orchestration and inter-cloud portability and management.

CHOONHWA LEE received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1990 and 1992, respectively, and the Ph.D. degree in computer engineering from the University of Florida, Gainesville, FL, USA, in 2003. He is currently a Professor with the Division of Computer Science and Engineering, Hanyang University, Seoul. His research interests include cloud computing, peer-to-peer and mobile networking and computing, and services computing technology.

EUNSAM KIM received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1994 and 1996, respectively, and the Ph.D. degree in computer and information science and engineering from the University of Florida, Gainesville, FL, USA, in 2006. He was with the Digital TV Research Laboratory, LG Electronics, South Korea, from 1996 to 2002. He is currently an Associate Professor with the Department of Computer Engineering, Hongik University, Seoul. His research interests include distributed computing, P2P streaming, cloud computing, mobile computing, and network storage systems.

SUMI HELAL received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, USA. He is currently a Professor with the Computer and Information Science and Engineering Department, University of Florida, Gainesville, FL, USA, and the Director of the Mobile and Pervasive Computing Laboratory. He also directs the Gator Tech Smart House, an experimental facility for developing and validating assistive technology in support of aging, disability, and independence. His research interests include pervasive and mobile computing, smart health and well being, and cloud-sensor systems.