Automating Cloud Service Deployment and Management using Model-driven Techniques

Anirban Bhattacharjee, Yogesh Barve, Aniruddha Gokhale
Vanderbilt University
Nashville, Tennessee, USA 37235
anirban.bhattacharjee,yogesh.d.barve,a.gokhale@vanderbilt.edu

Takayuki Kuroda
NEC Corporation
Kawasaki, Kanagawa, Japan 211-8666
t-kuroda@ax.jp.nec.com

ABSTRACT

Deployment and orchestration of services on cloud platforms is a labor-intensive and error-prone process because of high variabilities incurred in the configuration of the virtualized environment and meeting the software dependencies for each service deployed in these environments. Although many cloud automation and orchestration tools are available for deployment and management of composite cloud services, users are often required to specify low-level scripting details for service deployment and management. Using low-level scripting capabilities incurs a steep learning curve. With each tool adopting its own set of APIs and scripting languages, users are often locked into a specific technology making it hard to interoperate, deploy and manage the services across heterogeneous cloud platforms. To address these challenges and provide a technology- and platform-agnostic solution to cloud service deployment, we present a cloud automation and orchestration framework called CloudCAMP. CloudCAMP incorporates domain-specific modeling so that the specifications and dependencies of clouds and applications architecture are specified at an intuitive, higher level of abstraction without the need for defining all the low-level domain details. The extensible and reusable knowledge base maintained by CloudCAMP helps to complete the partial specifications and generate an entirely deployable Infrastructure-as-Code (IAC), which can be handled by the existing tools to deploy, manage and provision the services components automatically. We validate our approach quantitatively by showing a comparative study of savings in manual effort while using CloudCAMP.

KEYWORDS

cloud services, deployment and orchestration, automation, domain-specific modeling, knowledge base.

1 INTRODUCTION

Self-service application deployment, orchestration, and management are desired for enterprises to speed up time-to-market for their services while ensuring their reliable deployment, particularly when the services are hosted in the cloud environment. Presently, however, enterprises often tend to experience service outages and delays that stem predominantly from the use of manual efforts expended in service configuration, release integration and dealing with platform heterogeneity, which are often error-prone, tedious and slow [6, 9]. Further compounding the problem is the trend adopted by modern services that are architected as microservices, where capabilities of the business logic are realized from a collection of loosely coupled, distributed service components. Each of the components must be configured and deployed on cloud platforms – sometimes federated – in a specific order, and where the entire service is realized through the composition of these components [2, 3].

MODELS'18, Copenhagen, Denmark
2018. . . . $15.00
DOI:

1.1 Motivating the Problem

Consider a use case of a service that is to be deployed on a cloud platform. The service comprises a PHP-based website application that stores data in a relational database and is to be hosted on two cloud provider platforms. Figure 1 shows the application topology consisting of two connected software stacks, i.e., a Web front-end and a MySQL database backend. The left stack of the web server model holds the business logic of the frontend, which needs to be deployed on Ubuntu 16.04 server virtual machine (VM). This VM needs to be managed using the OpenStack cloud platform, and product data is stored in a database as shown in the right-hand side stack of Figure 1. The backend database is a MySQL DBMS, which needs to be deployed on an Ubuntu 14.04 server VM, which VM needs to be managed on the Amazon Elastic Compute Cloud (EC2) platform.

1.2 Problem Resolution Requirements

Based on this case study we elicit the key challenges that drive the following requirements for the solution presented in this paper.

1.2.1 Requirement 1: Reduction in Specification Details in Deployment Phase and Auto-completion of Infrastructure Provisioning. As depicted in Figure 1, to start the PHP and MySQL-based website, first the VM or container should be spawned with Ubuntu server operating Systems in the OpenStack [29] and Amazon EC2 cloud platforms, respectively. The PHP module requires Apache httpd server as a dependency. So, Apache needs to be installed and configured along with PHP and Java. On the back end server, MySQL needs to be installed and configured. Besides, the web application requires installing a PHP Database Connectivity driver to access the database. Moreover, the database service should start before the PHP application service to run the WebApp properly. Thus, a user requires extensive domain knowledge to provision even a simple web application correctly. Such provisioning is done typically in a tedious and error-prone script-centric way [8].

This motivating scenario illustrates the need to meet different platform and technology dependencies as well as deployment order

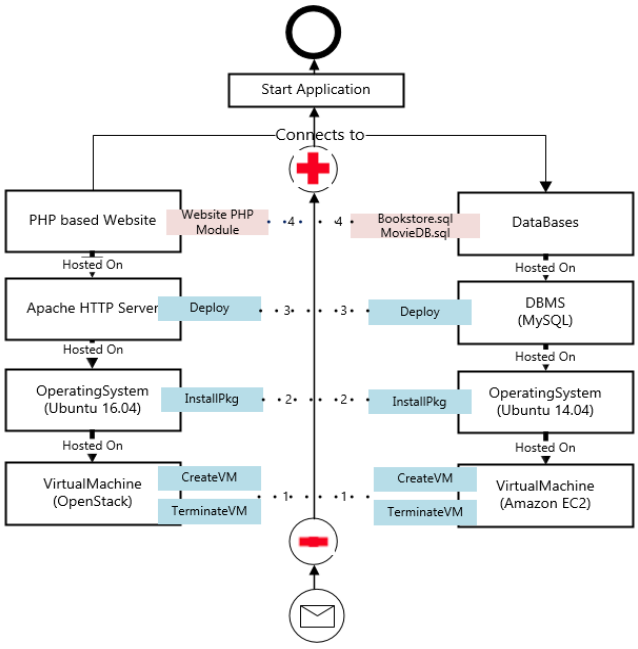


Figure 1: A TOSCA-compliant PHP- and MySQL-based Application Deployment Workflow

in which service components must be connected and started. A user is unlikely to possess the in-depth technical expertise needed to deploy such topologies across a range of choices. Thus, to improve productivity, and to reduce the learning curve and the effort required to provide intricate low-level details, there must be a way for service deployers to specify only the essential application components and their relationships intuitively. A framework with these capabilities should then be able to transform the partially defined business models to complete topologies using TOSCA-compliant Infrastructure-as-Code (IAC) solution, which keeps it technology and platform-agnostic and hence portable. [1]

1.2.2 Requirement 2: Support for Continuous Integration, Migration, and Delivery. Now suppose that in the use case of Figure 1, the enterprise wants to execute a management task to migrate the web front-end to Amazon's EC2 with the purpose of reducing the number of cloud providers used by their services. To migrate the frontend, the user must perform the following steps: (1) shut down the old virtual machine on OpenStack, (2) create the new virtual machine on Amazon EC2, (3) install the Apache HTTP server and the other dependencies, (4) deploy the PHP based frontend, and (5) set the database's IP-address, username, and password in the frontend's configuration. Moreover, migration can be stateful which means that the current state of the application needs to be actively replicated in a new VM and then the old VM be detached.

This migration strategy gives rise to several issues, such as having to deal with missing database drivers and missing configurations of the database service. These problems compromise the application's functionality, which requires experts who possess the knowledge to recognize these challenges in advance. As is often the case, manually performing the migration task requires sheer technical expertise about the different APIs and employed technologies. Application extensibility is another issue that users often must deal with, e.g., adding one database server or analytics tool with the application, for which similar challenges arise. This motivates the need for the fully automated platform that can generate right deployment plans and improve productivity and system robustness.

[1]TOSCA [26] is an OASIS standard for vendor-neutral topology and orchestration specification for cloud-based applications.

1.3 Limitations of Existing Approaches

The DevOps community today leverages orchestration solutions such as Cloudify, Apache Brooklyn, and Kubernetes, among others in conjunction with automation tools such as Ansible, Puppet, and Chef, among others. This state of the art (i.e., the extensive choices available to the developer) is reflected in Figure 2. Although IAC helps mask the heterogeneity stemming from the differences in the cloud platforms and their resource types, it requires elaborate specification of service topologies comprising requirements, functionalities, dependencies and relationships of the components. For instance, depending on the technology used such as MySQL versus PostgreSQL or PHP versus Node.js, the script must include the appropriate drivers. Moreover, additional dimensions of variability (i.e., addressing application's compatibility and cloud providers' incompatible APIs) as depicted in Box 1 of Figure 2 further amplifies the manual effort which is daunting, tedious and error-prone. Finally, existing approaches do not account for pre-deployment validation to check if the end-user requirements and software dependencies are met.

1.4 Solution Approach

To address these challenges (depicted in Box 1 of Figure 2) and meet the requirements of the desired solution, we propose a model-driven and scalable, rapid provisioning framework called CloudCAMP that significantly reduces the burden on the service providers to deploy and manage their applications. CloudCAMP abstracts the application component specifications and cloud provider specifications into intuitive representations as depicted in Box 2 of Figure 2. CloudCAMP complies with *Topology and Orchestration Specification for Cloud Applications (TOSCA)* specification [26], which enables the creation of portable and interoperable cloud applications. TOSCA defines an XML-based modeling specification [28] that formalizes the topology and management tasks of an application in the form of a plans-as-a-service template separating the application from the cloud provider-specific API [6, 27].

In this context, we make three contributions as follows:

(1) **Masking the low-level details of service specification and variability in tools and platforms:** We present an extensible metamodel for a domain-specific modeling language (DSML) of CloudCAMP that captures the commonalities and variabilities in the application component specifications, as well as operating systems, and cloud providers endpoint specifications. We also obtain the dependencies and intrinsic relationships among application components. Finally, the metamodel also captures the scaling and replication capabilities.

(2) **Model-to-Infrastructure-as-code (IAC) Transformation using a Knowledge Base:** We describe the generative aspects of

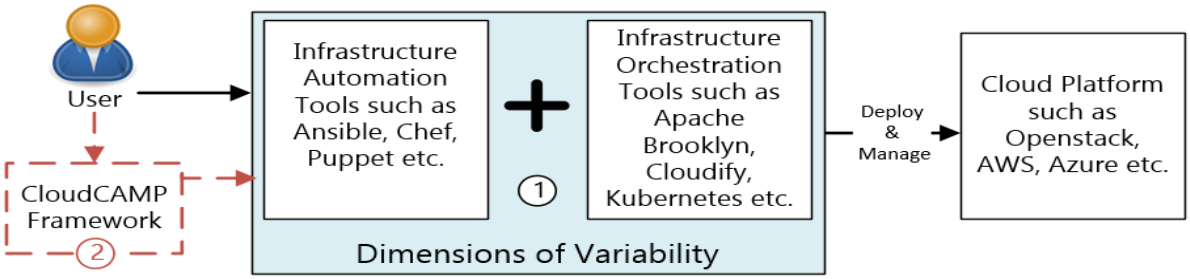


Figure 2: Box 1 (blue color) depicts the responsibilities of service deployment team, which is to define the low-level scripts so that existing automation tools can configure the application components and orchestration tools can provision the infrastructure for application components and execute them on heterogeneous cloud environments. Box 2 depicts the contributions of this paper which introduces a self-service framework and automates whole infrastructure design solutions for these tools.

CloudCAMP DSML that synthesizes a full-blown deployment IAC model using an underlying extensible knowledge base, which ensures the correct execution order of application components by checking the relationships among the application components of the business model. It is integrated with known constraint checking capabilities to verify the correctness of the model.

(3) **Concrete implementation and validation:** We implement our approach in a cloud-based model-driven engineering (MDE) environment called WebGME [24]. The generative capabilities of our approach are applied as a WebGME plugin which generates IAC code based on the TOSCA standard. These TOSCA-compliant IAC solutions are executed by our plugin to deploy, (re)start, stop the application components and manage the cloud services. We present a concrete implementation of CloudCAMP and evaluate its capabilities for representative case studies.

1.5 Organization of the paper

The rest of the paper is organized as follows: Section 2 presents a brief survey of existing solutions in the literature and compares to our solution; Section 3 presents the design of CloudCAMP; Section 4 evaluates our metamodel for a prototypical case study and presents a user survey; and finally, Section 5 concludes the paper alluding to future directions.

2 RELATED WORK

The problem of deployment and management abstraction has been explored in the area of cloud automation and orchestration. In this section, we compare existing efforts in the literature with our work. A preliminary version of CloudCAMP appears in [5].

Cloud orchestration tools like Apache Scalr (https://scalr-wiki.atlassian.net/wiki/display/docs/Apache), CloudFoundry (https://www.cloudfoundry.org/), Cloudify (http://getcloudify.org/) are excellent toolchains to deploy and manage applications on any cloud providers. They provide sophisticated techniques to monitor the health of the application and to migrate between the cloud providers using standardized approaches. However, they all suffer from the same limitations of requiring the users to define the complete and correct deployable model with all the functionalities and features. The use of these toolchains adds the burden of configuring the application components and integrating pre-deployment verification on application developers.

Although script-centric DevOps community provides toolchains for eliminating the disconnect between developers and operations providers [20], these tools still incur limitations in providing a self-service provisioning platform. In this context, Alien4Cloud [10] proposes a visual way to generate TOSCA topology model, which can be orchestrated by Apache Brooklyn. CHOReOS [22] also supports large-scale service deployment in the cloud. However, building the proper topology even using an MDE approach combined with the TOSCA specification needs domain expertise. This is precisely where CloudCAMP abstracts all the application and cloud-specific details in metamodel of its DSML and transforms the business model to TOSCA-compliant IAC using an extensible knowledge base comprising application-related dependencies.

Several patterns-based approaches are proposed to reduce the complexity of service deployment [14, 15, 23]. They differentiate between business logic and the deployment of service-oriented architecture platform. Each pattern offers a set of capabilities, and characteristics. Likewise, model-based patterns of proven solutions for the functional and non-functional properties of application service deployment in cloud infrastructures [19] are also proposed. For instance, MODAClouds (MOdel-Driven Approach for the design and execution of applications on multiple Clouds) [1, 13] allows users to design, develop and re-design application components to operate and manage in multi-cloud environments using a Decision Support System. Similar to CloudCAMP, they also support reuse and role-based iterative refinement in a component-based approach. However, their deployment plan generation lacks verification and extensibility. They also did not consider distributing application components in a heterogeneous cloud environment.

Several efforts come close to the CloudCAMP idea. For instance, ConfigAssure [25] is a requirement solver to synthesize infrastructure configuration in a declarative fashion. All the requirements are expressed as constraints on the configuration by the developer, and the provider predefines a configuration database containing variables as a deployment model. Kodkod [30] is a relational model finder which takes these arguments as a first-order logic constraint

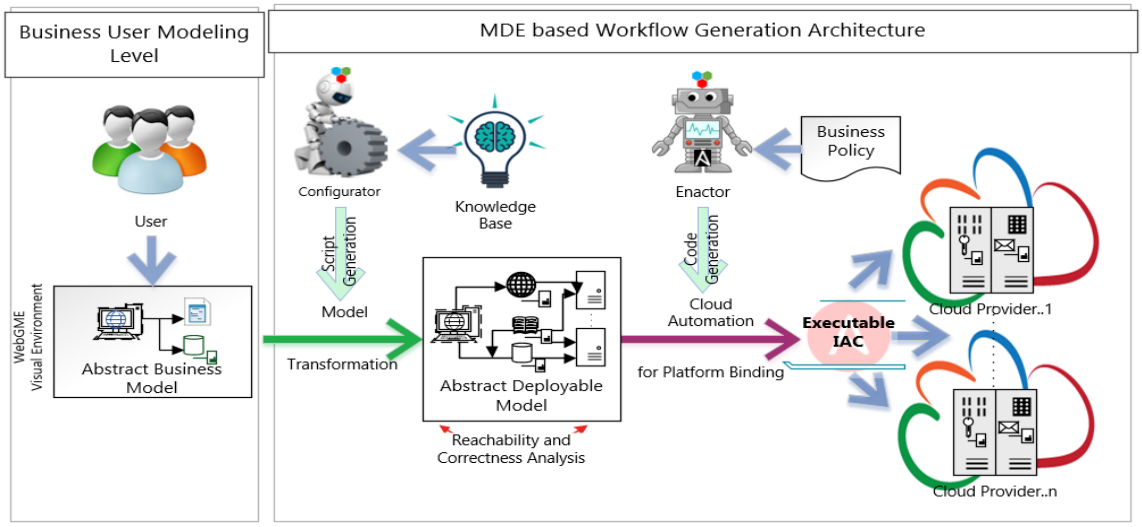


Figure 3: The CloudCAMP Workflow

in the finite domains. Engage [16] deploys and manages the application from a partial specification using a constraint-based algorithm. Aeolus Blender [11] comprises the configuration optimizer Zephyrus [12], the ad-hoc planner Metis [21], and deployment engine Arnomic. Zephyrus automatically generates an abstract configuration of desired system based on partial description. They guarantee meeting all the end-user requirements for software dependencies and provide an optimal solution for a given number of active virtual machines. In contrast to the use of knowledge base in CloudCAMP, these efforts use a CSP solver to transform the business model. CSP solvers, however, can take significant time to execute, and moreover, defining constraints on the configurations requires expert knowledge of the system.

Similar to CloudCAMP, Hirmer et al. [18] focus on producing complete TOSCA-compliant topology from users' partial business relevant topology. Users have to specify the requirements directly using definitions of the corresponding node types or are added manually for refinement. Their completion engine compares this specification with target models and combines the missing components to make it a fully deployable model, and then the service components can be executed in the right order using an OpenTOSCA toolchain [7]. CELAR [17] combines MDE and TOSCA specification to automate deployment cloud applications, where topology completion is fulfilled by requirement and capability analysis on node template. Unlike these efforts, the model transformation in CloudCAMP is based on querying the knowledge base and idempotent infrastructure code generation.

3 CLOUDCAMP DESIGN

This section delves into the design of CloudCAMP (Figure 3) showing how it meets the requirements discussed in Section 1.2.

3.1 System Architecture and Implementation

To better appreciate the CloudCAMP solution presented below, consider the fundamental requirements outlined earlier where a deployer needs only specify the application components, such as a Web App as shown in Figure 4, using intuitive notations provided by a DSML, and have the DSML transform it into deployable artifacts. Thus, the first step is for the user to utilize an intuitive, higher-level modeling framework that simplifies the modeling of business logic and automatically takes care of non-business centric deployment and management artifacts.

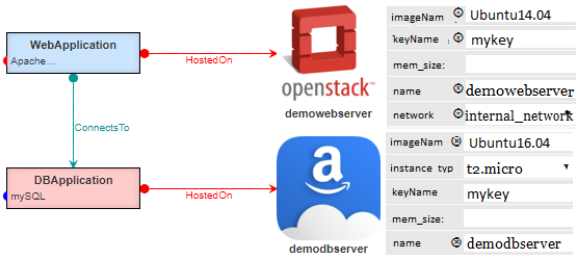


Figure 4: Desired Level of Abstraction for a WebApp Business Model

To that end, we have architected CloudCAMP's cloud-based service provisioning workflow as depicted in Figure 3. Below we explain the roles of the different actors involved:

(1) *Business User Modeling*: A business application is modeled as a compendium of different application components where the user has to select appropriate application component types from the CloudCAMP application pane to deploy the associated application code. The user needs to specify the host types and the cloud provider on which they want to deploy the application components, e.g., components in Figure 4.
(2) *Configurator*: This actor is responsible for transforming each abstract application component to a cloud automation task (e.g., Ansible-specific), which is an abstract deployable model for each application component. A user is required to specify key

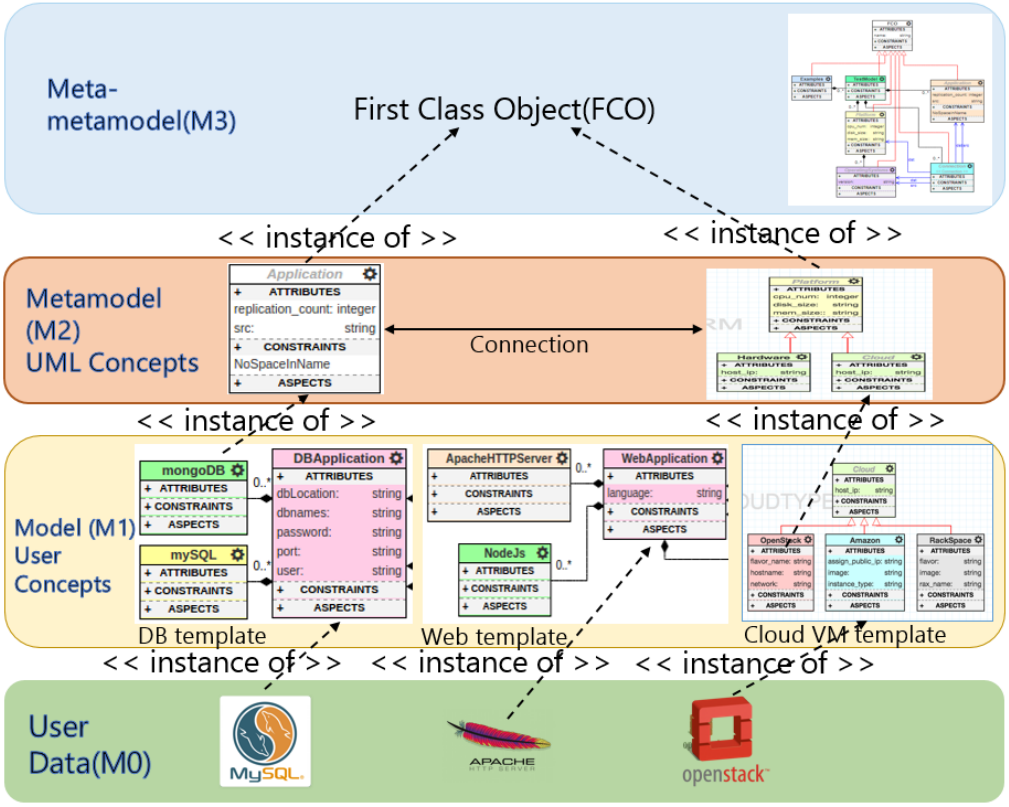


Figure 5: A Partial MOF model of CloudCAMP DSML and Platform

variability points in the model (e.g., the orchestration and configuration tool used). Given such an abstract description of a cloud application model as an input, the Configurator realizes the operational mapping between the specified application components and their attributes. Then it uses the workflow generation Algorithm 1 from Section 3.3 to query the knowledge base and finds all the required dependencies that are needed based on the selected host type.

(3) *Enactor*: It generates the infrastructure design workflow of IAC solutions by abiding by the business rules and cloud infrastructure specifications. The orchestration tools execute all the tasks in proper order to deploy and run the business applications correctly across cloud providers.

(4) *Knowledge Base:* A knowledge base is needed for auto-completing the partially specified deployment models. To that end, we predefine the software dependencies for application components in relational tables and maintain a normalized form with indexing to make the database easier to use. All the software packages needed for a particular application component are listed in the tables, and it is dependent on operating system and its version. For new application component types, the application developer needs to populate the tables with all software dependencies during application development phase.

The CloudCAMP DSML shown in Figure 5 is developed using the WebGME MDE framework (www.webgme.org). WebGME is a cloud-based framework that offers an environment for DSML developers to define their language and create model parsers that can serve as generators of code artifacts. The CloudCAMP runtime platform uses a microservices architecture comprising three services: (a) the modeling infrastructure, i.e., the WebGME UI, and orchestration and automation frameworks forming one service, (b) the WebGME modeling details are stored in a MongoDB NoSQL database, and (c) the knowledge base is hosted as a MySQL database service. The microservices are connected through the API endpoints and placed behind a HAproxy (http://www.haproxy.org/) load balancer. Thus, all the services can independently scale to support parallel spawning and configuration of multiple VMs or containers in the cloud platform.

3.2 CloudCAMP Domain-specific Modeling Language (DSML) Design

The CloudCAMP DSML abstracts the design complexities by separating the application from deployment and infrastructure technologies according to TOSCA specification as described in Requirement 1.2.1.

Design Rationale for CloudCAMP DSML Metamodels: DSMLs are realized through one or more interrelated metamodels that capture the DSML's syntax and semantics. In our case, to transform the business model to a full-blown deployment model, we needed to capture various facets of the application and cloud specifications in

our metamodel. CloudCAMP's deployment modeling automation metamodel was developed by harnessing a combination of (1) reverse engineering, (2) dependency mapping across heterogeneous clouds, (3) dependency mapping across different operating systems and their versions, (4) semantic mapping, (5) business policy, and (6) prototyping. Capturing this variability helped to enrich the expressive power, multi-cloud tool support and interoperability of the platform. Prototyping and reverse engineering helped to identify the different application components, cloud and operating system specific endpoints. The dependent software packages, their relationship mapping and configuration templates were realized in the metamodel by querying the knowledge base. The set of available building blocks, requirements, policies, and other information concerning the implementation of the services and all other known constraints are pre-defined in the high-level application metamodel.

To that end, CloudCAMP provides different node types, which are application components such as WebApplication, DatabaseApplication, DataAnalyticsApplication, etc., and various cloud providers such as OpenStack, Amazon, etc. The goal then is to concretize the abstract node type by matching the application developers' desired specification with the pre-defined functionalities captured in the CloudCAMP metamodel. Concretized node templates are then bound to specific cloud provider types, and their VMs and operating system to create a dependency graph that has to be executed to deploy the application on the desired target machine.

Snippets of the metamodels for CloudCAMP are shown in the M1 and M2 level of Figure 5, which are based on the Meta-Object Facility (MOF) standard provided by Object Management Group (OMG)[2]. Using our DSML, the application deployer can configure the node in defined cloud platform or particular target system without providing any deployment or implementation artifacts that contain code or logic. [3] The CloudCAMP metamodels are extensible and reusable, so new component types and platforms can be added as required in the CloudCAMP metamodel.

Metamodel for the Cloud Platforms: In designing the metamodel for cloud platforms, we observed (i.e., reverse engineered) the process of hosting applications across different cloud environments, and captured all the commonalities and variabilities. The specifications for different cloud platforms (OpenStack, Amazon EC2, and Azure) for provisioning virtual machines (VMs) with different operating systems (OS) are captured as the variability. The deployer can select a pre-defined VM flavor, available networks, and the available images, all of which are obtained by querying the specific cloud platform to populate our metamodel. The deployers can then choose their desired OS images to spawn the VMs/containers. They also must specify their environment file, the secret key for the selected cloud host type, which are the endpoints to bind to a particular cloud provider. Optionally, a pre-deployed machine can be specified by providing the IP address and OS.

Metamodel for Application Components: For cloud-hosted services, CloudCAMP provides different node types, which are application components such as WebApplication, DatabaseApplication, DataAnalyticsApplication, among others. For instance, the metamodel enables a deployer to choose the web server attribute (e.g., Apache web server), language for the code (e.g., nodeJS or PHP), and the database server attribute (e.g., MySQL or Redis database) from the provided dropdown list. Reverse engineering the application deployment process, we have captured and included these metamodeling concepts. The metamodel has been designed for extensibility so that in future we can add more application node types, e.g., stream processing operators that execute on systems such as Apache Kafka (https://kafka.apache.org/).

As an example, we will walk through the specifications needed to be captured for the WebApplication and DBApplication component types. As shown in the M0 level of Figure 5, the HTTP servers for the webEngines are captured in WebApplication component type, and that is related to the node template for a WebApplication. The development languages and frameworks (Node.js, PHP, Django, etc.) of the webApplication is taken as attributes in the software property as depicted in the M1 level of Figure 5, which is derived from Application type of M2 level, and our modeling tools metamodel is shown in M3 level. Similarly, as shown in the M0 level of Figure 5, the software for the database types are captured in DBApplication component type, and that is related to the node template for the Database Application. Related features, such as the user id, password, specific binding port number of the Database application, etc. are stated as attributes, which is captured in the M1 level of the MOF.

Defining the Relationship among Components: Four relationship types bind the node types in the metamodel as follows:

(1) *'hostedOn'* relationship type implies the source node type is required to be deployed on the destination node type, e.g., Webserver is hosted on Ubuntu 16.04 in OpenStack.
(2) *'connectsTo'* relationship type is used for deployment ordering to relate the source node type's endpoint to the required target node type endpoint if they are dependent on each other. The node types linked by 'connectsTo' can be configured in parallel, but the service at the source node needs to deploy only after starting the target node.
(3) *'deleteFrom'* connection type defines the source node type is required to be removed from the end node type.
(4) *'migrateTo'* connection type defines the source node type that is to be migrated to the end node type. The 'migrateTo' relation type cannot be defined without a 'deleteFrom' connection type to assure correctness of the business model.

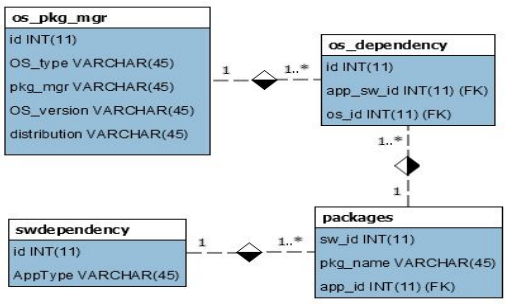


Figure 6: The Entity-Relation(ER) Diagram of CloudCAMP knowledge base

[2] http://www.omg.org/
[3] Due to space constraints, we do not show detailed screenshots of each metamodel. The interested reader can find these details in [4].

3.3 Generative Capabilities of CloudCAMP DSML

CloudCAMP DSML provides generative capabilities for an IAC solution by interpreting the instances of models for which it incorporates a built-in knowledge base. The CloudCAMP DSML in WebGME is built using JavaScript, NodeJS, and a MySQL database.

Knowledge Base for Generation of Infrastructure-as-code Solution for Deployment: The ER diagram of the knowledge base is depicted in Figure 6, and it reflects the artifact sets stored in the knowledge base. We have structured it as four tables: *os_pkg_mgr, os-dependency, packages* and *swdependency* to build the knowledge base. We store all the operating systems, their distributions, and versions in the os_pkg_manager table, and all available application component types, e.g., PHP based web application, MySQL based DB applications, etc. are stored in swdependency table. All the software packages needed for a particular application type is found using reverse engineering and stored in the packages table. For example, to install the scikit-learn package (http://scikit-learn.org), one needs to install python, python-dev, python-pip, python-numpy, etc. using apt-manager package, and then the scikit-learn package can be installed from the pip package manager. In a relational table called os_dependency, we map the software packages and their versions with operating systems and their versions and store it as a key-value pair. For instance, to install java8 on Ubuntu 16.04, we need different packages than to install java8 on Ubuntu14.04. We build the lookup table manually to handle these variability points. For new application component types, the application developer needs to populate the tables with all software dependencies.

CloudCAMP's generative capabilities are enabled via a WebGME plugin, which is invoked by a user after the modeling process. It generates and executes IAC as described in Algorithm 1. The VMs are spawned in the specified cloud platform based on the destination of 'HostedOn' connection [Lines 8-14]. Wherever possible, CloudCAMP will ensure that scripts specific to provisioning run in parallel to provide faster deployment. Once the VMs are spawned, *GenerateConfig()* queries the knowledge base to populate appModel [line17]. Based on the user's business model specifications, CloudCAMP fetches the desired results. Then, the Configurator component fills application-specific predefined configuration templates and generates infrastructure code, e.g., Ansible, for specific application components [line 29-34]. A similar approach is taken to configure the service-specific containers.

A sample of the automated SQL script used to query the knowledge base for deployment script generation is shown below:

```
SELECT pkg.pkg_name FROM packages pkg, swdependency dep WHERE
pkg.app_id = dep.id AND pkg.apptype = <LANGUAGE> AND pkg.sw_id IN
(SELECT app_sw_id FROM os_dependency WHERE os_id IN
(SELECT id FROM os_pkg_mgr WHERE
Concat(os_type, os_version)=<OS>,<VERSION>))
```

Determining the Order of Deployment and Execution: The Enactor component, which is a NodeJS script, builds the dependency tree for the application types defined in the metamodel and feeds it to the orchestration workflow engine. We generate scripts for automation tools (e.g., Ansible playbooks) for different component types, and these tools can in turn dispatch tasks to multiple hosts in parallel. If there is a 'connectsTo' relationship in the model, we let the dependent script complete first by defining the dependency chain [Line 19-22]. All the 'HostedOn' dependent building blocks run in a linear fashion. Thus, the Enactor remotely connects to the deployment hosts and deploys the application in proper order. The application deployer can deploy a predefined application using our tool without writing a single line of code and without any significant domain expertise.

Algorithm 1: Deployment Script Generation

```
1  cloudModel ← Objects to store cloud specs
2  appModel ← Objects to store app specs
3
4  Procedure GenerateIAC()
5  if ConectionType == 'HostedOn' then
6      cloudType ← the destination node of connection
7      appType ← the source node of connection
8      if cloudType == 'Desired Cloud Platform' then
9          while !cloudModel.empty() do
10             Traverse the cloudModel
11             Fill 'cloudType' specific API Template
12             Generate 'cloudType' specific workflow script
13             Execute script to spwan VMs
14         end
15     end
16     IPAddress(es) ← IP Address of target machine
17     GenerateConfig(IPAddress(es),appType)
18     Check Connection Type among app components
19     if ConectionType == 'connectsTo' then
20         Find the source and destination application type
21         Prepare workflow script to execute destination
             script(s) first and source script later
22     end
23 end
24
25 Procedure GenerateConfig()
     Input: IPAddress(es) of Application Component Type
26 Create empty Ansible Tree Structure
27 Fill 'hosts' with IPAddress(es) of App Component Location
28 if appComponent == 'Desired Application Type' then
29     while !appModel.empty() do
30         Traverse the appModel
31         Query dataBase for appType = 'appComponent'
32         Fill 'appType' specific API Templates
33         Create complete Ansible Tree Structure
34     end
35 end
36 Wait for SSH to be enabled in target machine(s)
37 Run workflow script to execute tasks in parallel
```

Generation of Infrastructure-as-code for Migration: The algorithm for generating a migration workflow (Requirement 1.2.2) is portrayed in Algorithm 2. The 'deleteTo' connection type specifies from where the user wants to move the application components and attaches a 'migrateTo' connection type to indicate the destination. The migrationType (stateless or stateful) must be selected, and depending on that, CloudCAMP decides to checkpoint application state or not before terminating the old VMs/containers [Line 17-23]. The 'migrateTo' relation type cannot be defined without 'deleteFrom' connection type to ensure correctness of the model.

Although actions are taken for live migration, an application component from one VM to another depends on the application

Algorithm 2: Migration Script Generation

```
cloudModel ← Objects to store cloud specs
appModel ← Objects to store app specs

Procedure MigrationIAC()
if ConectionType == 'deleteFrom' then
    cloudType ← the destination node of connection
    appType ← the source node of connection
    IPAddress(es) ← IP Address of target machine
    if cloudType == 'Desired Cloud Platform' then
        Generate 'cloudType' specific workflow script
        Execute script to terminate VMs
    end
end
if ConectionType == 'migrateTo' then
    GenerateIAC()
end
if migrationType == 'stateless' then
    Execute deletion and migration scripts in parallel
end
else if migrationType == 'stateful' then
    Checkpoint current application state on old machine
    Restore checkpoint on the current machine
    Execute deletion and migration scripts in parallel
end
```

component type, which is a hard problem. For example, live migration of DBApplication needs two-phase commit protocol, and consensus algorithm to make it reliable. For the sake of simplicity, in the Algorithm 2 we generalize our approach. Our future work will consider more complicated scenarios of live migration and application consistency and availability issues.

According to Algorithm 2, it will spawn a new VM with the new operating system for the 'migrateTo' destination node. For Stateful migration, our platform creates a manager node with a load balancer, and deploy the application on the current node. From that point of time, load balancer redirects all the new request to the current node, and it checkpoints the current state of the old node and restores it in the current node. Finally, it detaches the load balancer node. Thus, it produces the full infrastructure-as-code solution along with the related configuration files. All of these complete the Ansible layout tree structure helps to migrate application components from one node to another node.

Additionally, CloudCAMP can also handle continuous delivery and component addition/deletion, which is just a matter of updating the model with addition or removal of a component. For instance, to add a new database server, a user extends the model with a DBApplication node type and 'connectsTo' relationships from the webserver to the database server. CloudCAMP will generate IAC for the newly added component and executes it to deploy added component without hampering availability of the existing application. Since Ansible is idempotent, it always sets same configuration in target environment regardless of their current state.

Constraints checking of Business Models: We also validate the business model by checking for constraint violations thereby ensuring that the models are "correct-by-construction." We verify the correctness of the endpoint configurations for application component types, the relationship types, cloud-specific types, etc., and the business model as a whole before generating any infrastructure code. Examples of some of the constraints are shown below:

- ∀ Applications ∈ WebApplication ∃! WebEngine
- ∀ Applications ∈ DBApplication ∃! DBEngine
- ∀ Platform ∈ Openstack ∃! imageName
- ∀ Applications ∈ DataAnalyticsApp ∃ processEngine etc.

Thus, we validate the business model by satisfying the constraints and notify the user if there are any discrepancies in their business model.

4 VALIDATION AND EVALUATION OF CLOUDCAMP PLATFORM

This section describes results comparing the time and effort incurred in deploying application use cases using (a) manual efforts, where the operator must log into each machine and type the commands to install packages and deploy the applications, (b) manually writing scripts to deploy these applications, and (c) using the CloudCAMP framework.

4.1 Case Study 1: LAMP-based Web Service Deployment User Study

Use Case: prototypical three-tier Linux, Apache, MySQL, and PHP (LAMP)-based microservice architecture deployment is similar to the motivating example described in Section 1.1. Figure 7 shows application topology that illustrates the modeling effort in CloudCAMP, where the PHP-based web application needs to be 'HostedOn' on OpenStack platform on Ubuntu 16.04 VM, and the database service will be deployed on another OpenStack platform on Ubuntu 14.04 VM, and these two tiers must have 'ConnectsTo' relationship between them.

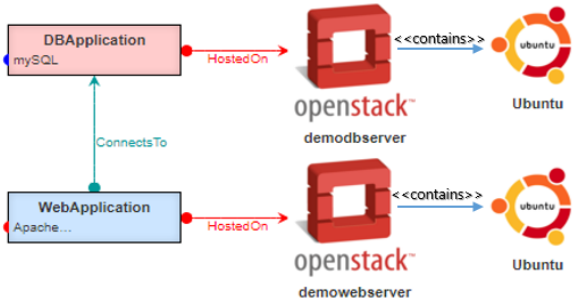


Figure 7: Sample LAMP Application Model

In addition to the structural model as shown, a user must also supply appropriate parameters to the different model elements. For instance, for the WebApplication node type, the language for source code (e.g., PHP) has to be specified as shown in Figure 8a along with the web server software to be used (e.g., Apache). Likewise, for the DBApplication node type, attributes such as database name, location, port, user, password, etc. need to be specified as shown in Figure 8b along with the database system used (e.g., MySQL). Since we reverse engineer the applications, all endpoints and all constraints are predefined and specified in the metamodel.

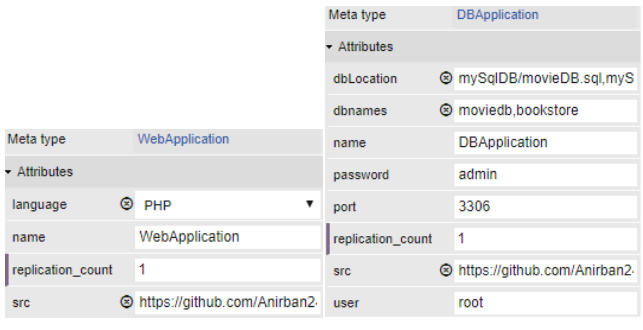
Manual Effort Rationale : In contrast to CloudCAMP, in a fully manual effort, the users will need to configure the files, create the handlers to specify the deployment order in the desired host, log into each host where the application components are deployed and manually install the packages, configure the software packages and finally start the different components in the correct order. In the manual scripting case, the user will first incur a significant learning curve for Automation and Orchestration tools. We expect that despite improving automation via these tools, the user will still suffer trial-and-error, which is likely to be amplified for complex deployment scenarios and hence decrease the productivity.

Quantitative Evaluation based on a User Study. We conducted a small user study in a Cloud Computing course for case study 1 involving sixteen teams of three students each. We measure both the time taken, and efforts (a) for a fully manual effort, (b) for writing scripts in Ansible and executing these manually and (c) using the CloudCAMP framework to deploy the scenario. Questionnaire as shown in Table 1 was created to conduct the survey. For each question, the students were asked to evaluate on a scale of 1–10 where 1 is easiest and 10 is hardest.

Table 1: Survey Questionnaire: For Q1–Q3, rate on a scale of (1-10), where 1 is easiest, 10 is hardest.

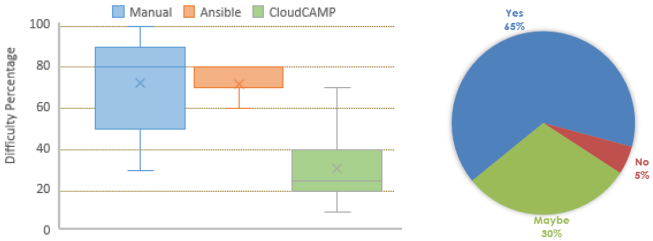
Num	Question
Q1	How easy is it to deploy PHPMySQL application manually?
Q2	How easy is it to deploy PHPMySQL using DevOps tool like Ansible?
Q3	How easy is it to deploy PHPMySQL using CloudCAMP?
Q4	How much time and effort did you require to deploy the application manually (in minutes)?
Q5	How much time and effort is required in deploying the application using DevOps tool like Ansible (in minutes)?
Q6	How much time and effort is required deploying the application using CloudCAMP (in minutes)?
Q7	How likely are you to use the CloudCAMP platform to deploy applications in future?

Responses to Q1, Q2, and Q3: Ease of use: As seen from Figure 9a, the "ease of use" rating for the CloudCAMP platform is much higher compared to manual and scripting efforts. The median difficulty in the manual effort is rated as 72.2%, and median difficulty in scripting effort is rated as 71.6%, while the median difficulty rating for CloudCAMP use is 30.9%. The visual drag and drop environment helps users to quickly deploy various scenarios of business application topology in distributed systems.



(a) (b)

Figure 8: (a) specifications related to WebApplication type and (b) specifications related to DBApplication type



(a) (b)

Figure 9: (a) Comparing difficulty percentages to deploy services in different approaches, (b) Likeliness of using CloudCAMP for future cloud services deployment

Responses to Q4, Q5, and Q6: Time to complete the whole deployment: The LAMP stack deployment with the provided source code comprises installing and configuring PHP, Apache HTTP server, and MySQL RDBMS. The average time the students took to complete the entire deployment process manually is 171 minutes, and the average time is 516 minutes to script and debug Ansible code correctly, whereas our rough estimates for students using the CloudCAMP-based topology creation and deployment will be only 15-20 minutes for the first time users. The average line of code written for the deployment process is 315 lines as per the survey as shown in Table 2.

Table 2: For Q5–Q6, median and mean±std.dev for deployment time, Lines of code written for deployment, migration time and Lines of code written for migration.

	Deployment Time(mins)	Lines to Deploy	Migration Time(mins)	Lines to Migrate
median	510	300	720	550
mean ± std.dev	516±244	315±47	653±231	553±142

Response to Q7:As shown in Figure 9b, 65% of the respondents agreed to use CloudCAMP tool to deploy cloud applications in future, whereas 30% are still unsure. Results from our user study strengthen our belief that the CloudCAMP platform will be a very resourceful tool for business application deployers. We have also conducted a user study specifying to create Docker Containers[https://www.docker.com/] and deploy the LAMP architecture inside it using scripting tools and found very similar results. Therefore, the benefits of the automation accrued using CloudCAMP can easily be understood for these use cases.

4.2 Case Study 2: Application Component Migration for LAMP-based Web Service

CloudCAMP platform also supports application component migration with ease for which we have two connection types 'deleteFrom'

and 'migrateTo'. As described in Scenario 1.2.2, suppose the user wants to migrate the database application component from one machine to another machine, which resides on different OpenStack platform. This assignment was to migrate the 'stateful' MySQL database service from one node to another node, and the students are asked to add load balancer node to make the service available all the time. CloudCAMP generates a new workflow structure based on the changed user specifications as described in Algorithm 2.

Responses to Q4, Q5, and Q6: Time to complete the whole migration: The average time the students took to write the scripts to complete the entire migration process is 653 minutes, with the median of 720 minutes as shown in Table 2. Whereas our rough estimates for students, using the CloudCAMP-based topology migration will be only 10-15 minutes for the first time users. The average lines of code written for the migration process are 553 lines as per the survey as shown in Table 2. According to the survey results, it is reasonable to claim that using CloudCAMP platform will significantly increase the productivity and efficiency of the application deployment and management team.

4.3 Case Study 3: Video Streaming Service Deployment and Management in Heterogeneous Environment

Our CloudCAMP GUI environment is capable of provisioning hundreds of machines in parallel based on the user-specified dependency and can deploy, manage, and monitor all the resources by selecting proper application components from the Application panel.

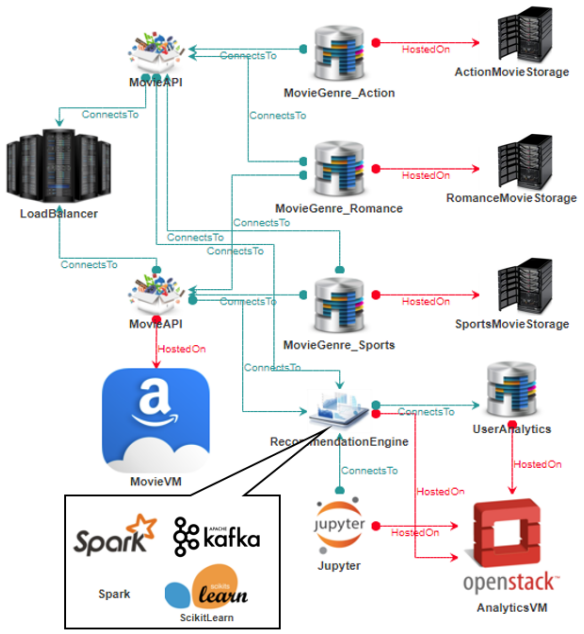


Figure 10: A Realworld Movie Service Provider's Application Components Stack

As a complicated application case study, in Fig. 10, we illustrate a scenario to aid Video Streaming Service deployment in the heterogeneous cloud environment. The frontend MovieAPIs needs to be hosted on Amazon EC2, and there will be different databases, which is MySQL databases, where all data regarding movies, their genres will be stored in storage hardware, and based on end user's selection, a video will be streamed. A load balancer is needed for the frontends to handle the number of user requests, and the entire user search pattern, their likes, and dislikes will be streamed to a database using stream processing system, e.g., Apache Kafka. The analytics VM needs to be hosted in private OpenStack cloud for privacy issues. Then from the data store in data analytics VM, recommendation models need to be trained online using Machine Learning toolkit, e.g., Apache Spark MlLib and provide a list of recommended movies to the user. Hosting such a grouped architecture is complicated, and service providers have to write thousands of lines of script to do that and to handle the replication strategy more machines need to deployed which is hidden from Fig. 10 for simplicity. CloudCAMP provides the capability to build infrastructure through declarative GUI based templates rather than writing thousands of lines of scripts and figuring of the software dependencies.

Moreover, continuous deployment of application components is relatively straight-forward in ClodCAMP. Suppose, if scikit-learn software toolkit needs to be deployed for the analytics, the user needs to drag the application component and specify the location of the host where he/she wants to deploy the application component. Migration and deletion are also supported similarly. Based on ballpark estimation, to architect Video Streaming Service deployment, IAC will be approx. 500-600 lines of code for each node and an hour effort even for expert users, which can be handled by CloudCAMP in an automated 'correct-by-construction' fashion.

5 CONCLUSIONS

This paper presented a model-driven approach for an automated, deployment and management platform for cloud applications. It aids the application deployer in modeling the service provisioning at a higher level of abstraction, and deploy its code without requiring significant domain expertise and requiring only minimal modeling effort and no low-level scripting. Using WebGME to define the CloudCAMP framework enables us to decouple its metamodel(s) and knowledge base from the generative aspects and allows extensibility.

Our future work will involve improving the soundness and robustness of the models using CSP solvers. We will also add reflection features to our framework so that the dynamic changes happening at the system level will be reflected back into the design view level, incremental deployment artifacts can be generated, and system changes effected. We will perform more user studies on more complicated scenarios to substantiate our claims.

CloudCAMP is available in open source from https://doc-vu.github.io/DeploymentAutomation/.

ACKNOWLEDGMENT

This work was supported in part by NEC Corporation, Kanagawa, Japan and NSF US Ignite CNS 1531079. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect views of NEC or NSF.

REFERENCES

[1] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, and others. 2012. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. IEEE Press, 50–56.

[2] Yogesh Barve, Prithviraj Patil, Anirban Bhattacharjee, and Aniruddha Gokhale. 2017. PADS: Design and Implementation of a Cloud-based, Immersive Learning Environment for Distributed Systems Algorithms. *IEEE Transactions on Emerging Topics in Computing* (2017).

[3] Anirban Bhattacharjee. 2017. MDE-based Automated Provisioning and Management of Cloud Applications. (2017).

[4] Anirban Bhattacharjee, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. 2017. *CloudCAMP: A Model-driven Generative Approach for Automating Cloud Application Deployment and Management.* Technical Report ISIS-17-105. Vanderbilt University, Nashville, TN, USA.

[5] Anirban Bhattacharjee, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. 2018. CloudCAMP: A Model-driven Generative Approach for Automating Cloud Application Deployment and Management. In *To Appear in IEEE Services Computing Conference, Work-in-Progress Session*.

[6] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2014. TOSCA: portable automated deployment and management of cloud applications. In *Advanced Web Services*. Springer, 527–549.

[7] Uwe Breitenbucher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. 2014. Combining declarative and imperative cloud application provisioning based on TOSCA. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 87–96.

[8] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and Johannes Wettinger. 2013. Integrated cloud application provisioning: interconnecting service-centric and script-centric management technologies. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 130–148.

[9] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems* 25, 6 (2009), 599–616.

[10] Jose Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel. 2016. Bidimensional cross-cloud management with TOSCA and Brooklyn. In *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE.

[11] Roberto Di Cosmo, Antoine Eiche, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. 2015. Automatic Deployment of Services in the Cloud with Aeolus Blender. In *Service-Oriented Computing*. Springer, 397–411.

[12] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. 2014. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 211–222.

[13] Elisabetta Di Nitto, Marcos Aurelio Almeida da Silva, Danilo Ardagna, Giuliano Casale, Ciprian Dorin Craciun, Nicolas Ferry, Victor Muntes, and Arnor Solberg. 2013. Supporting the development and operation of multi-cloud applications: The modaclouds approach. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*. IEEE, 417–423.

[14] Tamar Eilam, Michael Elder, Alexander V Konstantinou, and Ed Snible. 2011. Pattern-based composite application deployment. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*. IEEE, 217–224.

[15] Christoph Fehling, Frank Leymann, Ralph Retter, David Schumm, and Walter Schupeck. 2011. An architectural pattern language of cloud-based applications. In *Proceedings of the 18th Conference on Pattern Languages of Programs*. ACM, 2.

[16] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. 2012. Engage: a deployment management system. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 263–274.

[17] Ioannis Giannakopoulos, Nikolaos Papailiou, Christos Mantas, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. 2014. CELAR: automated application elasticity platform. In *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 23–25.

[18] Pascal Hirmer, Uwe Breitenbücher, Tobias Binz, Frank Leymann, and others. 2014. Automatic Topology Completion of TOSCA-based Cloud Applications.. In *GI-Jahrestagung*. 247–258.

[19] Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, and Trent Swanson. 2014. Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. (2014).

[20] Jez Humble and Joanne Molesky. 2011. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal* 24, 8 (2011), 6.

[21] Tudor A Lascu, Jacopo Mauro, and Gianluigi Zavattaro. 2013. A planning tool supporting the deployment of cloud applications. In *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*. IEEE, 213–220.

[22] Leonardo Leite, Carlos Eduardo Moreira, Daniel Cordeiro, Marco Aurélio Gerosa, and Fabio Kon. 2014. Deploying large-scale service compositions on the cloud with the CHOReOS Enactment Engine. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*. IEEE, 121–128.

[23] Hongbin Lu, Mark Shtern, Bradley Simmons, Meint Smit, and Marin Litoiu. 2013. Pattern-based deployment service for next generation clouds. In *Services (SERVICES), 2013 IEEE Ninth World Congress on*. IEEE, 464–471.

[24] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. 2014. Next Generation (Meta) Modeling: Web-and Cloud-based Collaborative Tool Infrastructure. *MPM@ MoDELS* 1237 (2014), 41–60.

[25] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. 2008. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management* 16, 3 (2008), 235–258.

[26] OASIS. 2013. Topology and orchestration specification for cloud applications. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf. (2013). OASIS Standard.

[27] OASIS. 2013. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html. (2013). OASIS.

[28] OASIS. 2013. TOSCA XML schema definition. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/schemas/TOSCA-v1.0.xsd. (2013). OASIS.

[29] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. 2012. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 55, 3 (2012).

[30] Emina Torlak and Daniel Jackson. 2007. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 632–647.