A Model-driven Approach to Automate the Deployment and Management of Cloud Services

Anirban Bhattacharjee*, Yogesh Barve*, Aniruddha Gokhale*
*Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, Tennessee, USA
Email: {anirban.bhattacharjee; yogesh.d.barve; a.gokhale}@vanderbilt.edu

Takayuki Kuroda†
†NEC Corporation
Kawasaki, Kanagawa, Japan
Email: t-kuroda@ax.jp.nec.com

***Abstract*—Although many provisioning tools are available for deployment and management of composite cloud services to overcome the manual efforts that are laborious, tedious and error-prone, users are often required to specify Infrastructure-as-Code (IAC) solutions via low-level scripting. IAC demands domain knowledge for provisioning the services across heterogeneous cloud platforms and incurs a steep learning curve. To address these challenges, we present a technology- and platform-agnostic self-service framework called CloudCAMP. CloudCAMP incorporates domain-specific modeling so that the specifications and dependencies imposed by the cloud platform and application architecture can be specified at an intuitive, higher level of abstraction without the need for domain expertise. CloudCAMP transforms the partial specifications into deployable Infrastructure-as-Code (IAC) using the Transformational-Generative paradigm and by leveraging an extensible and reusable knowledge base. The auto-generated IAC can be handled by 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 and scripting efforts versus using CloudCAMP.**

***Keywords*—cloud services, deployment and orchestration, automation, domain-specific modeling, knowledge base**

I. Introduction

Self-service application deployment and management are desired for enterprises to speed up time-to-market for their cloud services. This has become necessary since modern cloud services are often architected as microservices, where each of the components must be configured and deployed on cloud platforms in a specific order. Infrastructure and service provisioning for these complex use cases using low-level scripting environments degrade productivity and adversely impact the product time-to-market.

A. Motivating the Problem

Consider the case of a LAMP-based service deployment on a cloud platform. Figure 1 shows the desired cloud application topology consisting of two connected software stacks, i.e., a PHP-based web front-end and a MySQL database backend. The left side stack of the service model holds the business logic since the frontend will be deployed on Ubuntu 16.04 server. This server will be hosted in a virtual machine (VM), and the virtual environment is managed using the OpenStack cloud platform. The right-hand side of the stack shows the relational database, which is used to store and query the product data. The backend database is a MySQL DBMS, which will be deployed on the Amazon Elastic Compute Cloud (EC2) VM instance with an Ubuntu 14.04 server.

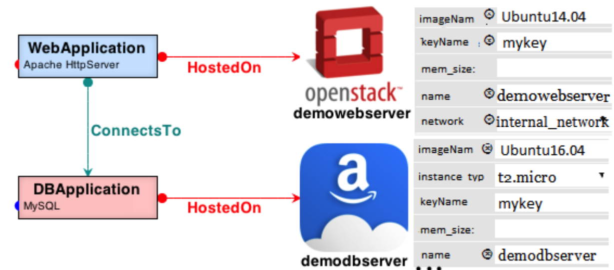

Fig. 1: Desired Level of Abstraction for a WebApp Business Model

The deployer needs to provision the PHP and MySQL-based e-commerce application stack from two aspects. In the cloud infrastructure provisioning aspects, the application topology needs to be woven into the execution environment which can be virtual machines (VM), containers or third party services. The deployer needs to select a proper image provided by the cloud provider, along with the security group, roles, network, number of instances, the storage unit to spawn new VM(s) in the target cloud platform.

In the service provisioning aspects, all the dependent software needs to be installed, and all the constraints need to be configured. For example, for the frontend of our motivating example, Apache Httpd needs to be installed and configured along with PHP and Java. Similarly, in the backend, MySQL needs to be installed and configured. Besides, the web application requires installing a PHP database connectivity driver to access the database. The installation process of the software depends on the operating systems, their versions, and package managers. Moreover, the database service should start before the PHP application service so as to run the WebApp properly. The IAC solution for the application provisioning requires all the installation and configuration details to execute the deployment plan.

B. Solution Approach: CloudCAMP

Our motivating example shows that a user must possess extensive domain knowledge to provision even a simple web application stack correctly. Although many provisioning tools are available for deployment and management of composite cloud

services, users are often required to specify Infrastructure-as-Code (IAC) solutions via low-level scripting. Instead a desired capability would require that (1) a deployer specify only the application components, such as a Web App, (2) the framework automatically transforms the business model into deployable artifacts.

To address these challenges and realize the desired capabilities, we propose a model-driven and scalable, rapid provisioning framework called CloudCAMP. CloudCAMP complies with TOSCA (Topology and Orchestration Specification for Cloud Applications), which enables the creation of portable and interoperable *plans-as-a-service* template for cloud services. Using TOSCA provides us standardization for decoupling software applications and its dependencies from the cloud platform specifications.

The key contributions in this paper include:

1) We present key elements of CloudCAMP's domain-specific modeling language (DSML) that masks low-level details of the application component specifications and cloud provider specifications and instead offers intuitive high-level representations;
2) We present the use of an extensible knowledge base and algorithms to perform Model-to-Infrastructure-as-code (IAC) transformations automatically; and
3) We present a concrete realization of CloudCAMP and validation in the context of real-world use cases.

C. Organization of the paper

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

II. Related Work

We compare existing deployment and management abstraction efforts in the literature with our work. A preliminary version of CloudCAMP appears in [1].

The community today leverages orchestration solutions such as CloudFoundry (https://www.cloudfoundry.org/), Cloudify (http://getcloudify.org/), etc among others in association with automation tools such as Ansible, Puppet, and Chef, among others. However, the different dimensions of variability (i.e., addressing application's compatibility and cloud providers' incompatible APIs) complicates the manual scripting effort using these tools. In this context, Alien4Cloud [2] proposes a visual way to generate TOSCA topology model, which can be orchestrated by Apache Brooklyn. However, building the proper topology even using model-driven approaches needs domain expertise. Unlike these approaches, CloudCAMP abstracts all the application and cloud-specific details in the metamodel of its DSML and transforms the business model to TOSCA-compliant IAC.

Several patterns-based approaches are proposed to reduce the complexity of service deployment [3]. Likewise, model-based patterns of proven solutions are used for service deployment in cloud infrastructures [4]. For instance, MODA-Clouds [5] allows users to develop and deploy 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. However, deployment plan generation lacks verification and extensibility.

ConfigAssure [6] is a requirement solver to synthesize infrastructure configuration in a declarative fashion. Aeolus Blender [7] comprises the configuration optimizer Zephyrus [8], the ad-hoc planner Metis, and deployment engine Arnomic. Zephyrus automatically generates an abstract configuration of the desired system based on a partial description. In contrast to the use of the knowledge base in CloudCAMP, these efforts use a Constraint Satisfaction Problems (CSP) solver to transform the business model. CSP solvers, however, can take significant time to execute and defining constraints on the configurations requires domain expertise, which is not needed in CloudCAMP.

Similar to CloudCAMP, Hirmer et al. [9] focus on producing complete TOSCA-compliant topology from users' partial business relevant topology using an OpenTOSCA toolchain [10]. CELAR [11] 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.

III. CloudCAMP Design and Implementation

This section delves into the design of CloudCAMP. We first put forth key requirements that CloudCAMP must satisfy and then discuss how our design meets those requirements.

A. Requirements for CloudCAMP Self-Service Platform

A self-service cloud platform such as CloudCAMP should require minimal specifications from the user who does not need to possess deep domain knowledge and maximally automate the provisioning process. Below we outline the key requirements that CloudCAMP must satisfy.

1) Requirement 1: Reduction in specification details: CloudCAMP must abstract the specification details from the users by identifying the commonalities of the provisioning stacks, which become the high-level reusable building blocks of the deployment and management pipeline that are captured as domain-specific artifacts. The minimal number of variability points then become the user inputs.

2) Requirement 2: Auto-completion of Provisioning : CloudCAMP must define transformation rules that convert the abstract business models into *correct-by-construction* complete, deployable TOSCA-compliant [12] Infrastructure-as-Code (IAC) solution [13].

3) Requirement 3: Support for Continuous Integration, Delivery and Migration: Since it is possible that an existing deployment may need to change the infrastructure provisioning (e.g., change the cloud platform) or change the application provisioning (e.g., replace the database server technology) or

both, CloudCAMP must decouple the two stages and seamlessly support continuous integration, delivery and migration.

B. CloudCAMP Domain-specific Modeling Language (DSML)

The CloudCAMP DSML abstracts the design complexities by separating the application from deployment and infrastructure technologies according to TOSCA specification as described in Requirement III-A1. The CloudCAMP DSML is developed using the WebGME MDE framework (www.webgme.org) and uses JavaScript, NodeJS, and a MySQL database.

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. The cloud providers and applications specifications, software requirements, policies, and other information concerning the implementation of the services and all other known constraints are pre-defined as the high-level building blocks in the metamodel. [1]

To that end, CloudCAMP provides different node types as per TOSCA specifications, which are the application components, and various cloud providers. The goal is to concretize the abstract node type by matching the deployers' desired specification with the pre-defined functionalities captured in the CloudCAMP metamodel and knowledge base. The concrete node templates are then woven to specific cloud provider types, and their VMs to create a dependency graph that has to be executed to deploy the application components in a specific order on the desired target machine(s). Using our DSML, the deployer can configure the node in a defined cloud platform or particular target system with ease.

1) 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 such as OpenStack, Amazon AWS, Microsoft Azure, etc for provisioning virtual machines (VMs) with different operating systems (OS) are captured. The deployers can choose their desired OS images to spawn the VMs/containers.

The deployer can select a pre-defined VM flavor, available networks, security groups, roles, and the available images, all of which are defined as variabilities in our metamodel. They also must specify their environment file, the secret key for the selected cloud host types, which are the endpoints to bind to a particular cloud provider as shown in Figure 1. Optionally, a pre-deployed machine can be specified by providing the IP address and OS. Available services and VM types for cloud platforms are pre-defined in the metamodel.

2) Metamodel for Application Components: : For cloud-hosted services, CloudCAMP provides different node types for application components such as Web Application, Database Application, DataAnalytics Application, etc. For instance, the metamodel enables a deployer to choose the web server attribute, language for the code, the database server attribute or the NoSQL database attributes from the provided list. The deployer has to specify the variable attributes to deploy the desired application component type.

3) 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
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.
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.

4) Extensibility of the Metamodel: CloudCAMP is an opinionated framework; however, with lots of freedom. The metamodel has been designed for extensibility so that in future we can add more application node types. Adding a new application component is time-consuming; however, it is a one-time effort, and it is reusable. The CloudCAMP metamodels are extensible and reusable, so new component types and platforms can be added as required in the CloudCAMP metamodel.

C. CloudCAMP Knowledge Base Design

The Knowledge Base of CloudCAMP comprises a database and the application type templates.

1) Knowledge Base Database Design: The ER diagram of the knowledge base database is depicted in Figure 2(a), which shows 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 database. We store (1) all the operating systems, their distributions, package manager and versions in the *os_pkg_manager* table, (2) all available application component types, e.g., PHP based web application, MySQL based DB applications, etc. in the *swdependency* table, and (3) all the software packages needed for a particular application type are found using reverse engineering and stored in the *packages* table. We build the lookup table manually to handle these variability points. The sample section of the database table structure is shown in Figure 2(b).

2) Knowledge Base Template Design: The knowledge base templates are designed by capturing the commonality in the application components, and comprises placeholders which need to be filled up by the CloudCAMP DSML by querying the knowledge base database.

3) Extensibility of the Knowledge Base: The knowledge base is extensible by design. Addition of new application components requires the design of new templates (at least in part) by reverse engineering the software stack. The commonalities and variabilities need to be identified, and according to that,

[1]Due to space constraints, we do not show detailed screenshots of each metamodel. The interested reader can find these details in [14].

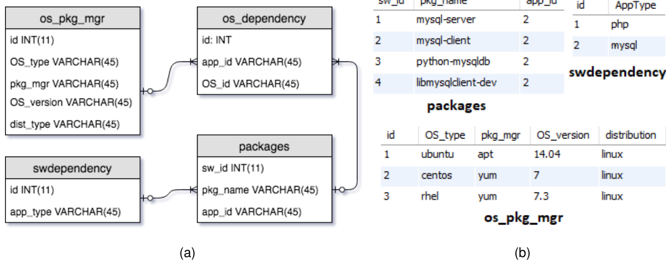

Fig. 2: (a) The Entity-Relation(ER) Diagram of CloudCAMP knowledge base and (b)Sample portion of KnowledgeBase Database tables

the template needs to be designed. The software dependencies for the application components required to be inserted in the knowledge base database tables.

D. Generative Capabilities of CloudCAMP DSML

We now present the transformation process of CloudCAMP that uses the DSML and Knowledge Base.

1) Knowledge Base for Generation of Infrastructure-as-code Solution for Deployment: CloudCAMP's generative capabilities (Requirement III-A2) 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 [line24-34] to populate the appModel [line17] based on the user's specifications. Then, the query result fills application-specific predefined configuration templates and generates IAC, e.g., Ansible, for specific application components [line 29-34] using template-based transformation. A similar approach is taken to configure the service-specific containers or to start the cloud-specific services.

2) Determining the Order of Deployment and Execution: The NodeJS script in CLoudCAMP 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 18-21]. All the 'HostedOn' dependent building blocks run in a linear fashion.

3) Generation of Infrastructure-as-code for Migration: For migration of application components on CloudCAMP, the 'deleteFrom' connection type specifies from where the user wants to move the application components and attaches a 'migrateTo' connection type to indicate the destination. The 'migrateTo' relation type cannot be defined without 'deleteFrom' connection type to ensure correctness of the model. The DSML will spawn a new VM with the new operating system for the 'migrateTo' destination node, and delete the current node afterward.

Algorithm 1: Deployment Script Generation

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

4) Support for Continuous Delivery: CloudCAMP can also handle continuous delivery and component addition/deletion (Requirement 3), 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 the same configuration in the target environment regardless of their current state.

5) Constraints checking for Correctness Business Models: We validate the business model by checking for constraint violations thereby ensuring that the models are well-formed and "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 IAC. We also verify other rule-based constraints to verify the components compatibility. For example, Amazon Kinesis delivery stream destination has to be Amazon Services (e.g., Redshift, S3); it cannot be Azure or OpenStack Services.

IV. Evaluating the Benefits of the CloudCAMP Platform

This section describes results comparing the time and effort incurred in deploying application use cases using (a) manual efforts, where the deployer 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.

A. Case Study 1: LAMP-based Service Deployment Study

Use Case: This is a prototypical three-tier Linux, Apache, MySQL, and PHP (LAMP)-based microservice architecture deployment similar to the motivating example described in Section I-A. Figure 1 shows the application topology illustrating the modeling effort in CloudCAMP.

This case-study and related user-study appear in our prior work [1]. Here, we describe the details of template-based transformation that happens behind the scenes within CloudCAMP DSML. As stated in Algorithm 1, the DSML traverses the business logic tree of Figure 1, which is defined by the deployer, and collects all the user-defined attributes as shown in Figure 3. It populates the pre-defined template for the specific application type with the user-defined attributes. The 'mysql_user' and 'mysql_root_pass' will be filled from specifications related to DBApplication type (Figure 3(b)).

Meta type	WebApplication
Attributes	
language	PHP
name	WebApplication
replication_count	1
src	https://github.com/Anirban2

(a)

Meta type	DBApplication
Attributes	
dbLocation	mySqlDB/movieDB.sql,myS
dbnames	moviedb,bookstore
name	DBApplication
password	admin
port	3306
replication_count	1
src	https://github.com/Anirban2
user	root

(b)

Fig. 3: (a) specifications related to WebApplication type and (b) specifications related to DBApplication type

The application components' software dependencies are gathered by querying the knowledge base database. For example, to install MySQL on a Ubuntu16.04 machine, the mysql-server and mysql-client software packages are needed. So, CloudCAMP DSML will query the knowledge base database and runs the template-based transformation to concretize the pre-defined partial template. The DSML copies the related configuration files in specific folders to configure MySQL correctly. Thus, the DSML will populate the pre-defined template file with all the details, and generate deployable Ansible-specific deployable IAC. After generating all the Ansible-specific files, the CloudCAMP executes these files in proper order to deploy the application by provisioning the cloud infrastructure as described in Algorithm 1.

1) Measure of Manual Effort:: We conducted a small user study in a Cloud Computing course for case study 1 involving sixteen teams of three students each. We requested users to manually 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. We have also requested them to write the ansible script to provision the same application stack and infrastructure. We measured the time taken, and efforts for (a) a fully manual effort, (b) for writing scripts in Ansible and executing these manually, and (c) using the CloudCAMP framework to deploy the scenario.

Quantitative Evaluation based on a User Study: The questionnaire as shown in Table I was created to conduct the study. For each question, the evaluation scale was 1–10 where one is the easiest and ten is the hardest.

TABLE I: Survey Questionnaire: For Q1–Q3, rate on a scale of (1-10)

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 4a, 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%.

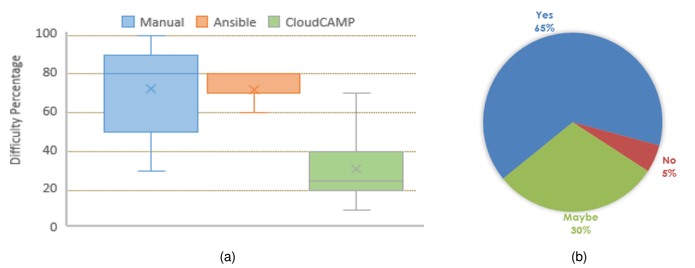

Fig. 4: (a) Comparing difficulty percentages to deploy services in different approaches, (b) Likeliness of using CloudCAMP for future cloud services deployment

Responses to Q4, Q5, and Q6: Time to complete the entire deployment:

The effort incurred by the user to deploy the LAMP model in the Cloud is shown in Table II [1], whereas using the CloudCAMP the same topology deployment time is approximately 15-20 minutes for the first time users.

TABLE II: 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 4b, 65% of the respondents agreed to use CloudCAMP tool to deploy cloud applications in the future, whereas 30% are still unsure.

Discussion: Results from our user study strengthen our belief that the CloudCAMP platform will be a very resourceful and productive 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. The visual drag and drop environment helps users to quickly deploy various scenarios of business application topology in distributed systems. Therefore, the benefits of automated provisioning accrued using CloudCAMP can easily be understood.

B. 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 III-A2, suppose the user wants to migrate the database application component from one machine to another machine, which resides on a different OpenStack cloud 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 section III-D3.

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 a median of 720 minutes as shown in Table II. 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 using manual effort for the migration process are 553 lines as per the survey is shown in Table II.

V. 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 service provisioning at a higher level of abstraction, and deploy its code without requiring significant domain expertise while requiring only minimal modeling effort and no low-level scripting. All the application components are the building blocks in our modeling environments and can be connected using exposed endpoints as a pipeline. The DSML will generate "correct-by-construction" IAC solution from the pipeline and execute the IAC to provision the application stack on the target cloud environment. Using WebGME to define the CloudCAMP framework enables us to decouple its metamodel(s) and knowledge base from the generative aspects while permitting extensibility. CloudCAMP significantly increases the productivity and efficiency of the application deployment and management team. 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] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "Cloudcamp: Automating the deployment and management of cloud services," in *International Conference on Services Computing (WIP)*. IEEE, 2018.

[2] J. Carrasco, J. Cubo, F. Durán, and E. Pimentel, "Bidimensional cross-cloud management with tosca and brooklyn," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016.

[3] H. Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Pattern-based deployment service for next generation clouds," in *Services (SERVICES), 2013 IEEE Ninth World Congress on*. IEEE, 2013, pp. 464–471.

[4] K. Képes, U. Breitenbücher, and F. Leymann, "The sepade system: Packaging entire xaas layers for automatically deploying and managing applications," *month*, 2017.

[5] D. Ardagna, E. Di Nitto, G. Casale, D. Petcu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Ballagny, F. D'Andria *et al.*, "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, 2012, pp. 50–56.

[6] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, 2008.

[7] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski, "Automatic deployment of services in the cloud with aeolus blender," in *Service-Oriented Computing*. Springer, 2015, pp. 397–411.

[8] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi, "Automated synthesis and deployment of cloud applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 211–222.

[9] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann *et al.*, "Automatic topology completion of tosca-based cloud applications." in *GI-Jahrestagung*, 2014, pp. 247–258.

[10] U. Breitenbucher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining declarative and imperative cloud application provisioning based on tosca," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 87–96.

[11] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Celar: automated application elasticity platform," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 23–25.

[12] OASIS, "Topology and orchestration specification for cloud applications," http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf, 2013, oASIS Standard.

[13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "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, 2013, pp. 130–148.

[14] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, "Cloudcamp: A model-driven generative approach for automating cloud application deployment and management," *Technical Report*, no. ISIS-17-105, sep 2017.