CloudsStorm: An Application-driven DevOps Framework for Managing Networked Infrastructures on Federated Clouds

Huan Zhou*‡, Cees de Laat* and Zhiming Zhao*
*Informatics Institute, University of Amsterdam, Amsterdam, Netherlands
‡School of Computer Science, National University of Defense Technology, Changsha, China
Email: {h.zhou, delaat, z.zhao}@uva.nl

***Abstract*—Most clouds provide dedicated virtual infrastructures to cloud applications with only limited programmability and controllability, which enlarges the management gap between infrastructures and applications. Traditional DevOps (development and operations) approaches are not suitable in today's cloud environments, because of the slow, manual and error-prone collaboration between developers and operations personnel. To address this issue, there have been a number of DevOps tools or frameworks proposed for cloud. However, most of them focus on infrastructures without considering the application requirements. It becomes even more difficult when managing infrastructures across multiple data centers or clouds. To mitigate this gap, we have designed CloudsStorm, an application-driven DevOps framework that allows the application directly program and control its infrastructure. In particular, it provides multi-level programmability and controllability according to the applications' specifications. We evaluate it by comparing its functionality to other proposed solutions. Moreover, we implement an extensible TSV-Engine, which is the core component of CloudsStorm for managing infrastructures' lifecycle. It is the first to be able to provision a networked infrastructure among public clouds. At last, we conduct a set of experiments on actual clouds and compare with other related DevOps tools. The experimental results demonstrate our solution is efficient and outperforms others.**

***Keywords*—application-driven; federal clouds; networked virtual infrastructure**

I. Introduction

The purpose of DevOps is to put application development and infrastructure runtime operations together to deliver good quality and reliable software. It encompasses continuous integration, test-driven development, build/deployment automation, etc. However, its main focus often is the application itself. Especially with the trend of migrating applications onto clouds, operations are not limited to fixed private servers any more. The virtual infrastructure provided by clouds is dynamic and on-demand, but often lacks sufficient programmability and controllability to fully satisfy complex application requirements. From the application perspective, cloud applications have become complex and large-scale, no longer being just simple web services. Some dynamic applications, like IoT (Internet of Things) applications, even need to control their underlying infrastructure during runtime, for instance to perform auto-scaling. Hence, it has become vital to fill the DevOps gap between the cloud application and its virtual infrastructure in order to better migrate more complex and demanding applications onto clouds.

Traditional DevOps approaches are slow, manual and error-prone, which are difficult to make the infrastructure suitable for the application. In order to settle this problem, some DevOps tools have been constructed to automate the provisioning and running of virtual infrastructure. For example AWS CloudFormation [1] provided by the Amazon Web Service cloud, is a useful tool to create and manage AWS resources, including automated provisioning and update. Nevertheless, it mainly works for web applications. The other limitation is that it is a vendor lock-in solution, which can only be used on Amazon EC2 infrastructure. There are also tools to manage the infrastructures from different clouds, avoiding vendor lock-in, such as Libcloud [2], jclouds [3] and fog [4].

All above tools mainly focus on the provisioning perspective, which consider the stage before applications running. They unify APIs of different clouds and provide some basic new APIs, which makes them API-centric. However, they do not provide functionality to manage the whole infrastructure's entire lifecycle. According to this, there are some environment-centric [5] tools to help developers orchestrate their applications. They include Puppet [6], Chef [7], Ansible [8], JuJu [9] and Nimbus [10]. They orchestrate the applications running on virtual infrastructures, but they more concentrate on deployment and configuration. In academic research, some DevOps architectures and systems are proposed, including CodeCloud [11], CloudPick [12] and CometCloud [13]. Some of them leverage the concept of managing "Infrastructure as Code" [14] during DevOps. The code here is preferred to be used for describing the infrastructure or configuration, such as Ansible using playbook to unify the configuration process on different machines. Some of these tools then interpret the code to do actual provisioning and configuring. However these static codes can hardly describe how the infrastructure dynamically adapts to the application. Moreover, these tools do not manage the networked infrastructure, especially if it crosses different data centers or clouds. In addition, some of them afford REST APIs and a centralized service to manage different infrastructures. This entails a single point of failure and a possible security vulnerability because of the leaking cloud credentials to a third-party broker.

In this paper, we propose an application-driven DevOps framework, CloudsStorm, to enhance the virtual infrastructure programmability and controllability. Contrary to "Infrastructure as Code", we put forward "Code as Infrastructures". Our "code" is not only used to describe but also can be directly executed. Moreover, we manage the federal cloud networked infrastructure through partitioning them into different sub-infrastructures. We make the following contributions.

1) CloudsStorm DevOps framework: We design the DevOps framework, CloudsStorm. It brings the infrastructure into the application development phase and enables the applications to directly program and control their virtual infrastructures instead of just describing what they desire. It also provides multiple levels of programmability and controllability according to developers' knowledge of infrastructure.

2) TSV-Engine: We implement a TSV-Engine to manage our partition-based infrastructures, that are distributed across different data centers or clouds. It is the key engine of CloudsStorm. It is also the first one to be able to provision a networked infrastructure among public clouds. With these three types of engine at different levels, there are four advantages: i) Fast. It can control several sub-infrastructures simultaneously to reduce the management overhead, such as provisioning, failure recovery and scaling, etc. ii) Extensible. It is easy for the application to derive its own engine to control the infrastructure in its private cluster. iii) Schedulable. TSV-Engine manages the infrastructure through different request queues, which makes the controlling processes schedulable. iv) Reliable. It partitions the entire infrastructure into small sub-infrastructures in multiple data centers. Even if some data center is down or not accessible, it may not influence the whole application.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 presents the DevOps framework and model of CloudsStorm. Then we describe the implementation of TSV-Engine in Section 4. Section 5 evaluates CloudsStorm from the aspect of performance. Finally, we conclude and discuss some application scenarios in Section 6.

II. Related Work

In order to mitigate the difficulty of virtual infrastructure maintenance for cloud applications, there have been substantial academic research and industrial tools developed in recent years. However many DevOps tools only focus on some specific steps in the infrastructure management and provide limited programmability. For example, Libcloud [2], jclouds [3] and fog [4] just unify several clouds' provisioning APIs. They automate the provisioning, but still need manual configuration and management on the infrastructure. Conversely, tools such as Puppet [6], Chef [7] and Ansible [8] all try to manage infrastructures by turning them into code. However, they are infrastructure-centric, focusing on configuration and installation. They standardize the configuration commands among different systems to make the code reusable, such as the cookbook of Chef and playbook of Ansible. The Nimbus [10] team develops a Context Broker and cloudinit.d to create the "One-Click Virtual Cluster". Furthermore, they offer scaling capability allowing users to automatically scale across multiple distributed clouds [15]. Juju [9] is application-centric which more focuses on application and services. CodeCloud [11] consists of a IM [16] (Infrastructure Manager). IM provides some specific REST APIs to control each individual VM (Virtual Machine). Based on this, CodeCloud leverages CJDL (Cloud Job Description Language) to describe the application and the elasticity of the infrastructure. CloudPick [12] is a system that considers the high-level constraints of the application on the infrastructure, including deadline and budget. However, these systems work as centralized services asking users to upload their cloud credentials, which requires trust in a third party. CometCloud [13] provides a heterogeneous cloud framework to deploy several programming models, such as master/worker, map/reduce and workflow. A video analytics system [17] is a notable application scenario of CometCloud. But CometCloud needs provisioned resources in advance to set up a cloud agent for each cloud.

Most of above tools mentioned support multiple clouds, meaning no vender lock-in for configuration and deployment; however this does not mean that they are capable of provisioning a federal infrastructure for running applications. On the other hand, none of above tools consider about provisioning a networked infrastructure, which the VMs can be inter-connected with end-to-end private connections. This is difficult to realize, especially in federated cloud environments. ExoGENI [18] [19] proposes Networked Infrastructure-as-a-Service (NIaaS) architecture based on SDN to permit the customization of network used by the infrastructure. SAVI [20] [21] builds up a test bed for IoT. It leverages OpenFlow to consider the network topology of the virtual infrastructure. However, all these are established on private data centers, which means the data centers in the federation must be totally under the control of the proposed solutions. For instance, these solutions must have direct access to the switch in the data center to control the network. Kraken [22] and SWMOA [23] also take network into account when provisioning over multiple private data centers or clouds. All of these solutions are not feasible to be applied on public clouds.

III. DevOps Framework and Model

In this section, we start by introducing the overview of the DevOps framework, CloudsStorm. The goal of this framework is to manage applications in the environment of federated clouds. Subsequently, we describe its core models in detail.

A. Framework Overview

Figure 1 illustrates the overview of the DevOps framework we propose. With this framework, cloud applications can achieve the goal of "Code as Infrastructures". It means that cloud application developers are not only able to develop their own applications but also able to program on the virtual infrastructures. The infrastructure management can be brought into the application development phase.

In this framework, there are three kinds of code, including infrastructure code, application code and runtime control policy. Cloud applications need only leverage these to control the whole lifecycle of the infrastructure their applications

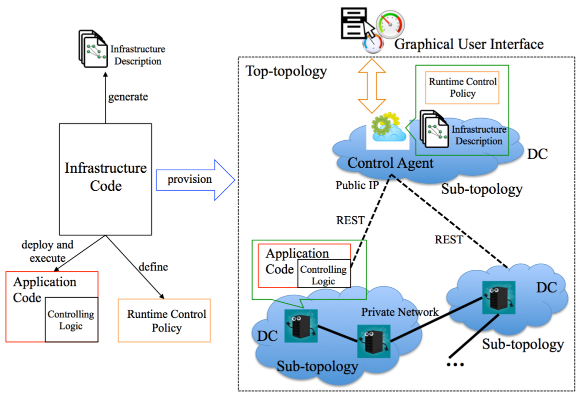


Fig. 1: Overview of CloudsStorm framework

rely on, including phases of provisioning, deployment, auto-scaling and destruction. Among these code types, the infrastructure code is the core of the framework. It first allows application developers to describe the infrastructure they wish, which also includes the network topology. Then it indicates which execution code should be running on which node of the infrastructure. The execution code is the part that the application developer should initially focus on. It is the main logic for the cloud application. The difference here is that the controlling logic can be embedded in the execution code to allow the application to directly control the infrastructure on demand. This is explained in Section III-B. At last, the infrastructure code defines the runtime control policy for dynamically managing the infrastructure in the runtime.

The networked infrastructure designed by the application developer can be automatically provisioned through executing the infrastructure code. The runtime environment for the cloud application is configured immediately afterwards. Then, the specified execution code is uploaded to the corresponding instance to execute. There is a control agent in the whole infrastructure. It can be explicitly assigned by the developer or generated by CloudsStorm in an autonomous way. It manages the infrastructure descriptions and has the whole view of the infrastructure. On one aspect, it affords REST APIs to interact with the execution code to receive control requests and combines the runtime control policy defined by the application to control the infrastructure, including failure recovery and auto-scaling, etc. On the other hand, it provides a graphical user interface for developers to manually check the status of their infrastructure and control it in a visual way. All the outward connections of this control agent use public IP addresses. For federated clouds, we propose partition-based infrastructure management. The whole topology of the infrastructure is called the “top-topology”. It is partitioned into small pieces of infrastructure, each referred to as a “sub-topology”. Each sub-topology belongs to a data center domain of a cloud provider. It describes how the nodes are connected in one data center and the top-topology describes how the sub-topologies are connected. It is worth mentioning that all the nodes are connected with private network IP addresses. The implementation technique and advantage of this private networked infrastructure are demonstrated in following sections.

B. Runtime Controlling Model

After executing the infrastructure code, all the infrastructures are provisioned and different components of the application run on the desired nodes. The infrastructure description is generated by the infrastructure code and uploaded to the control agent. Then control agent takes over the responsibility to manage the infrastructure. Here, the control agent is placed in a separate $subNI$ and its public IP is configured into all the other nodes for communication.

Figure 2 illustrates the sequence diagram of runtime controlling model. It consists of three controlling scenarios. In the failure recovery scenario, we assume that there is an auto failure recovery mechanism provided by cloud provider for each individual node. Therefore, we more focus on the failure that the data center is down or the network to data center is not accessible. The control agent detects the availability of each $subNI$. If it is not available, the control agent sets up a new $subNI$ from another data center DN of cloud CN to replace the failed one according to the runtime control policy RCP. Meanwhile, the private network topology among the infrastructures is preserved. In the auto-scaling scenario, there are two different modes for the application to control the infrastructure. One is an active mode, which means the application actively controls the infrastructure. This is the responsibility of the controlling logic embedded in the application code shown in Figure 1. It queries infrastructure information from the control agent, for example how many nodes in this data center. Then according to input conditions such as the input data size, the application decides whether to scale up or down and sends the request to the control agent. After receiving the request, the control agent takes the application-defined RCP into account, for instance budget, and finally takes control of the infrastructures to scale up or down by the calculated number num. The other mode is passive mode. The infrastructures are passively controlled by the control agent based on RCP and the infrastructure’s system information.

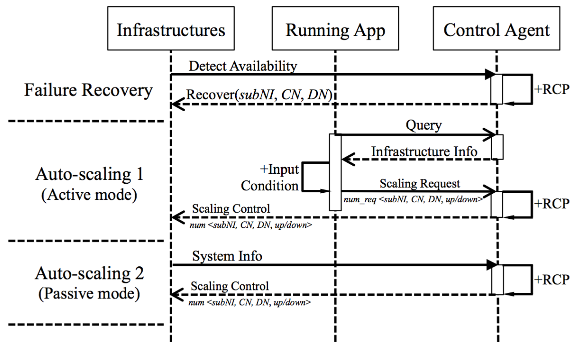


Fig. 2: Sequence Diagram of Runtime Controlling Model

In this model, the application can invoke REST APIs to communicate with the control agent. The application also defines RCP, which is human readable and in YAML format. The detailed content of RCP is not expressed in this paper.

IV. Implementation

To demonstrate this framework, we implement a prototype of CloudsStorm. In this section, we introduce the fundamental techniques to realize the core component of CloudsStorm, TSV-Engine, which is a partition-based infrastructure control engine. It is realized in Java and managed on GitHub. Then we describe the infrastructure lifecycle management based on TSV-Engine. In addition, TSV-Engine is also responsible for connecting the VMs to provision a networked infrastructure. To implement this, we adopt an IP tunnel technique to connect the VMs, proposed by our previous work [24].

TSV-Engine is the core engine of CloudsStorm, which is the elementary engine to manage the infrastructure lifecycle. "TSV" is short for "Top-topology", "Sub-topology" and "VM" as shown in Figure 3. A T-Engine is responsible for "Top-topology" management. In CloudsStorm, every application has one T-Engine. It helps the application to control the whole infrastructure and manage the connections among sub-topologies. During runtime, the application can generate different requests to dynamically change the infrastructure, including provisioning, recovering, scaling, deleting or stopping. The T-Engine takes these requests and queries the user database to set up a specified S-Engine for each cloud, for example, "S-Engine-EC2" for cloud "EC2". Meanwhile, the T-Engine queries the user credential to configure the S-Engine with a proper credential, which makes the S-Engine able to access that cloud. Then, the S-Engine manages each individual VM and its connections via the specified V-Engine. The V-Engine is responsible for the VM lifecycle from creation to stopping or deleting VMs. It also controls the connection between the VM and other VMs. These connections are based on the IP tunnel mechanism proposed by our previous work [24] to connect the VMs in different federated clouds with private network. After provisioning, the V-Engine can run the application-defined script to configure the runtime environment and deploy the application. The V-Engine is a basic engine. Different customized V-Engines can be derived from it depending on the VM's features, such as "V-Engine-ubuntu" for ubuntu VM, etc. If the application has specific operations on some VM, it can also customize its own V-Engine.

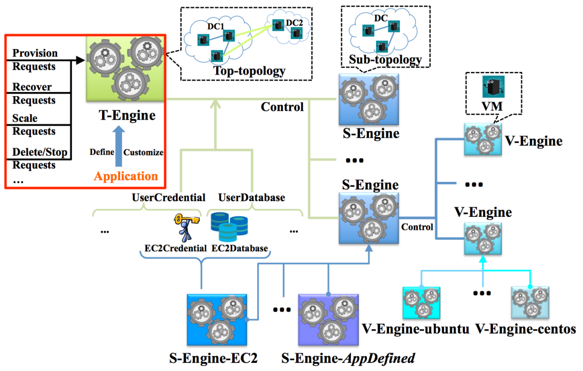


Fig. 3: Architecture of TSV-Engine

In addition, all the S-Engines and V-Engines are running in multi-thread. It means that the T-Engine can start several S-Engines at the same time. If these sub-topologies managed by these S-Engines belong to different data centers, there will no conflict among them. Then they can totally run in parallel. As the V-Engine running in a thread, the creation, configuration, deployment of all the VMs in one sub-topology can proceed simultaneously. In other words, TSV-Engine accelerates the controlling process. Other advantages of CloudsStorm based on TSV-Engine is analyzed in Section V.

V. Evaluation

In this section, we compare CloudsStorm with other related DevOps tools and evaluate it from two aspects, functionality and performance.

A. Performance Evaluation

To evaluate the performance, we first analyze the reliability of CloudsStorm's infrastructures and then evaluate its controllability.

1) Reliability: We assume the infrastructure reliability in one data center is r, which $0 < r < 1$. The entire infrastructure reliability NI_{con} set up by CloudsStorm is r_{con}. NI_{con} distributes the infrastructures among k data centers. Accordingly their reliabilities are $r_0, r_1, ..., r_{k-1}$. As the infrastructure reliability depends on the data center's available time, the failure possibility of a infrastructure in that data center is $1-r_j$, $\forall j \in [0,k)$. Considering these data centers are independent, the failure possibility of entire NI_{con} is $\prod_{i=0}^{k-1}(1-r_i)$. We assume that most infrastructures are replicated parts. For example, there are many replicated slaves in a "Master/Slave" framework. If some of them fails, the application can still run. Therefore, $r_{con} = (1-\prod_{i=0}^{k-1}(1-r_i))$. Obviously, $\prod_{i=0}^{k-1}(1-r_i) < (1-r_j)$, $\forall j \in [0,k)$. Then we get Equation 1.

$$r_{con} = (1 - \prod_{i=0}^{k-1}(1-r_i)) > r_j, \forall j \in [0,k) \tag{1}$$

It demonstrates that the infrastructures provisioned by CloudsStorm has higher reliability than the reliability of the entire infrastructure only in one data center.

2) Auto-scaling and failure recovery: These two are the key controllability of CloudsStorm. We design the experiment on ExoGENI to test the auto-scaling performance. In this experiment, there are two sub-topologies in the beginning, $subNI_1$ containing 1 VM and $subNI_2$ containing 8 "XOMedium" VMs. Each VM in $subNI_2$ is connected with the VM in $subNI_1$ via a private network link. This is a typical "Master/Slave" distributed framework. $subNI_2$ is defined as a scaling group. According to the scaling request, the infrastructure can scale up to other data centers based on one or multiple copies of $subNI_2$. At the same time, all the network links between the scaled copies and $subNI_1$ are connected. These connections leverage private addresses, which can be defined before actual provisioning. Hence, the "Master" in $subNI_1$ can always know where are the scaled resources are. Figure 4 illustrates that we scale up the 8 VMs of $subNI_2$ accordingly at 1, 2, 3, 4, 8 and 16 times. Each scaled $subNI_2$ is provisioned in independent data centers simultaneously. Therefore, the flat dashed line is the ideal scaling performance in theory. However, the provisioning performances of different data centers are not the same. This is demonstrated by the varied dashed line, which is the average value of the maximum provisioning overhead among the scaled $subNI_2$. Moreover, the end-to-end connections need to be set up. The more copies of $subNI_2$ requested, the more connections need to be configured. The solid line in the figure shows total cost. For each scale, we conduct 10 repeated experiments. The error bar denotes the standard deviation. It demonstrates that the scaling overhead does not grow at the same proportion as the number of VMs being created. Therefore, it is able to complete large-scale auto-scaling in a short time. In addition, most clouds have limitations on the resource allocation. For instance, ExoGENI only allow one user to apply a maximum of 10 VMs from one data center. The limitation for EC2 is 20. Nevertheless, with CloudsStorm, we can break through these limits to realize large-scale scaling by combining resources from different data centers and even clouds.

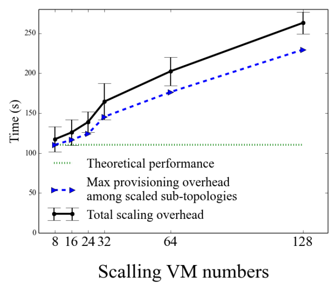


Fig. 4: Auto-scaling performance

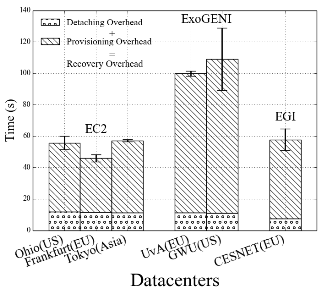
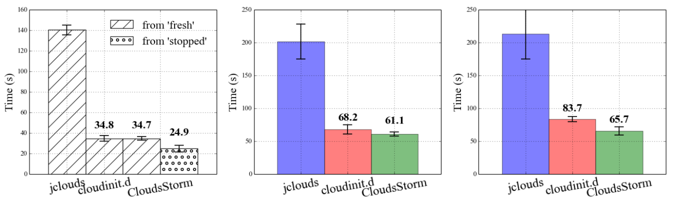


Fig. 5: Failure recovery performance

Figure 5 shows the experimental result on failure recovery. In this experiment, there are still two sub-topologies in the beginning, $subNI_1$ and $subNI_2$. Each of them contains only one VM, n_1 and n_2. These two nodes are connected with the private network. Then we simulate the case where the data center of $subNI_2$ is not available. CloudsStorm recovers the same sub-topology from another data center or cloud. Finally, the private network is resumed. Hence, the application is not aware of this infrastructure modification. We get the detaching overhead from CloudsStorm, which is the time for $subNI_1$ to disconnect the original link. It is illustrated by the bar covered with dots. On the other aspect, we continually test the private link from n_1 of $subNI_1$ to n_2 of $subNI_2$ and record the time from lost connection to the time that the link is resumed. This is the total recovery overhead. We conduct this experiment on 3 clouds currently supported and pick 6 data centers from them. In order to compare, n_2 always has 2 cores and around 8G memory with "Ubuntu 14.04". Correspondingly, they are "t2.large" of EC2, "XOLarge" of ExoGENI and "mem_medium" of EGI. The results show ExoGENI has a relative higher recovery overhead and some of its data centers are not stable. The performance of EC2 and EGI are close, however, most data centers of EC2 are more stable. These information are important to decide where to recover to satisfy the application QoS, considering about the recovery overhead and data center geographic information.

3) Comparison: Finally, we conduct a set of experiments to compare CloudsStorm with other DevOps tools. We pick jclouds from API-centric tools. It is adopted by a lot of environment-centric tools to be the basic provisioning engine, such as CloudPick [12]. From environment-centric tools, we pick Nimbus team's cloudinit.d. Other tools like Juju and IM provide graphical interfaces, which makes it difficult to measure performance. Both of jclouds and cloudinti.d do not support networked infrastructure. The ones which support networked infrastructure can only be applied in private data centers, which CloudsStorm cannot have the access permission, like SAVI. We pick EC2 to do these experiments, because this is the most popular cloud provider and commonly supported by these tools. First, we compare the scaling performance. The scaling request is to add 5 more "t2.micro" VMs in EC2 California data center. However, jclouds and cloudinit.d cannot directly support auto-scaling behavior, we use them to provision 5 new VMs in California data center to simulate this scenario. For each operation, we repeated 10 times. Figure 6(a) illustrates the results. For jclouds, the provisioning process proceeds in sequence, hence, its scaling overhead is much more bigger than the other two. If only considering the scaling performance from initial state, cloudinit.d and CloudsStorm have similar performance, demonstrated by the bars covered with slashes. CloudsStorm is a little bit stable than cloudinit.d. Moreover, EC2 supports stopping a instance. CloudsStorm can perform auto-scaling from "Stopped" status. It reduces the overhead, shown by the bars covered with dots. It is worth to mention that we do not consider deployment overhead in this experiment. Scaling from "Stopped" status can even omit the deployment. Through this way, CloudsStorm outperforms cloudinit.d much better, reducing the scaling overhead by more than half referring to Figure 6(b)(c).

The second experiment is to compare the provisioning performance including deployments. All of these three allow users to define a script to deploy applications immediately after provisioning. In this experiment, we choose California data center to provision 5 "t2.micro" VMs and install tomcat on each of them. Each test is repeated 10 times. Figure 6(b) shows the results. With jclouds, the application are installed one by one, which costs plenty of time. For CloudsStorm, there is a V-Engine responsible for each individual VM to provision and deploy. Therefore, it achieves the best performance according to the overhead and stability. The last experiment is based on the second experiment considering about the deployment dependency. In this experiment, 4 out of 5 VMs install tomcat



(a) Scaling (b) Including Deployment (c) Deployment with Dependency

Fig. 6: Performance comparison

and the remaining one installs a mysql database. In this case, there is a dependency when using jclouds and cloudinit.d, because they do not provision networked infrastructure and use public addresses to communicate. Tomcat can only be deployed after provisioning mysql VM to know the server address. Hence, jclouds needs to provision mysql VM first in its sequence. Cloudinit.d defines different levels to realize the dependency. In this scenario, the first level is the mysql VM and the second level contains four tomcat VMs. The difference for CloudsStorm is that it can provision networked infrastructure. The nodes are connected with application-defined private network links. The mysql server address is pre-defined before actual provisioning. Therefore, all the deployments can proceed simultaneously even with the dependency. Figure 6(c) demonstrates that the deployment dependency makes smaller influence on CloudsStorm's performance comparing to that on jclouds and cloudinit.d. We can reason out that if there are more dependencies, CloudsStorm has a greater advantage over others.

VI. Discussion

This paper presents an application-driven DevOps framework, CloudsStorm, and implementation of TSV-Engine. In framework design, we propose "Code as Infrastructures" instead of managing the infrastructure as code. We make the infrastructure to be part of the application. With its multi-level programmability and controllability, the application developer can design its infrastructure along with the application development. Though executing the codes, the application can run on clouds autonomously without pre-existing infrastructures. In implementation, we propose partition-based infrastructure management and totally develop new engines, instead of integrating other DevOps tools. For example, CodeCloud [11] uses Puppet [6] and CloudPick [12] leverages jclouds [3]. Not only it is the first one to realize networked infrastructure on public federated clouds but also its performance is demonstrated to outperform other tools.

There are plenty of application scenarios for CloudsStorm. To benefit from the networked infrastructure, a lot of programming models are easy to deploy, such as Master/Slave (including docker cluster), Map/Reduce and workflow. To benefit from the sufficient and efficient controllability, it is useful for IoT applications and sensor-cloud. To benefit from the application-driven design, we can bring the infrastructure into the application incremental development phase. For example, testing the application with several VMs in the beginning and final release running on a large-scale infrastructure. Therefore, CloudsStorm narrows the DevOps gap.

References

[1] AWS CloudFormation. http://aws.amazon.com/es/cloudformation/
[2] Apache Libcloud. http://libcloud.apache.org
[3] Apache jclouds. https://jclouds.apache.org
[4] Fog. http://fog.io
[5] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining devops automation for cloud applications using tosca as standardized metamodel," *FGCS*, vol. 56, pp. 317–332, 2016.
[6] Puppet. https://puppet.com/
[7] CHEF. https://www.chef.io
[8] Ansible. https://www.ansible.com/
[9] Juju. https://jujucharms.com/
[10] K. Keahey and T. Freeman, "Contextualization: Providing one-click virtual clusters," in *eScience'08. IEEE Fourth International Conference on*, 2008, pp. 301–308.
[11] M. Caballer, C. de Alfonso, G. Moltó, E. Romero, I. Blanquer, and A. García, "Codecloud: A platform to enable execution of programming models on the clouds," *J. Systems and Software*, vol. 93, pp. 187–198, 2014.
[12] A. V. Dastjerdi, S. K. Garg, O. F. Rana, and R. Buyya, "Cloudpick: a framework for qos-aware and ontology-based service deployment across clouds," *Software: Practice and Experience*, vol. 45, pp. 197–231, 2015.
[13] J. Diaz-Montes, M. AbdelBaky, M. Zou, and M. Parashar, "Cometcloud: Enabling software-defined federations for end-to-end application workflows," *IEEE Internet Computing*, vol. 19, no. 1, pp. 69–73, 2015.
[14] K. Morris, *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.
[15] P. Marshall, H. M. Tufo, K. Keahey, D. La Bissoniere, and M. Woitaszek, "Architecting a large-scale elastic environment-recontextualization and adaptive cloud services for scientific computing." in *ICSOFT*, 2012.
[16] M. Caballer, I. Blanquer, G. Moltó, and C. de Alfonso, "Dynamic management of virtual infrastructures." *J. Grid Comput.*, vol. 13, no. 1, pp. 53–70, 2015.
[17] A. R. Zamani, M. Zou, J. Diaz-Montes, I. Petri, O. Rana, A. Anjum, and M. Parashar, "Deadline constrained video analysis via in-transit computational environments," *IEEE Trans on Services Computing*, 2017.
[18] Y. Xin, I. Baldine, A. Mandal, C. Heermann, J. Chase, and A. Yumerefendi, "Embedding virtual topologies in networked clouds," in *Proceedings of the 6th International Conference on Future Internet Technologies*. ACM, 2011, pp. 26–29.
[19] I. Baldin, J. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, and J. Mills, "Exogeni: a multi-domain infrastructure-as-a-service testbed," in *The GENI Book*. Springer, 2016, pp. 279–315.
[20] J.-M. Kang, T. Lin, H. Bannazadeh, and A. Leon-Garcia, "Software-defined infrastructure and the savi testbed," in *International Conference on Testbeds and Research Infrastructures*. Springer, 2014, pp. 3–13.
[21] J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia, "Savi testbed: Control and management of converged virtual ict resources," in *Integrated Network Management, IFIP/IEEE International Symposium on*, 2013, pp. 664–667.
[22] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Online and elastic resource reservations for multi-tenant datacenters," in *Computer Communications (INFOCOM), IEEE International Conference on*, 2016.
[23] W. Shi, C. Wu, and Z. Li, "An online mechanism for dynamic virtual cluster provisioning in geo-distributed clouds," in *Computer Communications (INFOCOM), IEEE International Conference on*, 2016.
[24] H. Zhou, J. Wang, Y. Hu, J. Su, P. Martin, C. De Laat, and Z. Zhao, "Fast resource co-provisioning for time critical applications based on networked infrastructures," in *Cloud Computing (CLOUD), IEEE International Conference on*, 2016, pp. 802–805.