CloudsStorm: A framework for seamlessly programming and controlling virtual infrastructure functions during the DevOps lifecycle of cloud applications

Zhou, H.; Hu, Y.; Ouyang, X.; Su, J.; Koulouzis, S.; de Laat, C.; Zhao, Z.

DOI
10.1002/spe.2741
Publication date
2019
Document Version
Final published version
Published in
Software, practice & experience
License
CC BY

Received: 26 November 2018 | Revised: 23 June 2019 | Accepted: 8 July 2019
DOI: 10.1002/spe.2741

EXTENDED CONFERENCE PAPER

WILEY

CloudsStorm: A framework for seamlessly programming and controlling virtual infrastructure functions during the DevOps lifecycle of cloud applications

Huan Zhou[1,2] | **Yang Hu**[1] | **Xue Ouyang**[2] | **Jinshu Su**[2] | **Spiros Koulouzis**[1] | **Cees de Laat**[1] | **Zhiming Zhao**[1]

[1]Informatics Institute, Science Park, University of Amsterdam, Amsterdam, The Netherlands

[2]School of Computer Science, National University of Defense Technology, Changsha, China

Correspondence
Zhiming Zhao, Informatics Institute, Science Park, University of Amsterdam, 1098 XH Amsterdam, The Netherlands.
Email: z.zhao@uva.nl

Present Address
Zhiming Zhao, Informatics Institute, Science Park 904, University of Amsterdam, 1098 XH Amsterdam, The Netherlands

Funding information
H2020 European Institute of Innovation and Technology, Grant/Award Number: 643963, 654182 and 676247

Summary

The infrastructure-as-a-service (IaaS) model of cloud computing provides virtual infrastructure functions (VIFs), which allow application developers to flexibly provision suitable virtual machines' (VM) types and locations, and even configure the network connection for each VM. Because of the pay-as-you-go business model, IaaS provides an elastic way to operate applications on demand. However, in current cloud applications DevOps (software development and operations) lifecycle, the VM provisioning steps mainly rely on manually leveraging these VIFs. Moreover, these functions cannot be programmatically embedded into the application logic to control the infrastructure at runtime. Especially, the vendor lock-in issue, which different clouds provide different VIFs, also enlarges this gap between the cloud infrastructure management and application operation. To mitigate this gap, we designed and implemented a framework, CloudsStorm, which enables developers to easily leverage VIFs of different clouds and program them into their cloud applications. To be specific, CloudsStorm empowers applications with infrastructure programmability at design-level, infrastructure-level, and application-level. CloudsStorm also provides two infrastructure controlling modes, ie, active and passive mode, for applications at runtime. Besides, case studies about operating task-based and big data applications on clouds show that the monetary cost is significantly reduced through the seamless and on-demand infrastructure management provided by CloudsStorm. Finally, the scaling and recovery operation evaluations of CloudsStorm are performed to show its controlling performance. Compared with other tools, ie, "jcloud" and "cloudinit.d", the scaling and provisioning performance evaluations demonstrate that CloudsStorm can achieve at least 10% efficiency improvement in our experiment settings.

KEYWORDS

DevOps, federated clouds, infrastructure-as-a-service, networked virtual infrastructure

1 | INTRODUCTION

DevOps puts application development and infrastructure runtime operation together to deliver good quality and reliable software. It encompasses continuous integration, test-driven development, build/deployment automation, and continuous delivery.[1] Traditional DevOps approaches are slow, manual, and error-prone, however.[1] These approaches more focus on the application itself and treat its underlying infrastructure as a preset and constant cluster, mainly because they are faced with fixed physical infrastructure without any programmability. There might even be two separate teams responsible for software development and infrastructure operation respectively, which decreases the development efficiency and software quality. The virtual infrastructure programmability and controllability provided by clouds make it possible to mitigate the gap between the application and infrastructure, and involve the infrastructure operation into the software DevOps lifecycle more intimately.

From the infrastructure perspective, the virtual infrastructure provided by cloud computing is different from traditional physical infrastructure. Cloud computing, especially the infrastructure-as-a-service (IaaS) model, enables the programmability needed to customize the computing resources, ie, virtual machines (VMs). Not only the capacity and type of the computing resources can be planned and programmed,[2] but also their locations, ie, the chosen clouds and data centers used.[3] Moreover, provided network resources can also be customized.[4] These are virtual infrastructure functions (VIFs) given by the clouds with the IaaS model and usually in the form of representational state transfer (REST) application programming interface (API) to invoke and control the resources remotely. Overall, most of these clouds provide the basic VIF of provisioning or terminating an individual VM. However, current programmability and controllability of the entire infrastructure given by clouds are limited, where most of the programmed infrastructure customization and operations are performed manually. Moreover, these VIFs cannot be easily programmed to perform high-level operations, such as autoscaling and failure recovery. They also cannot automate the network resource configuration, especially for operating infrastructures across data centers. When dealing with federated cloud infrastructure, different VIFs of different clouds still suffer the vendor lock-in problem because of different API definitions.

From the application perspective, cloud applications have become complex and large-scale, no longer being just simple web services. Most of these applications rely on an underlying platform to manage resources and schedule the tasks, such as, Hadoop[*] (a distributed platform to perform MapReduce tasks for data processing), Kubernetes[†] (an emerging platform for managing a container cluster), Hyperledger Fabric[‡][5] (a permissioned blockchain platform for smart contract execution), etc. These platforms are usually operated on the top of a private cluster consisting of VMs or provided by some cloud providers in the manner of platform-as-a-service. However, when migrating these platforms and their applications onto IaaS clouds, there comes an issue about automation of the VM provisioning process, to set up the platform and deploy the application from scratch. Therefore, we take the Hadoop platform application as an example to show the current gap between cloud applications and virtual infrastructures. Moreover, we identify that there are three levels of programmability at the application development phase and two types of controlling modes at the operation phase are required to mitigate the gap. It is worth mentioning that these programmability and controllability are not only required by this type of applications.

Programmability Issue: When developing the cloud applications, three levels of infrastructure programmability are required. **(1) Design-level.** Hadoop applications need design-level programmability to describe the required underlying infrastructure to specify the computing resources' (VM) quantity, types, locations, and network, etc. Current topology and orchestration specification for cloud applications[§] (TOSCA) standard is not sufficient to define especially the location (ie, the cloud and data center) and the network topology of the VM cluster, because TOSCA more focuses on the application components specification, instead of the detailed cloud virtual infrastructure description. **(2) Infrastructure-level.** Hadoop applications require infrastructure-level programmability to automate the process of provisioning the VMs, configuring the network, and deploying the platform. The infrastructure-level programmability should provide high-level operations (eg, provisioning, scaling, and failure recovery, etc) on the infrastructure. Meanwhile, the parallelism of the operation should be easily specified to achieve controlling efficiency. There are also tools trying to automate this process, eg, jclouds.[¶] However, these API-centric[1] tools focus on the programmability and controllability on

[*] https://hadoop.apache.org/
[†] https://kubernetes.io/
[‡] https://www.hyperledger.org/projects/fabric
[§] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#overview
[¶] https://jclouds.apache.org/

each individual VM instead of the entire infrastructure topology. In comparison, there are also some environment-centric[1] tools to help developers orchestrate their applications, which include Puppet,[#] Chef,[‖] etc. These tools more focus on the cluster management, ie, deployment and configuration, instead of provisioning and scaling automation. Some academic research studies also propose architectures for developing and orchestrating applications on clouds, eg, CometCloud[6] and mOSAIC.[7] However, most of these architectures themselves are platforms, which require manually to set up the cluster in advance and lack the ability of provisioning VM resources. **(3) Application-level.** Hadoop applications require application-level programmability to directly adjust the infrastructure to fit for the application constraints or workload. Because of the pay-as-you-go business model, the overprovisioned infrastructure generates an extra monetary cost. For example, a traditional Hadoop platform is based on a fixed cluster. When there is no input data to process, the extra VM resources waste money if it is deployed on an IaaS cloud. Hence, the Hadoop application should be able to program and customize the number of VM resources required, according to the input workload, eg, the data size. During runtime, the Hadoop application can dynamically adjust the infrastructure resources according to the input data size, ie, scaling out or in VMs, to reduce monetary cost. However, to the best of our knowledge, none of the current DevOps tools supports to embed this type of infrastructure programming logic directly at the application level.

Controllability Issue: During runtime, applications require two types of controlling modes to manage the infrastructure. **(1) Passive mode.** The application should be able to perform some infrastructure operations when meeting certain predefined thresholds, eg, CPU or memory utilization, or even failure conditions. For instance, once one of the VMs is crashed for hosting the Hadoop applications, another VM should be recovered from a certain predefined data center and rejoin the cluster to maintain the application quality of service (QoS). CloudWatch[**] provides this type of controlling mode. However, it is a vendor lock-in solution only for Amazon Elastic Compute Cloud (EC2). **(2) Active mode.** In order to satisfy the application QoS requirements, applications also need to actively control their infrastructure. For instance, the Hadoop application requires to deploy computing resources close the data source to reduce transferring cost. Especially, in the emerging fog/edge computing domain, the internet of things (IoT) applications even need to control their cloud resources close to the edge nodes on demand according to the dynamic distribution of sensors.[8] The implementation of this type of controlling mode relies on the application-level programmability support mentioned above. It is therefore still a gap.

On the other hand, the programmability of the individual VM introduced by IaaS clouds also brings in opportunities for the application to enhance the controllability of the infrastructure. Because although different IaaS clouds provide different forms of VIF, they all have to afford a basic VIF, ie, provisioning or terminating a VM, according to the IaaS model. Hence, we have designed and implemented CloudsStorm[††] framework to leverage these basic VIFs to construct high-level operations and enhance the infrastructure programmability and controllability for the application. According to the main components developed by CloudsStorm, we conclude our main contributions as follows: (1) a YAML Ain't Markup Language (YAML)–based domain specific language, "*Infrastructure Description Code,*" is designed to address the design-level programmability; (2) a YAML-based "*Infrastructure Execution Code*" is designed to address the infrastructure-level programmability; (3) "*Infrastructure Embedded Code*" based on some general-purpose programming languages is implemented to address the application-level programmability and perform active control; (4) a YAML-based "*Runtime Control Policy*" is designed to address the passive controlling mode; 5) an "*Infrastructure Execution Engine*" is developed to interpret aforementioned codes and leveraged by the "*Control Agent*" to perform infrastructure operations during runtime. Besides, the second case study of this paper reveals the practice and experience of leveraging CloudsStorm to mitigate the gap for the example Hadoop application mentioned above.

The rest of this paper is organized as follows. Section 2 presents the overview of an CloudsStorm framework and some related models. We then describe some implementation details to explain how the enhanced programmability and efficient controllability can be achieved in Section 3. Section 4 then first describes two case studies to demonstrate how different levels of programmability provided by the CloudsStorm framework can be leveraged to mitigate the DevOps gap for cloud applications. Afterward, a set of experiments are conducted on actual clouds to evaluate controllability performance and compared with other related tools. Section 5 introduces the related work. Finally, the conclusion is given in Section 6.

[#] https://puppet.com/
[‖] https://www.chef.io/
[**] https://docs.aws.amazon.com/cloudwatch
[††] https://github.com/zh9314/CloudsStorm

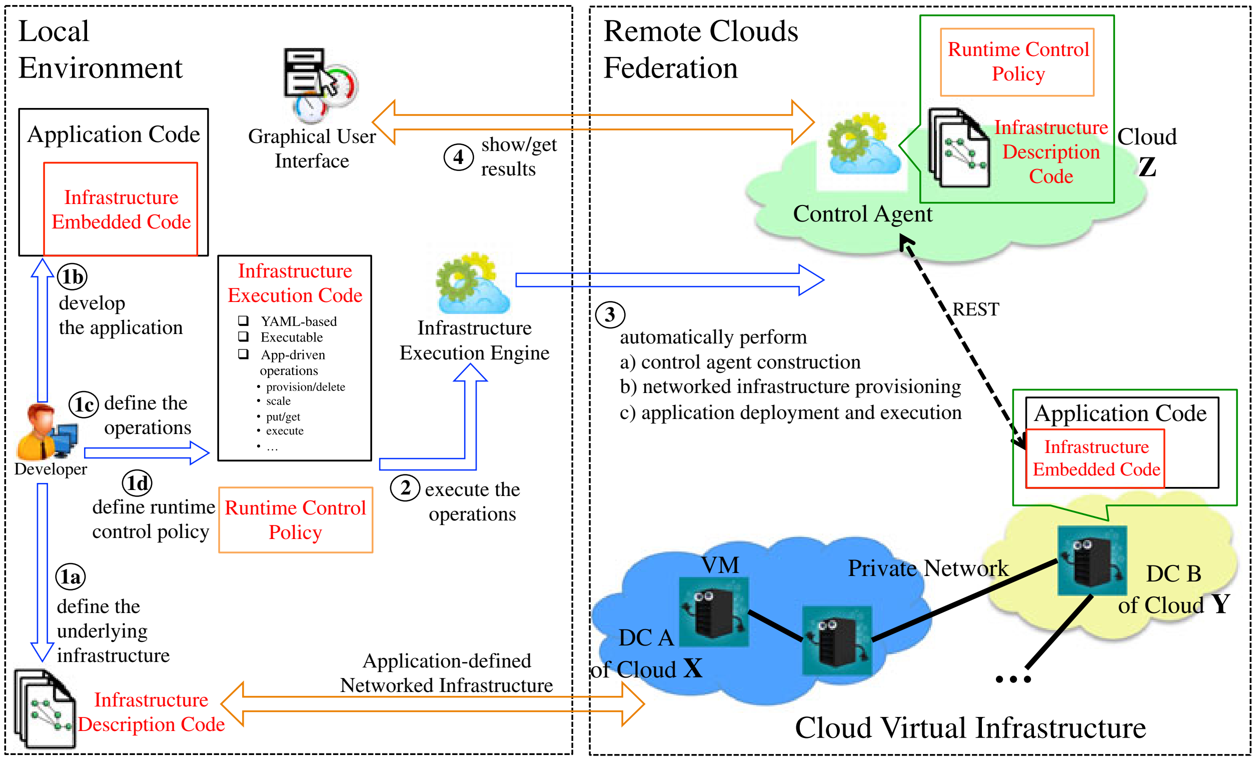

Local Environment
Application Code
Infrastructure Embedded Code
Graphical User Interface
4 show/get results
Remote Clouds Federation
Runtime Control Policy
Infrastructure Description Code
Cloud Z
Control Agent
1b develop the application
Infrastructure Execution Code
YAML-based Executable
App-driven operations
provision/delete
scale
put/get
execute
...
Infrastructure Execution Engine
3 automatically perform
a) control agent construction
b) networked infrastructure provisioning
c) application deployment and execution
REST
1c define the operations
Developer
1d define runtime control policy
Runtime Control Policy
2 execute the operations
Application Code
Infrastructure Embedded Code
1a define the underlying infrastructure
VM
Private Network
DC B of Cloud Y
DC A of Cloud X
Application-defined Networked Infrastructure
Infrastructure Description Code
Cloud Virtual Infrastructure

FIGURE 1 Overview of the CloudsStorm framework. REST, representational state transfer; VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

2 | THE CLOUDSSTORM FRAMEWORK AND MODELS

In this section, we start by introducing an overview of the CloudsStorm framework. The goal of this framework is to enhance cloud virtual infrastructure programmability and controllability when developing and managing cloud applications within the DevOps lifecycle. Subsequently, we describe its core models in detail.

2.1 | Framework overview

Figure 1 illustrates the CloudsStorm framework we propose. Through using the CloudsStorm framework, cloud application developers are not only able to develop their own application logic focusing on the software aspect, but are also able to program the virtual infrastructures focusing on the virtualized hardware aspect. Management on the underlying infrastructure in the operation phase can be programmed in advance and therefore brought into the application development phase. We still take the Hadoop application as an example to show the general process of using the CloudsStorm framework. The detailed demonstration of this example is shown as case study 2 in Section 4.2. The DevOps lifecycle with CloudsStorm is as follows.

1. For the first step in the development phase, CloudsStorm provides four types of code for infrastructure programming distinct from the original application programming code itself. The detailed syntax of these code is defined in Section 3.

   (a) The developer defines the underlying infrastructure topology through "*Infrastructure Description Code.*" In the example case, the developer defines that there are two VMs from "ExoGENI[‡‡]" cloud and "UvA" data center. One is configured to deploy a Hadoop platform, and the other is configured to emulate the data source for the experiment purpose. Meanwhile, these two VMs are defined to be connected within the private subnet "192.168.88.0/24". The benefit of the capability to define the network with a private IP address is discussed in Section 3.1.

[‡‡]http://www.exogeni.net/

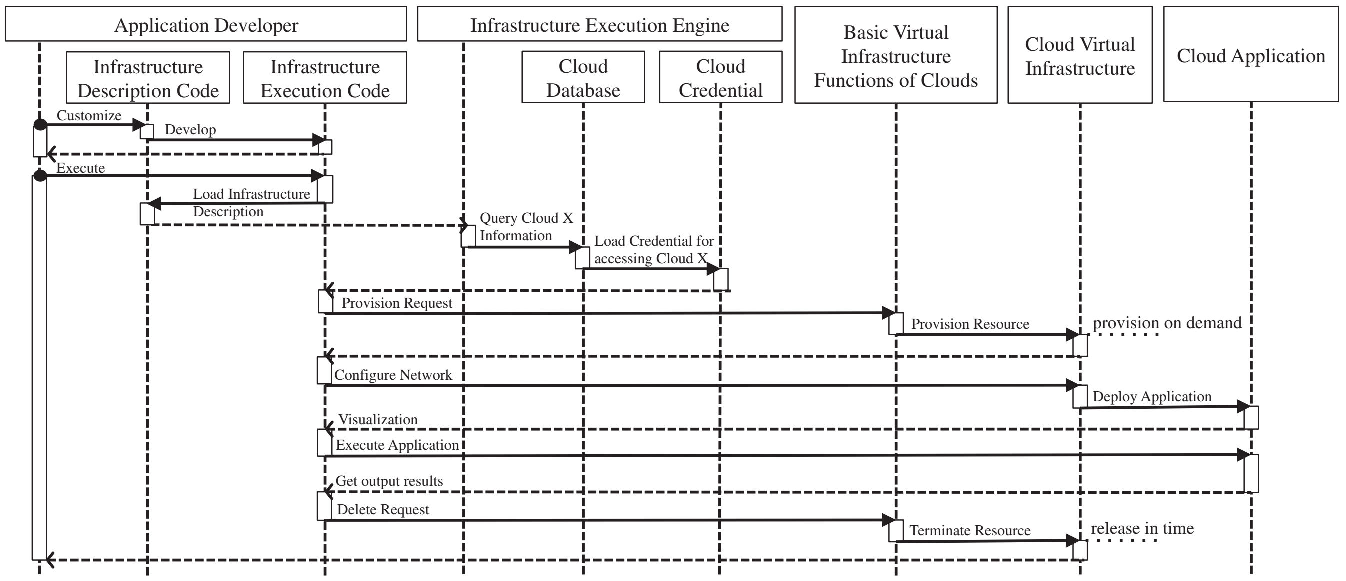


FIGURE 2 Sequence diagram of infrastructure code execution model

(b) According to the application language, the developer can leverage the corresponding "*Infrastructure Embedded Code*" to put the infrastructure control logic into the original application logic. In the example case, the developer rewrites the Hadoop data processing application that enables scaling out VMs according to the input data size, which enlarges the computing capacity of the underlying VM cluster. It is also programmed that the scaled VMs are terminated immediately after the application finishes the processing. The key function of the application is shown as Listing 1.

(c) The developer programs the "*Infrastructure Execution Code*" to perform operations of provisioning, deployment, and application execution, etc. In the example case, the developer programs operations, ie, provision the two VMs, download the input data, and execute the processing application developed in the previous step. These operations are important to set up the infrastructure for the application from scratch.

(d) The developer defines the "*Runtime Control Policy*" for identifying how to scale or recover the infrastructure when its resources perform insufficiently or even fail. In this case, the developer can define that when the VM's CPU utilization is above 50%, it is required to scale out one more VM.

2. CloudsStorm provides an "*Infrastructure Execution Engine*" to interpret the "*Infrastructure Execution Code*" for building the entire infrastructure from scratch. In this case, the developer invokes the engine through a command line with specifying the codes that he/she develops. The detailed method to invoke can be checked from the online manual.[§§]
3. The "*Infrastructure Execution Engine*" first sets up a "*Control Agent*" from a certain data center. Then the agent takes control of the infrastructure construction including the network connection, application deployment, and execution, etc. Via this manner, there comes the application operation phase. According to the runtime control model, all the control operations are performed by the "*Control Agent*" to dynamically adjust the infrastructure to satisfy the requirements of the application.
4. The "*Control Agent*" also provides a web-based graphical user interface (GUI) to show the current state of the infrastructure. In the example case, Figure 7 is a runtime snapshot showing that there are three scaled VMs in the Hadoop cluster to perform the processing task. The number of scaled VMs is in line with the input data size. Besides, there also web terminals provided to directly access each VM to check and get results.

[§§]https://cloudsstorm.github.io/

2.2 | Execution model

Figure 2 illustrates the execution model of the programmable infrastructure. It demonstrates how application developers leverage the programmed "*Infrastructure Description Code*" and "*Infrastructure Execution Code*" to automatically run their applications on clouds, which is a detailed description of step 2 in Section 2.1. After customizing the infrastructure as step 1 in Section 2.1, the developed "*Infrastructure Execution Code*" is executed by the "*Infrastructure Execution Engine*" and loads its required virtual infrastructure description from the "*Infrastructure Description Code*", including the number of VMs, network connections, the clouds or data centers involved in running the application, etc. The "Cloud X" information is then updated by querying the "Cloud Database". "Cloud X" represents a cloud defined in the "*Infrastructure Description Code*", where multiple clouds may be adopted. "Cloud Database" includes all the relevant data center information for these clouds, which are required to automatically control the VMs within these clouds. A "Cloud Credential" provided by the application developer is leveraged to access the desired cloud. The credentials can be strings of access keys or credential files. Both "Cloud Database" and "Cloud Credential" are application-defined and serve as a library for the "*Infrastructure Execution Engine*", explained in Section 3.6. After loading the credential information, the "*Infrastructure Execution Engine*" is able to invoke the basic "VIF" of the desired cloud, which contacts with the actual controller of that cloud. The request is then performed by the controller to provision certain VMs from that cloud. Moreover, CloudsStorm is also responsible for configuring the customized private network connection to construct "Cloud Virtual Infrastructure". Afterwards, the application is deployed onto the infrastructure depending on the "script" field, which is defined in the "*Infrastructure Description Code*". Meanwhile, the input data can also be prepared. Via the "*Infrastructure Execution Code*", the application developer also defines when and how to execute the application with the input data. While finishing the execution of the application, the "*Infrastructure Execution Code*" can be programmed to fetch the results from the VMs of remote clouds. Finally, excess computing resources can also be programmed to be terminated at that time to reduce costs.

Applications are therefore able to efficiently leverage the computing capability of clouds to get results through exploiting CloudsStorm because the computing resources are provisioned on demand and released immediately after acquiring those results. According to the pay-as-you-go policy of cloud, the longer you occupy a resource, the more you need to pay.

2.3 | Runtime control model

This section gives a detailed description of step 3 in Section 2.1. During the runtime operation phase of the application, the infrastructure is provisioned, and different components of the application run on the desired VMs. With the uploaded "*Infrastructure Description Code*" and "*Runtime Control Policy*", the "*Control Agent*" then takes over the responsibility to manage the infrastructure. Here, the "*Control Agent*" is placed in a separate VM of "Cloud Virtual Infrastructure" and other VMs are informed about the public IP of the "*Control Agent*" for communication.

Figure 3 illustrates the sequence diagram for the runtime control model. It consists of two controlling modes, *active*, and *passive*. The active mode is used for the programmed code to actively control the underlying infrastructure, which is a missing part for most current related tools. The control operation in this mode is performed by two types of code. During the normal infrastructure provisioning scenario, the "*Infrastructure Execution Code*" first sets up the "*Control Agent*" as mentioned above. On the other hand, the "*Infrastructure Embedded Code*" inside an application can actively invoke the "*Control Agent*" with the REST APIs to adjust its underlying infrastructure according to outside input conditions, eg, the input data size. The "*Control Agent*" performs the actual operations on the cloud infrastructure after checking constraints of the "*Runtime Control Policy*", eg, whether the budget is enough. Therefore, the application is able to actively customize the infrastructure to fit its requirements. This is demonstrated in the case study of Section 4.2.

The other is the passive mode. Figure 3 illustrates two scenarios when using passive control mode, which is autoscaling and failure recovery. The operations performed in passive mode are dependent on the "*Runtime Control Policy*" and monitoring information. In the scenario of autoscaling, the "*Control Agent*" analyzes the performance information collected from VMs of the infrastructure. If the CPU or memory usage of certain VMs meet the predefined threshold, the "*Control Agent*" performs a scaling operation according to the "*Runtime Control Policy*", In the other scenario of failure recovery, the unavailability of a certain data center can be known through continuous availability detection. Hence, according to the "*Runtime Control Policy*", the "*Control Agent*" is aware of where to recover that part of the unavailable infrastructure, ie, from which backup cloud and data center. We assume that there is an automatic failure recovery mechanism provided by the cloud provider for each individual VM. Therefore, we more focus on the failure case, where the data center is down or the network to the data center is not accessible. It is worth mentioning that once the operations are performed by the

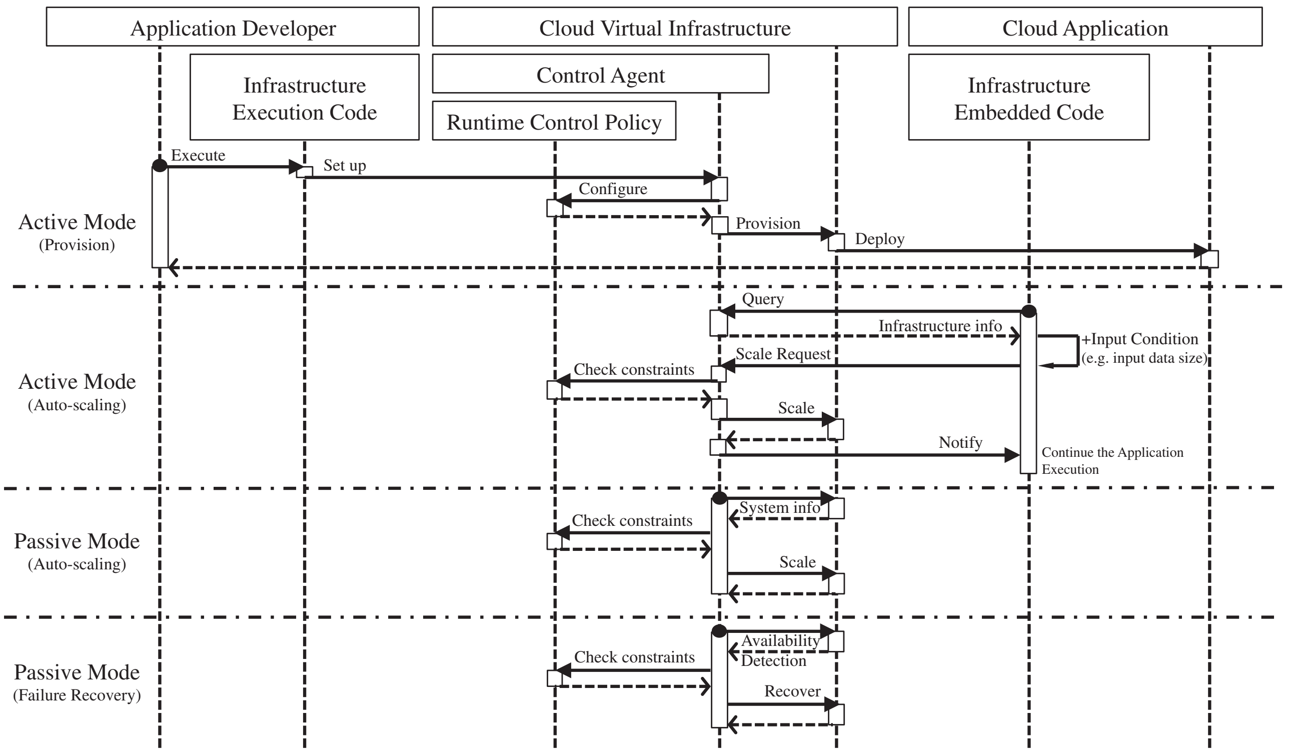


FIGURE 3 Sequence diagram for runtime control model

"*Control Agent*" onto the "Cloud Virtual Infrastructure", the "*Infrastructure Description Code*" managed by the "*Control Agent*" must be updated to keep synchronized with the actual status of the virtual infrastructure, eg, whether the VM is terminated or not.

3 | PROGRAMMABLE INFRASTRUCTURE IMPLEMENTATION

In this section, we introduce the implementation details of the programmable infrastructure. It includes four types of code provided by CloudsStorm, which the cloud application developer can leverage to program on the infrastructure. As mentioned above, they are the "*Infrastructure Description Code*", "*Infrastructure Execution Code*", "*Infrastructure Embedded Code*", and "*Runtime Control Policy*". The fundamental "*Infrastructure Execution Engine*" and "*Control Agent*" are also explained in this section.

3.1 | Infrastructure description code

The "*Infrastructure Description Code*" is proposed to provide the application developer with the design-level programmability on the infrastructure. In order to describe and manage the virtual infrastructure provided by different clouds or data centers, we propose a partition-based infrastructure management mechanism. This mechanism adopts a hierarchical description of the infrastructure topology, which is classified into three levels, ie, the levels of top topology, subtopology, and VM. Figure 4 shows an example topology to demonstrate this mechanism. The lowest level is the VM level. It enables the developer to customize a specific VM concerning different aspects. The level in the middle is the subtopology level. It provides a description of several VMs provisioned in the same data center. The top level is the top topology level. It includes all the subtopologies and describes the network connections among all VMs. The reason to differentiate the subtopology and top topology is to manage the infrastructure more flexibly and efficiently. Because some infrastructure operations are not intended to be applied on the entire infrastructure, ie, the top topology. For instance, we only want to terminate all the VMs from a specific data center. On the other hand, the operation performed on a subtopology is applied to all the VMs inside in parallel. Hence, we only need to terminate the subtopology from that data center, instead of terminating the VMs one by one.

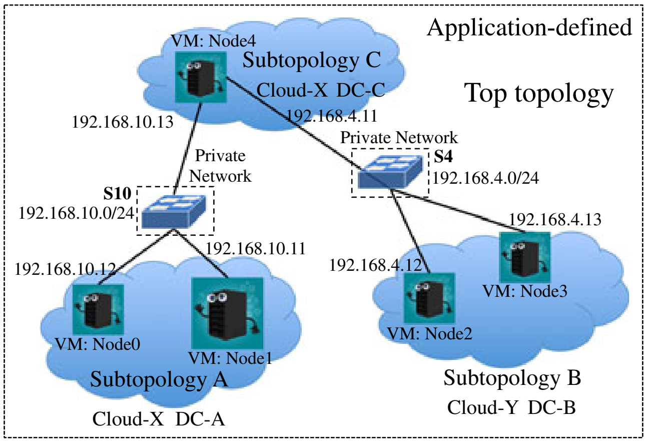


FIGURE 4 Partition-based infrastructure management example. VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

Besides, the network among the VMs is defined as a private network. Considering the fact that the public IP address for each VM in a public cloud is typically different after provisioning every time, it is useful for the application to define the topology as a private network during the design phase, ie, (1) the actual network is made transparent to the application. In the previous example case, the configuration file for the Hadoop application to identify the addresses of the computing nodes can always keep the same with the fixed defined private IP addresses. The first case study in Section 4.1.1 also demonstrates this benefit; (2) the infrastructure for the application is repeatable regardless of the geographical information, ie, data centers. In this case, the same network for the Hadoop cluster is provisioned every time, no matter which data center the VMs come from, even come across different data centers; (3) get rid of the provisioning dependency. For example, we do not have to provision a database VM before a web server VM, in order to configure the web server with the public IP address of the database. With the ability to predefine private IP addresses, these VMs can be provisioned in parallel to improve efficiency. This is also demonstrated in Section 4.3.2.

In addition, the switches placed in the figure only illustrate how VMs are connected. The technique detail to provision such networked infrastructure using an overlay network is demonstrated in our previous work.[4]

Syntax 1 Subtopology description

```
VMs:
- name: $Node
  [nodeType: $Type]
  [CPU: $num_1]
  [Mem: $num_2]
  OSType: $OS
  script: $Path
- *
```

In order to describe the networked infrastructure for the application, we adopt the YAML format to define the subtopology and top topology description, shown as Syntax 1 and Syntax 2, respectively. In these syntax descriptions, "-" is used to represent the start of a list and "∗" is used to represent repetition of the element above. Symbol "$" designates that the succeeding string identifies an application-defined variable. In addition, Symbol "|" indicates different alternative values, any of which can be used in the corresponding code. Symbol "[]" is exploited to represent an optional field. Syntax 1 defines the subtopology description, which is a list of "VMs". Every element contains the VM name, such as "Node0" or "Node1" in Figure 4. "nodeType" indicates the computing capacity of the VM, such as "t2.small" or "t2.medium" for the EC2 cloud, "XOSmall" or "XOMedium" for the ExoGENI cloud. However, the value of this field is heavily dependent on the cloud and not user-friendly. However, it can be omitted, if the following "CPU" and "Mem" are specified. CloudsStorm would automatically find the corresponding "nodeType", according to the vCPU number and memory capacity (in Gigabyte). For instance, the VM with 1 vCPU and 1 GB memory belongs to the type of "XOSmall" of ExoGENI. "OSType" indicates the specific operating system required by the application. "script" is the script path, which is leveraged to install and configure the runtime environment for the application.

Syntax 2 Top topology description

```
userName: $User
publicKeyPath: $Path
topologies:
- topology: $SubTopology
  cloudProvider: $Cloud
  domain: $DC
  status: 'fresh | running | deleted | failed | stopped'
- *
subnets:
- name: $Subnet
  subnet: $subnet
  netmask: $netmask
  members:
  - vmName: $SubTopology.$Node
    address: $IP
  - *
```

The top topology description is defined as Syntax 2. First, "userName" and "publicKeyPath" indicate whether the application developer wants to have a unified SSH account, *$User*, to access all the VMs, no matter which cloud the VM comes from. The access key is always the corresponding private key of the public key defined by "publicKeyPath". The top topology description also contains a list of subtopologies defined in "topologies". "topology" here is the application-defined name of a certain subtopology, such as "A", "B", or "C" shown in Figure 4. "cloudProvider" and "domain" specify the concrete data center where this subtopology is hosted. For example, there is a *$DC = California* data center from *$Cloud = EC*2. "status" indicates the status of this subtopology. They are used for the resources lifecycle management. Another list in the top topology definition is "subnets", including all the private subnets required by the application in the top topology. In each subnet, there are a field "members" to list all VMs in the subnet and their corresponding private IP addresses. "vmName" here is the full name, which consists of its subtopology name and the node name itself. For instance, there are two subnets shown in Figure 4, with the "name" of "S10" and "S4". Taking the example of subnet "S10", it has *$subnet* = 192.168.10.0 with *$netmask* = 24 and it contains three members "A.Node0", "A.Node1", and "C.Node4" with the corresponding private addresses.

3.2 | Infrastructure execution code

The "*Infrastructure Execution Code*" is proposed as a means to provide the application developer with the infrastructure-level programmability on the virtual infrastructure. This type of code enables the application developer to directly perform operations on the infrastructure, such as provisioning, terminating of some resources, and execution of commands on some VM. It is separated with the application logic and also based on YAML format. Comparing to the aforementioned static "*Infrastructure Description Code*", which is only the application-defined topology description, the "*Infrastructure Execution Code*" focuses on the infrastructure operations and therefore is executable. Hence, it is important for the application developer to automatically provision the infrastructure, deploy, and start running their applications on clouds through programming the "*Infrastructure Execution Code*".

Syntax 3 "SEQ" Infrastructure Code

```
- CodeType: 'SEQ'
  OpCode:
    Operation: 'provision | delete | execute | put | get | vscale | hscale | recover | start | stop'
    [Options:]
      [$String_i: $String_j]
      *
    [Command: $String]
    ObjectType: 'SubTopology | VM | REQ'
    Objects: $Object_1 [||$Object_2]...
```

The "*Infrastructure Execution Code*" is basically a set of operations defined sequentially in a list. In order to combine these basic operations to complete a complex task, we define two code types. One is "SEQ", which only contains one operation; a list of "SEQ" codes are executed one at a time. The other is "LOOP", which contains several operations and performs repeatedly for a number of iterations or within a certain period. The syntax of "SEQ" is shown in Syntax 3. It contains only one operation expressed by "OpCode". Current alternative operations are "provision", "delete", "execute", "put", "get", "start", "stop", "vscale", "hscale", and "recover". They are specified in the field "Operation". Field "Options" is a list of key/value pairs to specify whether there are some arguments needed for this operation. When the operation is "execute", the *$String* of "Command" is the specific command to be executed on the VM. Both of these two fields are optional. Field "ObjectType" indicates whether this operation is operated on a "SubTopology" or on an individual "VM". In addition, "REQ" is used to represent a scaling or recovery request. The concrete examples are in the case study 2 of Section 4.1.1. "Objects" then refers to the objects set. To define this set, we adopt the symbol "||" from parallel λ-calculus[9] to express parallel operation, such that all "Objects" are operated in parallel, improving the operation efficiency. We can further prove the computability that all the high-level operations can be derived from the basic cloud VIFs using λ-calculus. In summary, "provision" and "delete" operations are leveraged to acquire and release cloud resources; "start" and "stop" can be used when the cloud supports the operation of starting or stopping a VM; the "execute" operation is used to execute the application; "hscale" is exploited to do horizontal scaling, which is used to add/remove computing resources to/from current infrastructure; "vscale" is exploited to do vertical scaling, which is used to increase/decrease a VM capability while keeping the original network connection; and "recover" is for subtopology level failure recovery.

In order to finish complex tasks, we provide the "LOOP" code type as shown in Syntax 4. It consists of several operations executed in sequence instead of only one operation. Apart from this, there are three kinds of conditions for exiting a loop. "Count" is defined as the maximum number of iterations for this loop. "Duration" is defined as the maximum amount of time for executing in this loop. "Deadline" is defined as a certain timing to exit this loop, which is represented in Unix timestamp. There must be at least one condition defined for a "LOOP" code. If there are several conditions defined, then the loop is ended when one of the conditions is met.

The final "*Infrastructure Execution Code*" is therefore an ordered combination of these codes. We omit the detailed operation definitions and define the rest as follows:

- CodeType: "SEQ | LOOP";
- *.

A case study of a task-based application demonstrating how the "*Infrastructure Execution Code*" is programmed is shown in Section 4.1.

Syntax 4 "LOOP" Infrastructure Code

```
- CodeType: 'LOOP'
  [Count: $num]
  [Duration: $time_1]
  [Deadline: $time_2]
  OpCodes:
  - Operation: 'provision | delete | execute | put | get | vscale | hscale | recover | start | stop'
    [Options:]
      [$String_i: $String_j]
      *
    [Command: $String]
    ObjectType: 'SubTopology | VM | REQ'
    Objects: $Object_1 [||$Object_2]...
  - *
```

3.3 | Infrastructure embedded code

The "*Infrastructure Embedded Code*" is proposed to provide the application developer with application-level programmability on the virtual infrastructure. These are interfaces developed in a specific general-purpose programming language (currently Java). Since it is not a domain-specific language as mentioned above, the "*Infrastructure Embedded Code*" can be embedded inside the application logic when adopting interfaces in the same programming language as the cloud application. These interfaces mainly provide functions to query the status of current infrastructure, and provision, delete, and scale some resources of the infrastructure. The final request made by the implemented interface is actually translated to a REST request to the "*Control Agent*", since the "*Control Agent*" mentioned in Section 2.3 provides some REST APIs to be invoked to perform infrastructure runtime management. One advantage of this design is that the related library for supporting the "*Infrastructure Embedded Code*" is relatively lightweight. Because the major library required is simply the one able to make corresponding REST calls. Therefore, each VM in the infrastructure does not need the heavy libraries of "*Infrastructure Execution Engine*" to perform operations. Another advantage is that operations on the infrastructure can be performed in parallel with application execution. Because the infrastructure operation is then performed by the "*Control Agent*" after the application makes the REST request.

Pseudocode 1 Pseudocode for Infrastructure Embedded Code

```
1: ...original application logic block 1 ...
2: Initialize the ControlAgent
3: executionID = ControlAgent ⇒ infrasOperation(request)
4: ...original application logic block 2 ...
5: if CtrlAgent ⇒ waitInfrasOperation(executionID, timeOut) != NULL then
6:     ...continue with application logic block 3 ...
7: else
8:     Throw an Exception of the Infrastructure Operation
9: end if
```

Pseudocode 1 shows the general procedure to leverage the "*Infrastructure Embedded Code*". Line 3 in Pseudocode 1 is to make a REST call to invoke the "*Control Agent*" to perform the infrastructure operation. It is worth mentioning that this function is nonblocking, which means the "*executionID*" is immediately returned back from the "*Control Agent*". Here, "*executionID*" is a string value to identify the operation. The following original application logic block 2 can therefore be executed concurrently during the infrastructure operation. The function in line 5 awaits the accomplishment of the infrastructure operation, since the application logic block 3 can only be executed after the infrastructure is adjusted. In addition, this function is a blocking one, which only returns when the operation of "*executionID*" is finished. The input parameter of "*timeOut*" is set to ensure the function can always return when the infrastructure operation cannot be properly executed. The detailed usage of the "*Infrastructure Embedded Code*" in Java is demonstrated in the case study 2 of Section 4.2.1.

3.4 | Runtime control policy

The "*Runtime Control Policy*" is proposed to provide the controllability in the passive mode of the control model, which is enforced according to the monitoring information.[10] This policy defines some operations, which are performed when some condition is met. During the runtime, this policy is managed by the "*Control Agent*" and the infrastructure is therefore passively adjusted. Syntax 5 shows the syntax of the "*Runtime Control Policy*", which is also based on the YAML format. The syntax is designed to contain a list of policies, termed as "CTRLPolicies". Each policy consists of two parts.

One part contains the objects and the "Metrics", which defines a set of performance thresholds according to the monitoring information of some infrastructure resources. The "Metrics" is a set of key-value pairs, ie, (1) the key of the pair defines the metric type to monitor such as "CPU" for CPU utilization, "MEM" for memory utilization, "ALIVE" for availability detection, and "$\$String_1$" for some application-defined metric. (2) The value of the pair defines how the condition should be met. Among them, "TimeUnit" defines the monitoring interval in second. "AboveThreshold" or "BelowThreshold" defines the situation counted when the metric is above or below a certain threshold. Then, the final condition is met when the aforementioned situation happens "SeqTimes" ($\$num_4$) times sequentially or "TotalTimes" ($\$num_5$) times in total. As long as the condition defined by one of the key-value pairs is met, following defined infrastructure operations will be triggered.

Syntax 5 Runtime Control Policy

```
BudgetPerHour: $num
CTRLPolicies:
- ObjectType: 'SubTopology | VM'
  Objects: $Object_1 [||$Object_2]...
  Metrics:
    CPU | MEM | ALIVE | $String_1:
      [AboveThreshold: $num_1]
      [BelowThreshold: $num_2]
      [TimeUnit: $num_3]
      [SeqTimes: $num_4]
      [TotalTimes: $num_5]
    *
  OpCodes:
  - Operation: ...
    ...
  - *
- *
```

These operations are defined in the other part of the policy, termed as "OpCodes". The detailed syntax of "OpCodes" is the same as the definition of that in Syntax 4 and therefore is omitted. The performance of this controllability, such as autoscaling, failure recovery, is evaluated in Section 4.3. Moreover, "BudgetPerHour" allows the developer to add a monetary constraint to all of the infrastructure operations. Its value is in dollars and it limits the maximum cloud resource usage per hour.

3.5 | Infrastructure execution engine and control agent

The "*Infrastructure Execution Engine*" is the elementary engine implemented to complete the execution procedure demonstrated in Figure 2. It is responsible for interpreting the "*Infrastructure Execution Code*", provisioning the application-defined virtual infrastructure among clouds. Especially, it includes provisioning the corresponding private network. It contains T-Engine, S-Engine, and V-Engine, as shown in Figure 5. A T-Engine is responsible for "Top topology" management. Hence, T-Engine is the entry point for the application to access and control its entire infrastructure. It also manages the connections among subtopologies. T-Engine takes the interpreted operations as input requests to provision or delete the corresponding cloud resource. According to the execution model in Section 2.2, T-Engine first queries the relevant clouds information. The information in the cloud database is provided by our framework and this database component works as a supporting library. T-Engine then sets up the corresponding S-Engine for a specific cloud, eg, "S-Engine-EC2" for cloud "EC2". Meanwhile, the T-Engine loads the corresponding cloud credential to make the S-Engine able to access the cloud. This cloud access credential is provided by the application developer for authentication and billing. The V-Engine is responsible for operations on each individual VM in its virtual infrastructure. All operations are transferred from the upper level to this VM level and V-Engine is the final engine used to complete a particular operation. In CloudsStorm, V-Engine is also responsible for building up the virtual network functions on each VM to provision the networked infrastructure required by the application. Currently, we implement virtual network functions based on the tunneling technique, proposed by our previous work,[4] to connect the VMs from federated clouds as a private network. After provisioning, the V-Engine is able to execute the application-defined script to configure the runtime environment and deploy the application. Here, V-Engine is defined as a basic class. Different customized V-Engines can be inherited from it depending on the features of the VM, eg, "V-Engine-ubuntu" for an Ubuntu VM. If the application has specific operations on some VM, it can customize its own V-Engine. In addition, a new cloud can also be supported by deriving its own V-Engine. Our programmable infrastructure framework can therefore be easily extended to support different clouds. This pluggable V-Engine implementation is realized according to the factory design pattern in software engineering. Moreover, all the S-Engines and V-Engines use multi-threading to run in parallel, allowing the T-Engine to start several S-Engines at the same time. If subtopologies managed by these S-Engines belong to different data centers, there will be no conflict among them, and they can run totally in parallel. This is the same for V-Engines, ie, the

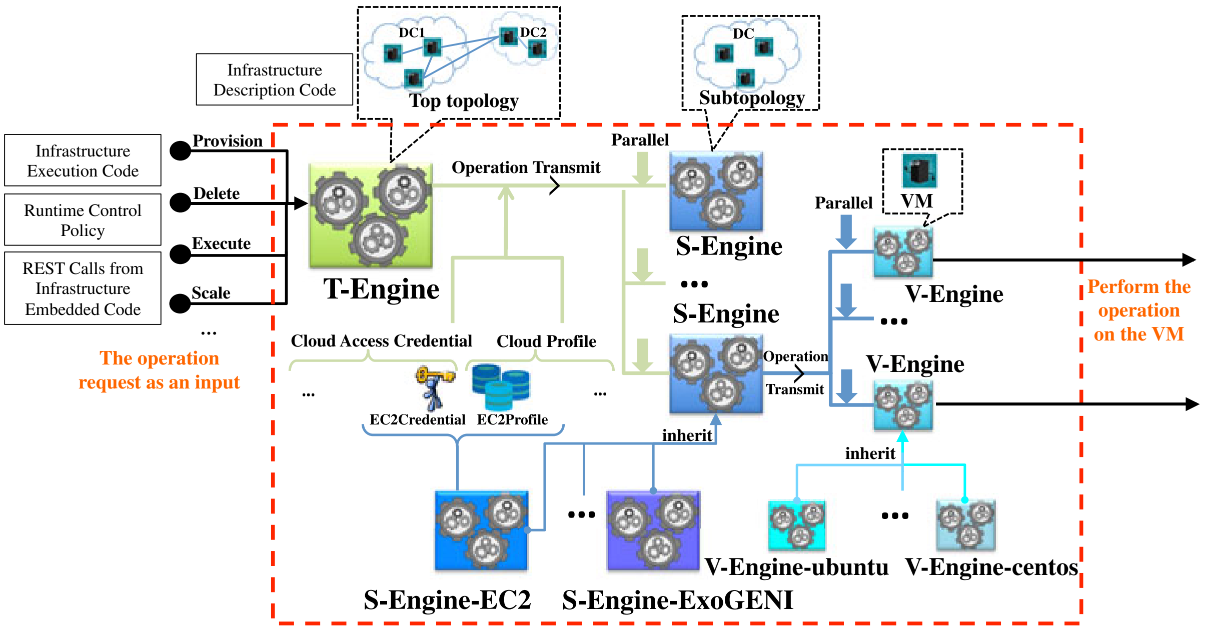


FIGURE 5 Implementation details of infrastructure execution engine. EC2, Amazon Elastic Compute Cloud; REST, representational state transfer; VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

operations on all the VMs in one subtopology can proceed simultaneously. This mechanism is leveraged to realize the parallel symbol "||" in multiple "Objects" definition in Section 3.2. In other words, "*Infrastructure Execution Engine*" can accelerate operations to reduce the total time needed by an application utilizing the cloud, reducing the application's total cost as paid to cloud providers.

From the implementation perspective, the factory design pattern in software engineering is leveraged to make it possible for customizing different types of engines from the basic class. The adapter design pattern is adopted for upper-level engine easily controlling different types of engines at the lower level in parallel. All these design patterns make the entire framework pluggable and highly efficient. Just similar to the MapReduce framework, the developer only needs to provide the minima VIFs of a new cloud, ie, provisioning and terminating a VM. CloudsStorm then is able to plug in this basic VIF and afford high-level infrastructure programmability and controllability of this new cloud for the application developer. CloudsStorm currently supports three cloud providers, EC2, ExoGENI,[‡‡11] and European Grid Infrastructure[¶¶] (EGI).

In addition, "*Control Agent*" plays as the key role for the application to control the infrastructure during runtime. But still, "*Infrastructure Execution Engine*" is the main component of the "*Control Agent*". The other component of "*Control Agent*" provides web services. During runtime, these web services contain a set of REST APIs to receive the infrastructure operation requests and then invoke "*Infrastructure Execution Engine*" to perform. They also provide a web-based GUI to show the status of the infrastructure. The detailed implementation of "*Control Agent*" is also open source.[##]

3.6 | Relevant libraries and logging

As discussed above, there are two essential "libraries" required for executing infrastructure code. One of them is the cloud database. It contains detailed information about data centers for selected clouds. This information includes (1) geographic positioning of each data center, which can be leveraged to do locality-aware or data-aware provisioning; (2) endpoint information, which describes a URL of a data center controller needed for actual provisioning; (3) VM types (CPU, memory) supported in each data center and their characteristic data (eg, price). This information is not application-defined but provided by our framework. The other library is for cloud credentials. It defines key-value pairs that specify the security tokens needed to access a cloud or the file paths for cloud credential files. For instance, two tokens, ie, "accessKey" and "secretKey", are needed for accessing EC2. Hence, the credentials are given and managed by the application developers

[¶¶]https://www.egi.eu/
[##]https://github.com/zh9314/CloudsStormCA

themselves. This way of key management avoids the privacy issue of sending cloud credentials to a third proxy broker for provisioning. Both of these two libraries are organized in YAML format.

Finally, a logging component is built in CloudsStorm. The log file is also organized in YAML format. For each operation defined in the "*Infrastructure Execution Code*", there will be a log element in the log file to record the operation overhead after the operation is finished. Some extra information is also recorded. For example, the detailed provisioning overhead, which is the time starting from the sending out of the cloud request to the point where the VM is activated and accessible, is also recorded for each "provision" operation. Moreover, for the "execute" operation, all the standard outputs of this operation are recorded in the log as well, providing another way to obtain the output results of the application.

Due to space considerations, the detailed syntax of the above components are not explained, which can be checked from the online manual of CloudsStorm[§§].

4 | CASE STUDIES AND EVALUATIONS

In order to demonstrate how to benefit from the CloudsStorm framework, we conduct experiments with two case studies. They include the task-based application and data-aware processing. We show that the developer is able to build the application on clouds more efficiently using CloudsStorm. Moreover, the evaluation results prove that the cost of cloud resource usage is also reduced when adopting CloudsStorm. Finally, we evaluate the controllability performance of the framework itself and compare it with some other related tools. In addition, we also apply CloudsStorm in building disaster early warning systems on clouds.[12]

4.1 | Task-based applications: coprogramming "task infrastructure" for elastic scheduling

Our experiments are conducted on real clouds instead of simulators. In order to show the programmability at the design level and infrastructure level in CloudsStorm, we demonstrate a case study of a task-based application, in this case for software testing. In this scenario, we assume there is a software, which is required to be tested for many times. We leverage cloud resources to develop this application and execute it on multiple data centers to get results. In order to develop this application, we pick several data centers from ExoGENI[‡‡]. These data centers are located in Sydney (in Australia), University of Amsterdam (in Europe, UvA for short), Boston (in eastern USA), and Oakland (in western USA). The task of the application in this case study is to provision a VM in each aforementioned data center and leverage "sysbench" to test the CPU/memory performance of each data center. The provisioning overhead is recorded as well. For testing the network performance, we test the bandwidth from various VMs in data centers of Oakland, Sydney, and Boston, to the VM in the UvA data center. In this case study, these tests require to be repeated regularly. For comparison, all the VMs have one virtual CPU and 3 GB memory, which is the capacity of a "XOMedium" for VM in ExoGENI.

Finally, the goal of this case study is to test (1) whether the application can leverage our programmable framework, CloudsStorm, to program on its infrastructure; (2) whether these task-based applications can run on clouds effectively and get the desired results; (3) whether cloud resources can be provisioned on demand and released in time for reducing cost.

4.1.1 | Example solution and results

In order to complete the task in this application scenario, we first design our networked infrastructure topology according to Section 3.1. We define four subtopologies with one VM in each subtopology. All these five VMs are in the same subnet, which is "192.168.10.0/24". We then leverage the "*Infrastructure Execution Code*" to define the entire process for performing tests on the selected clouds. The pseudocode is shown as follows. We use the "LOOP" code to repeat the test tasks. First, we provision the corresponding cloud resources; then, we define the operations of the CPU test and memory test simultaneously on all the object VMs. The definition of parallel operations is achieved by still using "||" in Section 3.2. For the network performance test, though the public IP addresses of the VMs cannot be determined, we can always use the defined private IP address to conduct the bandwidth test. This is another advantage of provisioning a networked infrastructure on clouds with CloudsStorm. Finally, we get the results and wait for executing another round of tests.

Pseudocode 2 Example solution with the Infrastructure Execution Code

```
for a certain time period or a certain count do
    Provision 'SubTopology' from OSF||SYD||BBN||UvA
    Execute CPU test simultaneously on the 'VM' of   OSF||SYD||BBN||UvA
    Execute memory test simultaneously on the 'VM' of   OSF||SYD||BBN||UvA
    Perform bandwidth test from the 'VM' of OSF to the 'VM' of UvA   (always test: "192.168.10.11" -> "192.168.10.10")
    Perform bandwidth test from the 'VM' of SYD to the 'VM' of UvA   (always test: "192.168.10.12" -> "192.168.10.10")
    Perform bandwidth test from the 'VM' of BBN to the 'VM' of UvA   (always test: "192.168.10.13" -> "192.168.10.10")
    Get the results
    Wait for executing another round of tests
end for
```

In this case, all the test results are recorded in the infrastructure code log, described in Section 3.6. Hence, we do not need explicitly to define an operation to retrieve results from remote resources. We directly extract test results from the log, analyze it and present the results in Figure 6. It illustrates four types of results as a boxplot, including the result generated by the task of provisioning test, CPU test, memory test, and bandwidth test. The *x*-axis refers to the four corresponding data centers. For this case, we extract 1500 iterations of test cases to obtain the results; the time interval between each test case

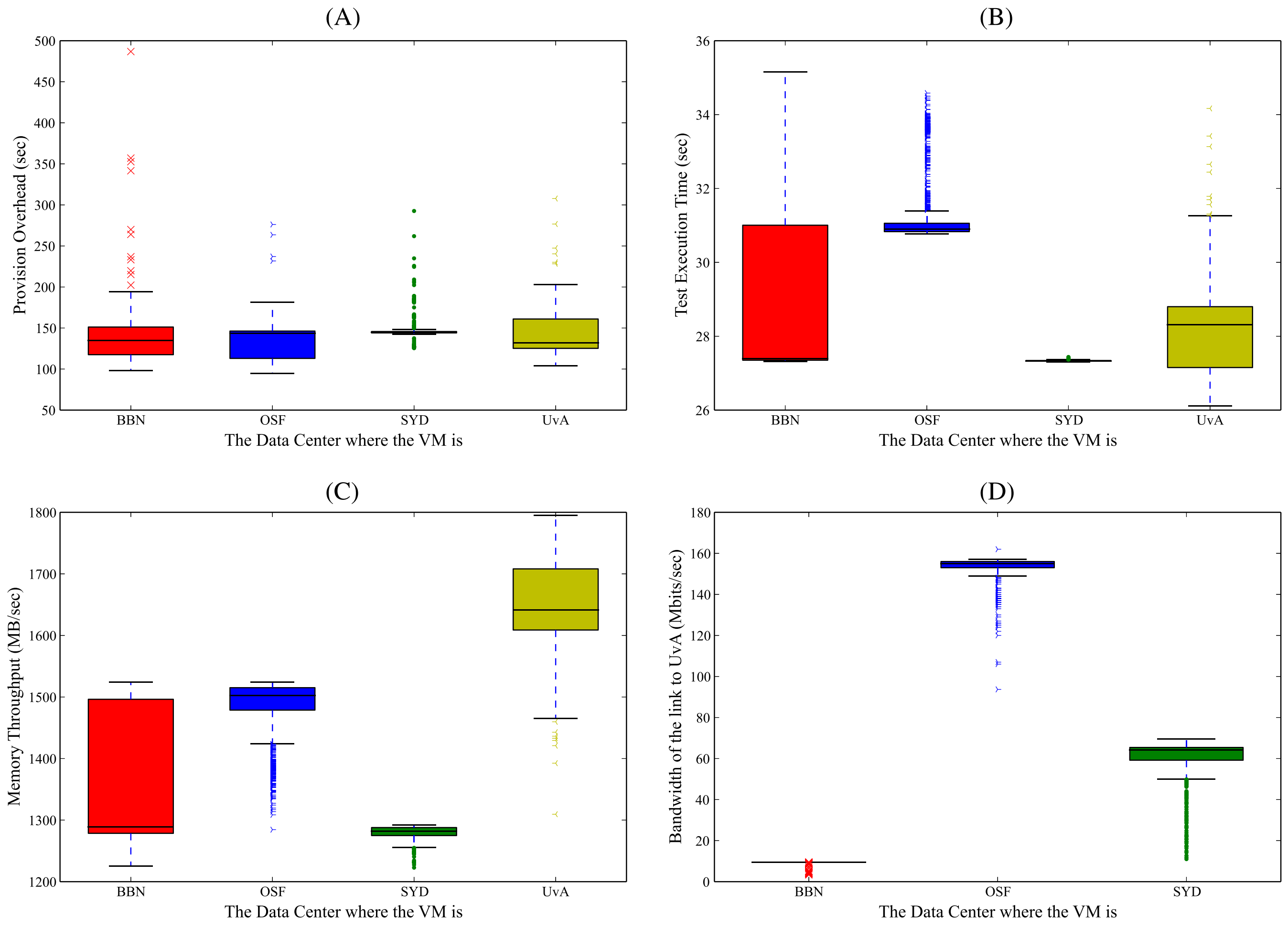


FIGURE 6 Results of performance test tasks in this assumed task-based application scenario based on CloudsStorm. BBN, Boston; OSB, Oakland; SYD, Sydney; UvA, University of Amsterdam; VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

```
public boolean dataAwareProcessing(Job wordCountJob, String inputFilePath){
  /** Get the data size of the input file in GigaByte.
      The returned value is applied with a ceiling function on the actual data size, e.g., 4.2G -> 5G. **/
  int dataSize = getInputDataSize(inputFilePath);

  CtrlAgent ctrlAgent = new CtrlAgent();
  HScalingRequest hScaleReq = new HScalingRequest();
  hScaleReq.cloudProvider = ''ExoGENI'';
  hScaleReq.dataCentre = ''UMass (UMass Amherst, MA, USA) XO Rack'';
  hScaleReq.scalingDirection = ''OUT'';
  hScaleReq.targetObjectType = ''VM'';
  // The VM defined in the infrastructure description code, which is to be scaled.
  hScaleReq.targetObjects = ''hadoop_1_node.Node0'';
  String exeID = null;
  // Horisontally scaling out certain number of VMs to be the datanode of Hadoop, according to the data size.
  if( (exeID = ctrlAgent.init().addHScalingReq(hScaleReq, dataSize).hscale()) == null )
      return false;

  // At the same time, the raw input files are uploaded to the HDFS. Because the above function is non-blocking.
  String hdfsDir = upload2HDFS(inputFilePath);

  //Waiting for the underlying infrastructure to be ready. Time out is set to dataSize*200 seconds.
  if( !ctrlAgent.waitInfras(exeID, dataSize*200) )
    return false;

  // Set the input/output path of the word count job and start data processing.
  FileInputFormat.addInputPath(wordCountJob, new Path(hdfsDir));
  FileOutputFormat.setOutputPath(wordCountJob, new Path(''/output''));
  wordCountJob.waitForCompletion(true);

  hScaleReq.scalingDirection = ''IN'';
  // Horisontally scaling in to release extra computing resources and therefore reduce cost.
  if( (exeID = ctrlAgent.init().addHScalingReq(hScaleReq, dataSize).hscale()) == null )
      return false;
  if( !ctrlAgent.waitInfras(exeID, 200) )
    return false;
  return true;
}
```

Listing 1 Infrastructure embedded code example in data-aware processing

is about 10 minutes. For this case study, the detailed infrastructure topology descriptions and code can be downloaded,[‖] due to the space limitations of this paper.

4.1.2 | Evaluation

In this section, we evaluate the efficiency of the CloudsStorm framework, where the efficiency, in this case, refers to how much cost (money) we can save through leveraging CloudsStorm to run these task-based applications. The efficiency thus comes from two aspects. One is that CloudsStorm can provision application-defined cloud resources on demand and release them immediately after they are no longer needed. The other is that CloudsStorm can perform operations in parallel to reduce the entire execution time for the application. We therefore first compare our cost with the cost of manually setting up and traditionally configuring cloud resources. We then compare our cost with the cost of on-demand provisioning the resource and executing the application, which can be achieved by using a specifically developed script or some vendor-specific tools. However, the operations only run sequentially. Due to the fact that cloud providers charge based on usage time (eg, EC2 charges in seconds), the cost of cloud resources is proportional to the resource usage time. We can therefore measure the cost of cloud resources as being directly proportional to the resource total usage time. This information is also recorded in the log as the time of the operational overhead. Through analyzing these logs, we can calculate the cloud resource usage time of manual and on-demand configuration, respectively. Finally, the comparison of the cost can be achieved.

The comparison results various with the number of task iterations are shown in Table 1, where *cost*(MC) and *cost*(OC) refer to manual and on-demand configuration, respectively. In the case of *MC*, we set up these VMs manually and run the application to perform tests. Hence, cloud resources need to be kept running during the entire application execution lifecycle. Taking the aforementioned scenario as an example, the actual execution time for each test case is about 5 minutes and the remaining 10 minutes is waiting. Therefore, the traditional manual method completely wastes resources during

[‖] https://github.com/CloudsStorm/ExampleRepo/releases/download/st/ExoGENISoftwareTests.zip

	The number of task interactions						
	10	**50**	**100**	**250**	**500**	**1000**	**1500**
cost(CloudsStorm) / *cost*(MC)	47.4%	44.7%	44.4%	44.5%	44.2%	44.2%	44.1%
cost(CloudsStorm) / *cost*(OC)	41.8%	41.5%	41.4%	41.5%	41.5%	41.4%	41.3%

TABLE 1 Cloud resource usage evaluation among manual configuration, on-demand configuration and using CloudsStorm in case study 1, varied with the task iterations

this waiting time. In the other case of *OC*, we can develop a specific script to automate provisioning and terminating resource from some specific cloud to reduce the wasteful waiting time. However, the script cannot perform operations in parallel. For example, in the aforementioned scenario, the provisioning, CPU, and memory tests can be performed in parallel. Hence, compared to the manual configuration, the cost saved by CloudsStorm depends on the intervals between the tasks. Compared to the on-demand configuration, the cost saved by CloudsStorm depends on how many operations are in parallel. According to the parameters' setting in our case study, Table 1 shows that the cloud resource usage cost using CloudsStorm is always around 44.5% and 41.5% of the cost using manual and on-demand configuration, respectively, even when executing a different number of task iterations.

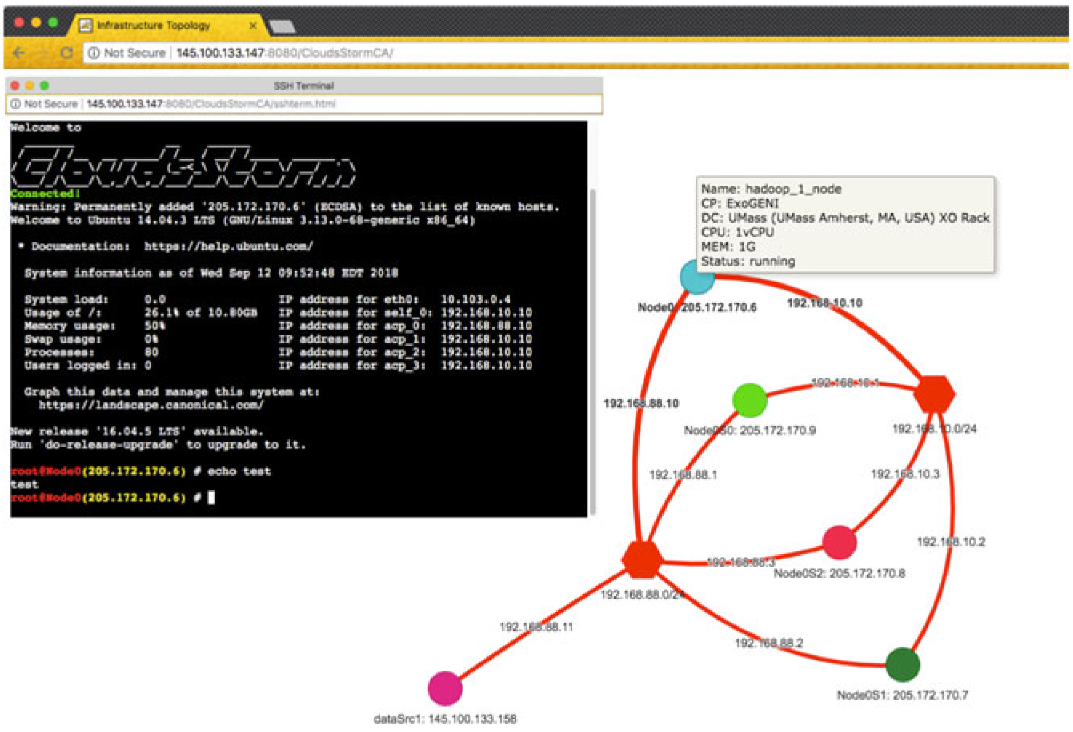
4.2 | Big data applications: co-controlling “data infrastructure” for data-aware processing

Another case study we conduct with the CloudsStorm framework is related with data processing applications. In this case study, we base it on the well-known big data processing platform Hadoop to develop the classic demonstration application of “Word Count”. Through leveraging the programmability provided by CloudsStorm, our developed word count application achieves the controllability of its own infrastructure. Especially, the “*Infrastructure Embedded Code*” programmed inside the application logic empowers the application with the ability to dynamically adjust the underlying infrastructure at runtime according to input data size. The underlying infrastructure can therefore provide a proper amount of computing resources with network connections on demand, without overprovisioning. In this way, the computing resource consumption is reduced.

4.2.1 | Example solution and results

In order to develop this data-aware processing application, the application developer would first design the infrastructure topology. For demonstration, there are two subtopologies within the entire top topology in the initial virtual infrastructure design, termed as “dataSrc1” and “hadoop_1_node” in this example. Each of them contains one VM and is from a different data center of ExoGENI. Here, subtopology, “data_src_1”, is from the “UvA” data center, which is located at UvA, to be assumed as the data source of this application. Subtopology, “hadoop_1_node” is designed to be from the “UMass” data center located in the USA. The VM defined in the subtopology “data_src_1” is termed as “dataSrc1” and the one in “hadoop_1_node” is termed as “Node0”. These two VMs are connected within a private subnet. In the beginning, only one VM, “Node0”, constructs the Hadoop cluster and downloads the input data from “dataSrc1”. Afterwards, the word count application based on Hadoop is executed. All of these steps are automatically realized through utilizing the “*Infrastructure Description Code*” and the “*Infrastructure Execution Code*”, which are similar with the case study in Section 4.1.1. However, in this case study, we further leverage the “*Infrastructure Embedded Code*” of CloudsStorm to help the application achieve more controllability on the infrastructure. The following Listing 1 shows how the function of the data-aware processing part is implemented within the CloudsStorm framework in Java.

In this example function, we first get the data size in gigabytes of the input file, with the value then rounded up to an integer, eg, the data size returned for a 4.2 GB file is 5. The developer then can program a request on the infrastructure to scale out the computing resources according to the data size. In this example, the application is programmed to scale out a certain number of VMs based on the template of “Node0” from the “UMass” data center of ExoGENI. The number here is equal to the data size, which is also actually up to the developer to decide on how many more VMs are indeed needed. Meanwhile, the location of the scaled ones can also be programmed. After sending the request to the “*Control Agent*”, an “exeID” is immediately returned back. It works as a token to identify this operation. Hence, this program does not need to wait until this operation on the infrastructure completes. It also means that other operations of the application can also be executed in parallel during the infrastructure operation. In this example, we upload the input data onto the HDFS (Hadoop Distributed File System) in order to be processed. In this phase, no more computing resources are needed as long as the storage of a one-node cluster is enough for the input data. When the input uploading operation is finished, the interface “waitInfras” provided by the “*Infrastructure Embedded Code*” can be invoked to ensure the scaling operation on the infrastructure is completed. Then, the computing task of “word count” is started with a scaled cluster to achieve


Infrastructure Topology
Not Secure | 145.100.133.147:8080/CloudsStormCA/
SSH Terminal
Not Secure | 145.100.133.147:8080/CloudsStormCA/sshterm.html

```
Welcome to
CloudsStorm
Connected!
Warning: Permanently added '205.172.170.6' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-68-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Wed Sep 12 09:52:48 EDT 2018

  System load:     0.0                IP address for eth0:   10.103.0.4
  Usage of /:      26.1% of 10.80GB   IP address for self_0: 192.168.10.10
  Memory usage:    50%                IP address for acp_0:  192.168.88.10
  Swap usage:      0%                 IP address for acp_1:  192.168.10.10
  Processes:       80                 IP address for acp_2:  192.168.10.10
  Users logged in: 0                  IP address for acp_3:  192.168.10.10

  Graph this data and manage this system at:
    https://landscape.canonical.com/

New release '16.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

root@Node0(205.172.170.6) # echo test
test
root@Node0(205.172.170.6) # █
```

```
Name: hadoop_1_node
CP: ExoGENI
DC: UMass (UMass Amherst, MA, USA) XO Rack
CPU: 1vCPU
MEM: 1G
Status: running
```

FIGURE 7 Example infrastructure topology from the CloudsStorm graphical user interface [Colour figure can be viewed at wileyonlinelibrary.com]

better performance. After executing the task, the result is stored in the corresponding folder of HDFS. Extra VMs, which offer the computing resource, therefore can be released to save cost.

Figure 7 is a snapshot to demonstrate the web-based GUI offered by the "*Control Agent*" for this example. In this case, the "*Control Agent*" is also dynamically set up on the "UvA" data center, and its public IP is "145.100.133.147". Being aware of this public IP, the developer is able to access this GUI from the browser. Figure 7 shows the networked infrastructure description. It illustrates how "Node0" constructs the original one-node Hadoop cluster. At the time of the snapshot, three scaled VMs, ie, "Node0S0", "Node0S1", and "Node0S2", are also shown in the GUI. Especially, the network connection is also scaled according to the original VM "Node0". In addition, the size of the circle, which represents the VM, is proportional to its CPU and Memory capacity. The color of the circle identifies which subtopology this VM belongs to. The VM type and location (cloud and data center) information can also be popped up when hovering over the circle. It is also worth mentioning a terminal of a specific VM can directly pop up through double-clicking the corresponding circle, working as a web terminal. This terminal is convenient to check the result on a remote VM. The detailed infrastructure topology descriptions and code can be downloaded,[***] due to the space limitations of this paper.

4.2.2 | Evaluation

In this section, we conduct several experiments on this case study to evaluate CloudsStorm. We execute the word count program on a Hadoop cluster with different scales and different input data size. There are two groups of experiments. One is performed by a traditional word count program with a Hadoop cluster of fixed scale. The other one is performed by the newly developed word count program with the data-aware processing function similar to that shown in Listing 1. Each experiment is repeated five times.

Figure 8A shows the execution time of different phases of data uploading and processing, according to different input data sizes and cluster scales. It demonstrates that the data uploading time is not related to the cluster scale. On the other hand, the data processing time decreases with the increase of the cluster scale. However, after a specific cluster scale, the processing time no longer decreases so dramatically. It is therefore essential to customize the infrastructure computing capacity properly according to the input data size. Through leveraging CloudsStorm, the infrastructure can be

[***]https://github.com/CloudsStorm/ExampleRepo/releases/download/hdp/HadoopDataAwareProcessing.zip

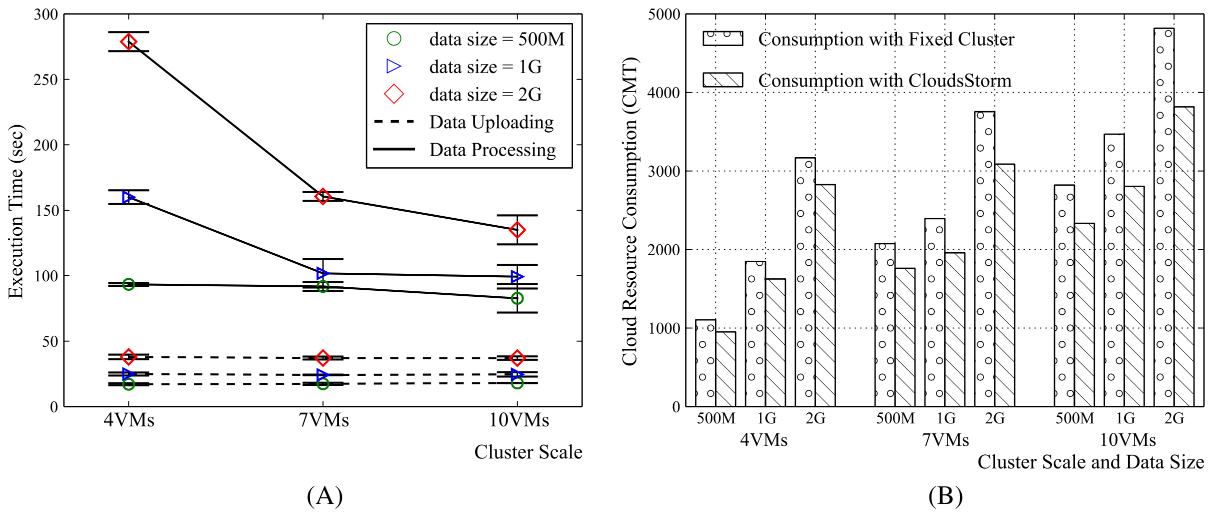


FIGURE 8 Data-aware processing evaluation using the CloudsStorm framework. A, Execution time in different phases; B, Cloud resource consumption comparison. VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

programmed to dynamically adjust the input data size. Compared to the traditional application case, redundant computing resources can be avoided.

Figure 8B demonstrates that the cloud application programmed with the CloudsStorm framework is more efficient on the cloud resource consumption. The cloud resource consumption is calculated as the summation of resource consumption of each VM. The consumption of a VM is calculated as the product of that VM's CPU (vCPU numbers), memory (in GigaBytes), and the corresponding task execution time (in seconds). Hence, the resource consumption is termed as “CMT” and it is in proportion to the monetary cost for using the cloud. This figure shows that it is always beneficial and saves the costs when adopting CloudsStorm to develop the application, because the cluster contains only one VM in the input data uploading phase. Compared to the traditional application execution case, the cluster needs to be set up and configured in advance, and the extra computing resource is wasted during the data preparation phase, which is more concerned with I/O and storage resources. Although some clouds provide a vendor lock-in solution to define a policy on how to scale, this indicates a lack of programmability and controllability at the infrastructure level. It means that the solution cannot make the infrastructure adjusted to the application in a more fine-grained way. For example, in this case, the infrastructure can only be scaled after the data preparation phase instead of making these two operations in parallel. Therefore, this case study demonstrates the cloud application developed with CloudsStorm achieves more programmability and fine-grained controllability on its virtual infrastructure.

4.3 | Controllability performance evaluation

In this section, we evaluate the controllability performance of CloudsStorm, including autoscaling, failure recovery, etc.

4.3.1 | Autoscaling and failure recovery

Autoscaling and failure recovery are the key controllability of the infrastructure provided by CloudsStorm. We first design an experiment on ExoGENI to test the autoscaling performance. In this experiment, there are initially two subtopologies, ie, $subNI_1$ containing one VM and $subNI_2$ containing eight “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 out 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” VM in $subNI_1$ can always know where the scaled resources are. Figure 9A illustrates that we scale out the eight VMs of $subNI_2$ accordingly 1, 2, 3, 4, 8, and 16 times. Each scaled $subNI_2$ is provisioned from independent data centers simultaneously. The time for provisioning each scaled subtopology can be defined as T_i, where $1 \leq i \leq 16$ and $i \in \mathbb{N}$. The flat dashed line represents the ideal scaling performance in theory. This theoretical performance refers to the entire provisioning performance with the theoretical assumption that all the data centers have the same provisioning performance and there is no interference in between. It means that the time for scaling different subtopologies is same and a constant value t,

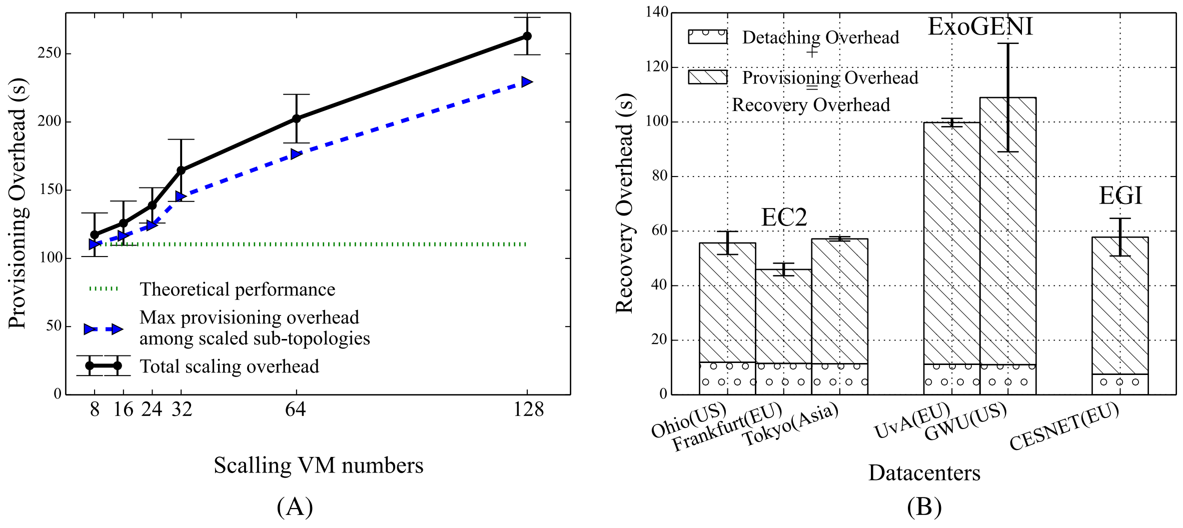


FIGURE 9 Controllability performance evaluation. A, Horizontal scaling out performance; B, Failure recovery performance. EC2, Amazon Elastic Compute Cloud; EGI, European Grid Infrastructure; VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

ie, $T_i = T_j = t$, for $\forall i, j \in \mathbb{N}$ and $1 \le i,j \le 16$. It is then obvious to derive that $\max_{1 \le i \le 16} T_i = t$. Therefore, no matter how many VMs need to be provisioned, as long as all the subtopologies (each subtopology contains eight VMs) are in different data centers, the entire provisioning overhead should remain the same. 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. Hence, the more copies of $subNI_2$ requested, the more connections need to be configured. The solid line in the figure shows the 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 efficient to achieve large-scale autoscaling. In addition, most clouds have limitations on resource allocation. For instance, ExoGENI only allows 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.

Figure 9B shows the experimental result on failure recovery. In this experiment, there are still two subtopologies in the beginning, ie, $subNI_1$ and $subNI_2$. Each of them contains only one VM, ie, n_1 and n_2. These two nodes are connected with a private network. We then assume the case where the data center of $subNI_2$ is not available. CloudsStorm recovers the same subtopology from another data center or cloud. Finally, the private network is reconstituted. 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 measured time duration is the total recovery overhead. We conduct this experiment on three clouds currently supported and pick six data centers from them. In order to compare, n_2 always has two cores and around 8 GB memory with Ubuntu 14.04 installed. Correspondingly, they are instances of the "t2.large" of EC2, "XOLarge" of ExoGENI, and "mem_medium" of EGI VM configurations. The results show that ExoGENI has a relatively 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. This kind of information is important when deciding where to recover, to satisfy the application QoS, considering the recovery overhead and data center geographic information.

4.3.2 | Comparison with related tools

Finally, we conduct a set of experiments to compare CloudsStorm with other DevOps tools. We pick jclouds from the set of API-centric tools. jclouds is adopted by a lot of environment-centric tools to be the basic provisioning engine, such as CloudPick.[13] From the set of environment-centric tools, we pick Nimbus.[14] team's cloudinit.d[†††] Other tools like Juju and infrastructure manager (IM) provide graphical interfaces, which make it difficult to measure performance. Both jclouds

[†††]http://www.nimbusproject.org/doc/cloudinitd/latest/

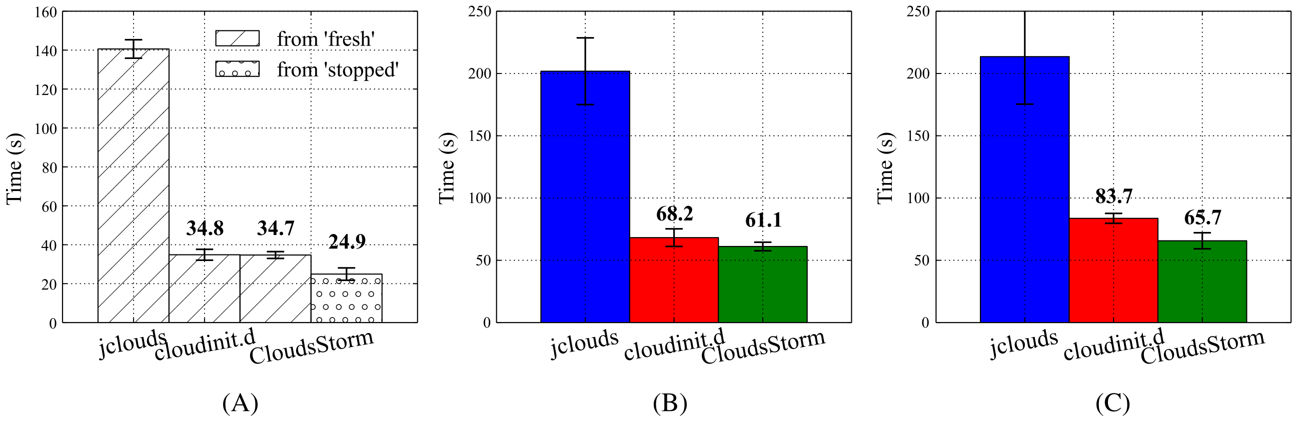


FIGURE 10 Performance comparison. A, Scaling; B, Including deployment; C, Deployment with dependency [Colour figure can be viewed at wileyonlinelibrary.com]

and cloudinti.d do not support networked infrastructure. The ones who support networked infrastructure can only be applied in private data centers, which CloudsStorm cannot have the access permission, like SAVI. We pick EC2 to conduct 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 the California data center. However, as jclouds and cloudinit.d cannot directly support autoscaling behavior, we use them to provision five new VMs in California data center for the assumption of this scenario. Each operation, we repeated 10 times. Figure 10A illustrates the results. For jclouds, the provisioning process proceeds in sequence; hence, its scaling overhead is much larger than the other two. If only considering the scaling performance from "Fresh" (defined in Section 3.1) state, cloudinit.d and CloudsStorm have similar performance, demonstrated by the bars covered with slashes. CloudsStorm is a little bit more stable than cloudinit.d. Moreover, EC2 supports stopping an instance. CloudsStorm can perform autoscaling from "Stopped" status. It reduces the overhead, shown by the bars covered with dots. It is worth mentioning 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, reducing the scaling overhead by more than half referring to Figure 10B and Figure 10C.

The second experiment is to compare the provisioning performance including deployments. All three of our chosen systems allow users to define a script to deploy applications immediately after provisioning. In this experiment, we chose the California data center to provision 5 "t2.micro" VMs and install Apache Tomcat on each of them. Each test is repeated 10 times. Figure 10B shows the results. With jclouds, the applications 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 the deployment dependency. In this experiment, four out of five VMs install Tomcat 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 the MySQL VM to know the server address. Hence, jclouds needs to provision the 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 predefined before actual provisioning. Therefore, all the deployments can proceed simultaneously even with the dependency. Figure 10C demonstrates that the deployment dependency has a smaller influence on the performance of CloudsStorm 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.

5 | RELATED WORK

In order to mitigate the difficulty of virtual infrastructure management for cloud applications, there has been substantial academic research and several industrial tools developed in recent years. In the academic area of cloud computing, there has been plenty of papers working on how to control the infrastructure to satisfy the application requirements. For example, Venkateswaran and Sarkar[15] proposed a model to deploy applications on hybrid clouds and yield the best-fit hosting combination. Ziafat and Babamir[3] focused on the algorithm selecting a proper cloud to run the task according to the geographical location of the data center. Fu et al[16] put forward eight recovery patterns for sporadic operations on public cloud to maintain the service quality of the application. Wang et al[17] tried to optimize the makespan and the reliability of a workflow application through evaluating the resource reputation. Ilyushkin et al[18] conducted evaluations on the scaling policy for the workflow. However, these works mainly focus on how to adjust the infrastructure instead of providing the actual ability for the application to control the infrastructure itself. They evaluate their methods through simulation or manual configuration. More controllability and programmability are therefore required for the application to control the infrastructure. On the other hand, there are also some existing academic works to enhance the controllability and programmability of the cloud virtual infrastructure. CodeCloud[19] consists of an IM.[20] Infrastructure manager provides some specific REST APIs to control each individual VM. Based on this, CodeCloud leverages to describe the application and the elasticity of the infrastructure. CloudPick[13] 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. mOSAIC[7] is a deployable platform providing model-driven cloud application development. CometCloud[6] provides a heterogeneous cloud framework to deploy several programming models, such as master/worker, map/reduce, and workflow. A video analytics system[21] is a notable application scenario of CometCloud. However, these tools need infrastructure resources provisioned in advance, which does not have controllability on the underlying infrastructure at runtime. Koulouzis et al[22] leveraged the programmability of the network to optimize the data transfer, but the control on the switch of the data center is required, which is not practical for cloud computing from the customer perspective.

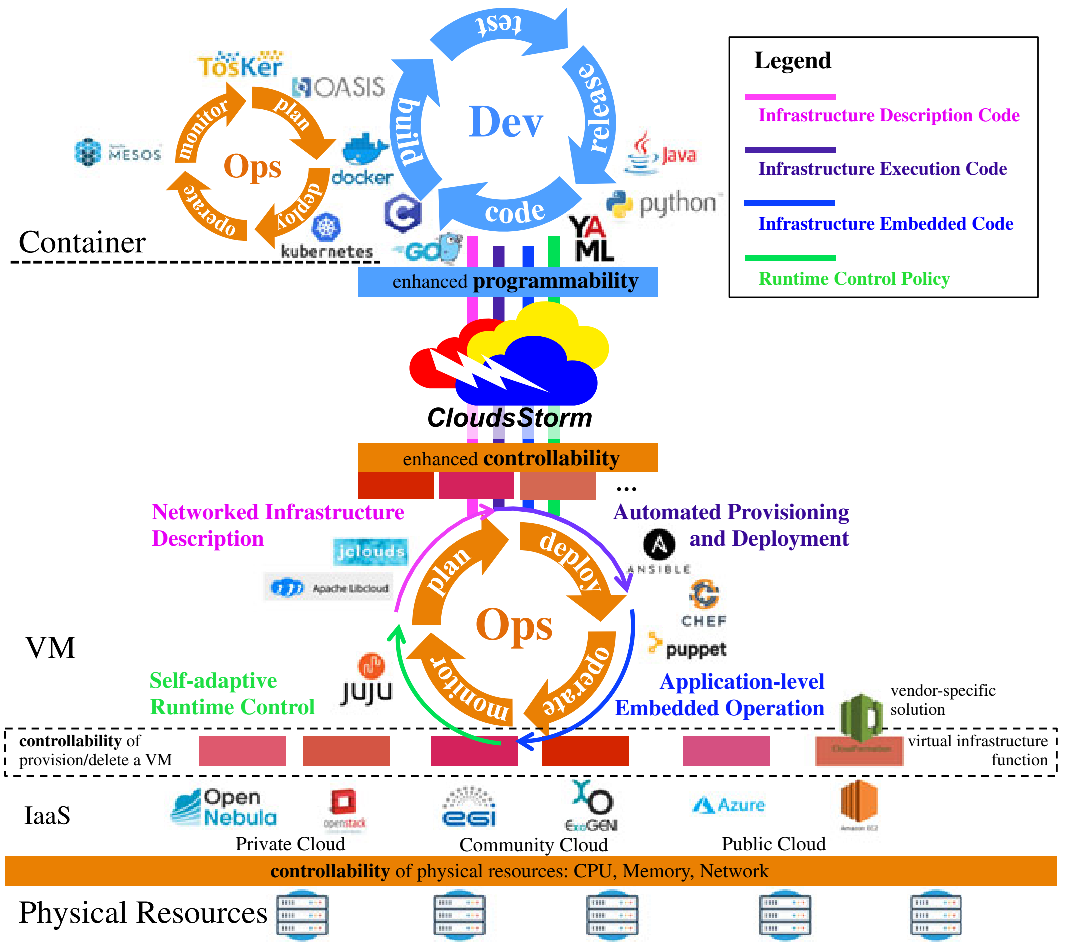


FIGURE 11 The infrastructure programmability and controllability provided by the CloudsStorm framework for DevOps and comparison with related tools. IaaS, infrastructure as a service; VM, virtual machine [Colour figure can be viewed at wileyonlinelibrary.com]

From the perspective of industry, there are also many DevOps tools or frameworks proposed for the cloud application to manage the infrastructure. Figure 11 illustrates the position and the aspect which they mainly focus on during the DevOps lifecycle. In the IaaS cloud computing model, different clouds or tools, such as OpenNebula or OpenStack, provide different VIFs, ie, APIs, to access their physical resources. This basic controllability of provisioning or terminating a VM is leveraged to afford programmability for the higher-level application. These VIFs are illustrated as blocks with different colors. In the "Dev" cycle, tools such as Libcloud and jclouds, unify provisioning APIs from several clouds to empower controllability on the individual VM. However, configuration and management on the entire infrastructure are still done manually. In order to manage VMs of a cluster, some clouds provide a tool to describe several VMs, eg, CloudFormation[‡‡‡] of EC2 provides the programmability to describe the infrastructure only consisting of EC2 VMs. To avoid the vendor lock-in problem and manage the federated cloud virtual infrastructure, tools like Chef, Ansible, and Puppet, etc, are developed, which provide programmability to describe infrastructure from different clouds. Among these tools, Chef and Ansible more focus on application deployment and configuration. They standardize the configuration commands among different systems to make the code reusable, such as the cookbook of Chef and playbook of Ansible. This code of unified description and configuration is proposed as "Infrastructure as Code".[23] However, it cannot describe infrastructure operations. Anyhow, all these three tools more focus on the infrastructure itself and try to make the operation simple at runtime, instead of narrowing the gap to make the application aware of the infrastructure. Juju takes one step further to realize an application-defined infrastructure through describing application components and their hosting VMs, where the components are some typical software modules. The controllability on the infrastructure, like autoscaling, can also be leveraged to ensure the QoS of a certain software component. However, this model-driven approach can only provide a way to describe the relationship between application components and the infrastructure. It lacks more fine-grained infrastructure programmability and controllability to be embedded inside the arbitrary code for a cloud application. Besides, Puppet and Chef adopt Ruby as the domain-specific language, which requires knowledge about the Ruby programming language.

Another aspect of work only focuses on the programmability and controllability between the container level and the application level, which is illustrated as the upper yellow "Dev" circle in Figure 11. If treating containers as the infrastructure, containers are more flexible and lightweight to provide the infrastructure controllability, and programmability compared to VMs. In particular, Docker[§§§] performs operating-system-level virtualization to make the container closer to a lightweight VM. Kubernetes[†] and Mesos[¶¶¶] further enhance the programmability to describe dependencies among several containers and orchestrate them to form a virtual cluster. Besides, TOSCA[§], which is also based on YAML, is proposed to standardize the description of dependency among the application components and its underlying infrastructure, eg, the OS of the hosting machine. It more concentrates on the services from the application perspective; for instance, the connection of infrastructure is described as the service dependency. It also more focuses on the static topology description without the ability to define some direct operations on the infrastructure. To enforce this standard, TosKer[24] implements an engine, which regards Docker as the infrastructure. Although the container or docker is able to provide a VM-like environment, it is still more close to software virtualization, eg, it shares the kernel of the host machine. Moreover, it always requires the VM to be the underlying layer in the cloud environment, due to the reason for isolation and security. Therefore, VM is still indispensable. However, these tools at this level are not able to afford further controllability on the lower-level infrastructure. For example, ECSched[25] can only provide the container level of scheduling, without the ability to manipulate the VM.

Other tools or frameworks, such as HTCondor[###] and Open MPI,[‖‖‖] mainly focus on how to conduct applications on a fixed distributed system. They concentrate more on the application perspective, empowering the application to run in parallel and fit the distributed environment. Nevertheless, they are not designed for cloud environments, and therefore lack controllability of the infrastructure. Another option to mitigate the DevOps gap between the application and the infrastructure is to build the entire DevOps stack from the hardware level, which means the infrastructure should be specifically designed. For example, SAVI[26,27] builds up a testbed for internet of things. It leverages OpenFlow to consider the network topology of the virtual infrastructure. Kraken[28] and SWMOA[29] also take the network into account when provisioning virtual infrastructures over multiple private data centers. However, all these solutions are applied to private data centers, where the hardware in the data center, such as switches, routers, etc, must be able to be totally controlled. It

[‡‡‡]https://aws.amazon.com/cloudformation/
[§§§]https://www.docker.com/
[¶¶¶]http://mesos.apache.org/
[###] https://research.cs.wisc.edu/htcondor/
[‖‖‖]https://www.open-mpi.org/

TABLE 2 Functionality comparison among different DevOps tools and frameworks

		Programmability					**Controllability**				
	Main Language	**Infrastructure Description**	**Federated Clouds**	**Networked Infrastructure**	**Public Cloud**	**Provision from scratch**	**Automated Configuration**	**Auto Scaling**	**Failure Recovery**	**Multi Mode**	**Decentra-lization**
jclouds	Java	×	-	×	√	√	√	×	×	×	×
Libcloud	Python	×	-	×	√	√	√	×	×	×	×
Puppet/Chef	Ruby	-	√	×	√	√	√	-	-	×	-
Ansible	YAML	×	√	×	√	×	√	-	-	×	-
SAVI	Not Known	×	√	√	×	√	×	-	-	×	×
Juju	YAML	-	√	×	√	√	√	-	-	×	×
cloudinit.d	Python	×	√	×	√	√	√	-	-	×	×
CodeCloud(IM)	CJDL(XML)	-	√	×	√	√	√	√	×	×	×
CloudPick	GUI	√	√	×	√	√	√	-	√	×	×
CometCloud	Java	√	√	×	√	×	×	×	×	×	-
mOSAIC	Java	√	√	×	√	×	√	√	√	×	×
CloudsStorm	YAML	√	√	√	√	√	√	√	√	√	√

Abbreviations: CJDL, cloud job description language; GUI, graphical user interface; YAML, YAML Ain't Markup Language.

is obvious that these solutions are not feasible in the context of public clouds, which can only afford some limited general controllability on the physical hardware infrastructure.

Finally, we pick several tools from related ones. Table 2 shows the functionality comparison among different related tools and frameworks. These functionalities are related to the three levels of programmability and two types of controlling modes we propose. Here, "Infrastructure Description" is related with the design-level programmability. "Federated Clouds" refers to whether multiple clouds are supported. "Networked Infrastructure" is used to represent whether the infrastructure can be defined and provisioned with a private network. "Public Cloud" is to indicate whether the public cloud is supported or on its own testbed. "Provision from scratch" is related with the infrastructure-level programmability. It refers to the ability to execute some infrastructure operations programmatically. "Automate Configuration", "Autoscaling", and "Failure Recovery" are the controllability features. "Multimode" indicates whether both of the controlling modes are supported, which are explained in Section 2.3. "Decentralization" demonstrates the infrastructure management is in a decentralized way. If it is not, it means all the application developers' infrastructures are managed by one administration, which requires everyone's cloud credentials. In this table, "×" and "$\surd$" means supported and not supported, respectively. "−" indicates partial support. For instance, some tools only provide GUI for infrastructure description instead of languages, which is identified as "−". Some tools partially support autoscaling and failure recovery, because some manual work is still needed actually to trigger the process. Another case is where the autoscaling refers to configuration scaling on pre-existing resources instead of provisioning new ones. The partial support for the "Decentralization" of controllability means an application developer can set up his own server for managing all his infrastructures instead for one application.

It is also worth mentioning that CloudsStorm is integrated as a provisioning component of the software release of European project software workbench for interactive, time critical, and highly self-adaptive cloud applications (SWITCH).[30] Therefore, the software provided by SWITCH project is not compared here.

6 | CONCLUSION

This paper has presented CloudsStorm, a framework for seamless programming and operating VIF during the DevOps lifecycles of cloud applications. Figure 11 also illustrates the controllability and programmability provided by CloudsStorm, comparing to other tools among different levels of DevOps stack. It shows that CloudsStorm is mainly proposed to work at the VM level. First, CloudsStorm provides more design-level programmability to describe the federated infrastructure, shown as the vertical pink line below CloudsStorm in Figure 11. Notably, the "*Infrastructure Description Code*" of CloudsStorm enables the ability to describe the networked infrastructure, which makes the infrastructure transparent to the application. Second, the "*Infrastructure Execution Code*" is proposed to provide the infrastructure-level programmability. It corresponds to the vertical purple line below CloudsStorm in Figure 11. This part of code not only provides the static description but also can be executed. Third, the "*Infrastructure Embedded Code*" is proposed to empower the more fine-grained infrastructure programmability within the arbitrary code for general purposes, instead of at the application module level. It corresponds to the vertical blue line below CloudsStorm in Figure 11. In addition, we propose two types of control mode, ie, passive and active mode, to achieve more comprehensive controllability for the application to manage its infrastructure. The "*Runtime Control Policy*" is proposed for this purpose, which refers to the vertical green line below CloudsStorm in Figure 11. In order to implement the framework, we put forward the network connection method[4] to construct the networked infrastructure and develop an "*Infrastructure Execution Engine*". This engine is designed to be extensible to easily plug in a new cloud, which only requires basic functions of how to provision and terminate a VM from that cloud. The dynamically provisioned "*Control Agent*" also leverages this engine at runtime to perform infrastructure operations, which are programmed at the development phase. Moreover, one "*Control Agent*" is designed to be responsible for only one application-defined infrastructure. It therefore works in a decentralized way to be more efficient, instead of acting as a centralized infrastructure management framework for many applications. Besides, the GUI provided by the "*Control Agent*" affords an intuitive way to check and access cloud resources. Finally, it is worth mentioning that YAML format is broadly adopted for the syntax of code in CloudsStorm. Benefiting from the straightforward YAML format, the syntax of CloudsStorm is easy to learn and program with, especially for the definition of parallel operations.

To demonstrate the CloudsStorm framework, we present two case studies, which show how to migrate task-based applications and data processing applications onto real clouds within CloudsStorm. The former one focuses on the infrastructure-level programmability, which mainly harnesses the "*Infrastructure Execution Code*" to control the infrastructure seamlessly. The latter one more concentrates on the application-level programmability, which mainly leverages the "*Infrastructure Embedded Code*" to make the underlying infrastructure data-aware. The evaluation results of both case studies demonstrate cloud applications developed and executed within CloudsStorm are efficient concerning cloud resource usage when compared against traditional manual or fixed infrastructure configurations. The efficiency of CloudsStorm is achieved through the infrastructure programmability and controllability it offers to allow for on-demand and parallel infrastructure management. In the end, we conduct some experiments to evaluate the controllability performance of CloudsStorm, including autoscaling and failure recovery. We also compare its performance with some other tools. Because of the parallel operation it provides, the experimental results demonstrate that the controllability performance of CloudsStorm outperforms others. In conclusion, CloudsStorm enhances infrastructure programmability at the cloud application development phase and meanwhile improves the infrastructure controllability for the application at runtime during the infrastructure operation phase. Besides, we can derive that CloudsStorm is able to preserve the application QoS because of (1) the efficiency of the control operation itself. (2) The application is able to leverage "*Infrastructure Embedded Code*" to program the infrastructure according to the application requirement, eg, customize the infrastructure capability according to the input workload or provision the computing resources (VMs) close to the data source. (3) The infrastructure is able to be managed according to the application-defined "*Runtime Control Policy*". The operation on the infrastructure can be passively triggered through resource monitoring. The application QoS is therefore able to be kept satisfied when the VM resources are not sufficient or even fail.

For future work, there are two directions. One is going up to construct more complex functionalities to achieve some controllability at a higher level. It is useful to simplify the operation on the infrastructure for the application to leverage. We are therefore going to demonstrate how to migrate different applications onto IaaS clouds using CloudsStorm. The goal is to simplify the programming complexity and reduce monetary cost by directly control the cloud infrastructure on demand. Another direction is to support container level controllability. Because there is a trend to wrap the application as a container, especially for web services, the controllability on containers would enable more fine-grained controllability for the application to exploit its infrastructure.

ACKNOWLEDGEMENTS

The author would like to thank Paul Martin for proofreading. This research has received funding from the H2020 European Institute of Innovation and Technology of European Union's Horizon 2020 research and innovation program under grant agreements 643963 (SWITCH project), 654182 (ENVRIPLUS project), and 676247 (VRE4EIC project). The author, Huan Zhou, was also sponsored by China Scholarship Council.

ORCID

Huan Zhou https://orcid.org/0000-0003-2319-4103
Zhiming Zhao https://orcid.org/0000-0002-6717-9418

REFERENCES

1. Wettinger J, Breitenbücher U, Kopp O, Leymann F. Streamlining DevOps automation for cloud applications using TOSCA as standardized metamodel. *Futur Gener Comput Syst*. 2016;56:317-332.
2. Wang J, Taal A, Martin P, et al. Planning virtual infrastructures for time critical applications with multiple deadline constraints. *Futur Gener Comput Syst*. 2017;75:365-375.
3. Ziafat H, Babamir SM. Optimal selection of VMs for resource task scheduling in geographically distributed clouds using fuzzy c-mean and MOLP. *Softw: Pract Exper*. 2018;48:1820-1846.
4. Zhou H, Wang J, Hu Y, et al. Fast resource co-provisioning for time critical applications based on networked infrastructures. Paper presented at: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD); 2016; San Francisco, CA.
5. Androulaki E, Barger A, Bortnikov V, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the 13th EuroSys Conference (EuroSys); 2018; Porto, Portugal.
6. Diaz-Montes J, AbdelBaky M, Zou M, Parashar M. CometCloud: enabling software-defined federations for end-to-end application workflows. *IEEE Internet Comput*. 2015;19(1):69-73.
7. Petcu D, Di Martino B, Venticinque S, et al. Experiences in building a mOSAIC of clouds. *J Cloud Comput Adv Syst Appl*. 2013;2(1):12.
8. Nastic S, Sehic S, Le D-H, Truong H-L, Dustdar S. Provisioning software-defined IoT cloud systems. Paper presented at: 2014 International Conference on Future Internet of Things and Cloud; 2014; Barcelona, Spain.
9. Faure G, Miquel A. *A Categorical Semantics for the Parallel Lambda-Calculus*. Research report. INRIA; 2009.
10. Taherizadeh S, Jones AC, Taylor I, Zhao Z, Stankovski V. Monitoring self-adaptive applications within edge computing frameworks: a state-of-the-art review. *J Syst Softw*. 2018;136:19-38.

11. Baldin I, Xin Y, Mandal A, Ruth P, Heerman C, Chase J. ExoGENI: a multi-domain infrastructure-as-a-service testbed. In: *Testbeds and Research Infrastructure. Development of Networks and Communities: 8th International ICST Conference, TridentCom 2012, Thessaloniki, Greece, June 11-13, 2012, Revised Selected Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2012:97-113.
12. Zhou H, Taal A, Koulouzis S, et al. Dynamic real-time infrastructure planning and deployment for disaster early warning systems. In: *Computational Science - ICCS 2018: 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part II*. Cham, Switzerland: Springer International Publishing AG; 2018:644-654.
13. Dastjerdi AV, Garg SK, Rana OF, Buyya R. CloudPick: a framework for QoS-aware and ontology-based service deployment across clouds. *Softw: Pract Exper*. 2015;45:197-231.
14. Keahey K, Freeman T. Contextualization: providing one-click virtual clusters. Paper presented at: 2008 IEEE 4th International Conference on eScience; 2008; Indianapolis, IN.
15. Venkateswaran S, Sarkar S. Architectural partitioning and deployment modeling on hybrid clouds. *Softw: Pract Exper*. 2018;48(2):345-365.
16. Fu M, Zhu L, Sun D, Liu A, Bass L, Lu Q. Runtime recovery actions selection for sporadic operations on public cloud. *Softw: Pract Exper*. 2017;47(2):223-248.
17. Wang X, Yeo CS, Buyya R, Su J. Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm. *Futur Gener Comput Syst*. 2011;27(8):1124-1134.
18. Ilyushkin A, Ali-Eldin A, Herbst N, et al. An experimental performance evaluation of autoscaling policies for complex workflows. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE); 2017; L'Aquila, Italy.
19. Caballer M, de Alfonso C, Moltó G, Romero E, Blanquer I, García A. CodeCloud: a platform to enable execution of programming models on the clouds. *J Syst Softw*. 2014;93:187-198.
20. Caballer M, Blanquer I, Moltó G, de Alfonso C. Dynamic management of virtual infrastructures. *J Grid Comput*. 2015;13(1):53-70.
21. Zamani AR, Zou M, Diaz-Montes J, et al. Deadline constrained video analysis via in-transit computational environments. *IEEE Trans Serv Comput*. 2017. Early access.
22. Koulouzis S, Belloum AS, Bubak MT, Zhao Z, Živković M, de Laat CTAM. SDN-aware federation of distributed data. *Futur Gener Comput Syst*. 2016;56:64-76.
23. Morris K. *Infrastructure as Code: Managing Servers in the Cloud*. Sebastopol, CA: O'Reilly Media, Inc; 2016.
24. Brogi A, Rinaldi L, Soldani J. TosKer: a synergy between TOSCA and docker for orchestrating multicomponent applications. *Softw: Pract Exper*. 2018;48:2061-2079.
25. Hu Y, Zhou H, de Laat C, Zhao Z. ECSched: efficient container scheduling on heterogeneous clusters. In: *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings*. Cham, Switzerland: Springer Nature Switzerland AG; 2018:365-377.
26. Kang J-M, Lin T, Bannazadeh H, Leon-Garcia A. Software-defined infrastructure and the SAVI testbed. In: *Testbeds and Research Infrastructure: Development of Networks and Communities: 9th International ICST Conference, TridentCom 2014, Guangzhou, China, May 5-7, 2014, Revised Selected Papers*. Cham, Switzerland: Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; 2014:3-13.
27. Kang J-M, Bannazadeh H, Leon-Garcia A. SAVI testbed: control and management of converged virtual ICT resources. Paper presented at: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM); 2013; Ghent, Belgium.
28. Fuerst C, Schmid S, Suresh L, Costa P. Kraken: online and elastic resource reservations for multi-tenant datacenters. Paper presented at: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications; 2016; San Francisco, CA.
29. Shi W, Wu C, Li Z. An online mechanism for dynamic virtual cluster provisioning in geo-distributed clouds. Paper presented at: IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications; 2016; San Francisco, CA.
30. Štefanič P, Cigale M, Jones AC, et al. SWITCH workbench: a novel approach for the development and deployment of time-critical microservice-based cloud-native applications. *Futur Gener Comput Syst*. 2019;99:197-212.

How to cite this article: Zhou H, Hu Y, Ouyang X, et al. CloudsStorm: A framework for seamlessly programming and controlling virtual infrastructure functions during the DevOps lifecycle of cloud applications. *Softw: Pract Exper*. 2019;49:1421–1447. https://doi.org/10.1002/spe.2741