EMPOWERING DYNAMIC TASK-BASED APPLICATIONS WITH AGILE VIRTUAL INFRASTRUCTURE PROGRAMMABILITY

A PREPRINT

Huan Zhou
Informatics Institute
University of Amsterdam
Amsterdam, 1098 XH
School of Computer, NUDT
h.zhou@uva.nl

Yang Hu
Informatics Institute
University of Amsterdam
Amsterdam, 1098 XH
y.hu@uva.nl

Jinshu Su
School of Computer
National University of Defense Technology
Changsha, 410073
sjs@nudt.edu.cn

Mingmin Chi
Shanghai Key Laboratory of Data Science
Fudan University
China, Shanghai
mmchi@fudan.edu.cn

Cees de Laat
Informatics Institute
University of Amsterdam
Amsterdam, 1098 XH
delaat@uva.nl

Zhiming Zhao
Informatics Institute
University of Amsterdam
Amsterdam, 1098 XH
z.zhao@uva.nl

ABSTRACT

The IaaS (Infrastructure-as-a-Service) offered by Clouds provides applications with the capability of customizing VMs and configuring their network. Compared to traditional service-based IaaS applications such as persistent web services, most task-based applications have a relatively short duration but are triggered on demand. A typical way to support such kinds of application is to provision a shared and fixed virtual infrastructure based on pre-estimated size in advance, and then perform all the processing tasks. However, due to unpredictable workloads, this solution can lead to either cost inefficiency caused by over-provisioning, or failure to deliver the performance required by applications. CloudsStorm is a dynamic control framework proposed to provide applications with agile programmability and flexibility in controlling the virtual infrastructure. With its front end, applications can design their networked infrastructure and program that infrastructure with our interpreted infrastructure code language. With the back-end engine, the infrastructure code can be executed to provision the networked infrastructure, deploy and execute the application to obtain results, and release resources. Moreover, we adopt multi-threading to support parallel operation. Finally, we conduct experiments in an assumed scenario to demonstrate functionalities of CloudsStorm. The evaluation results prove CloudsStorm is efficient for task-based applications that need to exploit Clouds but reduce the monetary cost.

***k*eywords** infrastructure-as-a-service · programmable infrastructure · task-based applications · parallel operation · DevOps

1 Introduction

Cloud environments provides elastic and on-demand services for running distributed applications. Typical Cloud service models include SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service) [1]. Among those services, the IaaS model offers applications a rich capability for configuring virtual machines, networks, and customizing software platforms on top; however, it also requires application developers to have profound technical knowledge about planning and configuring underlying virtual infrastructure, in particular when it has to use resources from different data centers or Clouds. For service-based applications, such difficulties are not obvious, since they typically only require a small or fixed number of VMs. However, it becomes an urgent problem when supporting task-based applications such as on-demand data processing or scientific workflows, which are often highly distributed and do not run persistently in Cloud, but rather on demand.

Scientific applications are typical examples of task-based applications [2], such as earthquake prediction or genome sequence processing. Another example is that of a data processing application that executes data processing tasks to achieve some data analysis results. These applications often share common features: i) they all have obvious steps during runtime, ii) they do not run persistently to wait for requests, and iii) they take some data as input and output the results in the end. To run such applications on Cloud using SaaS or PaaS types of services, applications may be limited by the capability for configuring the platform or environment's required data processing components, in particular when they are inherited from legacy systems. Meanwhile, there might be a trust concern regarding submitting data to a public computing cluster to process data with privacy or security issues. Therefore, an application-defined virtual infrastructure is often required by such applications. By using the Cloud IaaS, application developers need provision them from certain providers, configure the required runtime environments including network and manipulate them at runtime. However, most of these operations are done manually.

There have been quite lot of infrastructure provisioning and deployments tools developed in past years, for instance in supporting the DevOps (development and operations) approaches of software engineering [3]. For example, Puppet[1], Chef[2], Ansible[3] are mainly used to automate the deployment and configuration. JuJu[4] is able to automate the provisioning process from Cloud, but it is insufficient for the application to dynamically control the underlying virtual infrastructure. There are also some vendor lock-in solutions provided by Cloud providers, for instance, AWS CloudFormation provided by the Amazon Web Service Cloud. However, the goal of all the tools above is mainly to automate deploying service-based applications on Cloud.

There has been some pioneering exploration in academic research to migrate task-based applications. For instance, [4] and [5] tried to model the Cloud resource performance to mitigate the influence by the performance uncertainty of the Cloud, in order to ensure the quality of service (QoS) for a scientific application. [6] leveraged Cloud to offload some computing tasks of a scientific workflow running on a local cluster. However, in these works, the cloud resources are provisioned in advance and fixed. Application-defined on-demand infrastructure manipulation is limited. None of them provides a mechanism to automate the process of running these task-based applications on Clouds efficiently.

In this paper, we propose an application-defined programmable infrastructure framework, CloudsStorm[5], for IaaS Clouds. It implements an executable language to make the Cloud virtual infrastructure programmable and enable task-based applications to be run on Clouds automatically and efficiently. In order to achieve this goal, the main contributions of this paper are as follows: i) Cloud Virtual Infrastructure Topology Description: this is a YAML (YAML Ain't Markup Language) based language to allow the application describe the underlying networked topology provided by Clouds. ii) Executable Infrastructure Code: this is an interpreted language, which is also based on YAML. The application developer uses it to define how to provision resources provided by Clouds and configure the runtime environment, as well as how to prepare the data, how to execute the application and how to get the results. iii) Backend TSV-Engine: this is the back-end engine to provide all functionalities needed to support executing the infrastructure code. We propose a partition-based virtual infrastructure management mechanism. It divides the cloud virtual infrastructure by data centers and provisions the networked topology across different data centers or Clouds. TSV-Engine also implements multi-threading to control all resources simultaneously. This is also leveraged to support the definition of parallel operations in the infrastructure code.The application is therefore able to control its virtual infrastructure more flexibly and efficiently.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 presents the execution model and related syntax of the programmable infrastructure, which form the front-end. Then we describe the implementation of the back-end TSV-Engine in Section 4. Section 5 demonstrates an application scenario and evaluates some the experimental results. Finally, we conclude and discuss future work in Section 6.

2 Related Work

Before cloud computing, the application developer was faced with a dedicated physical machine or a private cluster consisting of many servers. Cloud computing confers the ability to provision resources on demand. Hence, from the client perspective, the biggest difference brought by Cloud computing is its pay-as-you-go business model. Therefore,

[1]https://puppet.com
[2]https://www.chef.io
[3]https://www.ansible.com
[4]https://jujucharms.com
[5]https://cloudsstorm.github.io/

when we migrate applications onto Clouds, it is justifiable to take the provisioning phase into consideration during the DevOps lifecycle, especially when exploiting the IaaS Cloud model. In order to mitigate the difficulty of this, there has been much academic research and many industrial tools developed in recent years. Libcloud[6] and jclouds[7] are API-centric tools, which mainly wrap up different Cloud providers' APIs to provide a unified way to access Clouds. Application developers still need to integrate this API to manage cloud resources via their own code. Conversely, infrastructure-centric tools such as Puppet, Chef and Ansible adopt the concept of "Infrastructure as Code" [7]. They try to manage the computing infrastructure with some reusable code. However, the code here mainly focuses on configuration and installation, which is not executable. They standardize the configuration commands among different systems to make the code reusable, such as the cookbook of Chef or playbook of Ansible. Juju is application-centric and focuses more on deploying a specific application or a service. Kubernetes[8] is a well known orchestration tool for containers. It schedules and manages the resources to run the application in the form of containers, whereas most Clouds do not directly afford container services due to security and performance isolation issues. In summary, most state-of-the-art work of Cloud applications focuses on either the deployment [8] and execution [9] phase, or the infrastructure planning phase [10] [11]. However, support for the programmability and controllability of the interaction between applications and the actual Clouds hosting their underlying infrastructures is still missing. Such support is required for applications involving dedicated environments to ensure system performance, for example real-time interactive simulation [12].

It is worth mentioning that all the tools and standards above are mainly leveraged for running service-based applications on Clouds. For task-based applications, one solution is to build a specific Cloud to support these applications. For instance, SAVI [13] builds up a test-bed for IoT. It leverages OpenFlow to set up the network topology of the virtual infrastructure. However, this is established on private data centers, which do not belong to the Cloud business model. The other solution is to provision resources in advance and build an agent for each Cloud. For example, CometCloud [14] provides a heterogeneous cloud framework to deploy applications based on several programming models such as master/worker, map/reduce and workflow. However, a lot of manual work is involved in setting up the environment and resources are not provisioned on demand to avoid wastage. It actually builds PaaS on top of IaaS to run task-based applications.

There are some discussions about predicting the performance of the Cloud to ensure the QoS of a scientific application, such as [4] and [5]. However, they do not mention how to automate and orchestrate running an application on Clouds. Qian [6] proposes to offload some scientific computing tasks onto Clouds. Nonetheless, the Cloud resource is provisioned in advance and the application is also manually configured to offload the corresponding tasks. The provisioning gap therefore still exists in the Cloud application DevOps lifecycle, especially for task-based applications.

3 Programmable Infrastructure Front End

In this section, we start introducing the programmable infrastructure front end by illustrating the infrastructure code execution model. It explains how the task-based application is executed on Clouds through CloudsStorm. We then present the syntax of the virtual infrastructure topology description and executable infrastructure code. All these relevant syntaxes are based on YAML because YAML is human-readable and easy to learn.

3.1 Execution Model

Figure 1 illustrates the execution model of the programmable infrastructure. It demonstrates how application developers leverage the developed "Infrastructure Code" to run their applications on Clouds. For the first step, "Infrastructure Code" loads its required virtual infrastructure from the application-defined "Topology Description", including the number of the VMs, network connections, the Clouds or data centers used to run the application, etc. The "Cloud X" information is then updated by querying the "Cloud Database". "Cloud X" represents a Cloud defined in the topology description, of which multiple Clouds may be leveraged. It includes all the relevant data center information for these Clouds, since each data center from a different Cloud will have its own representation of the resources desired. The content is shown in the example of Section 5. This part is provided by CloudsStorm. Application developers only need to afford their credentials to access these Clouds in the application-defined "Cloud Credential". After loading the credential information, "Infrastructure Code" is able to provision the computing resources (VMs) provided by the application-defined Clouds, specifically from the specified data centers based on their geographical locations. Meanwhile, the application-defined network topology and runtime environment required by the application is automatically

[6]http://libcloud.apache.org
[7]https://jclouds.apache.org
[8]https://kubernetes.io

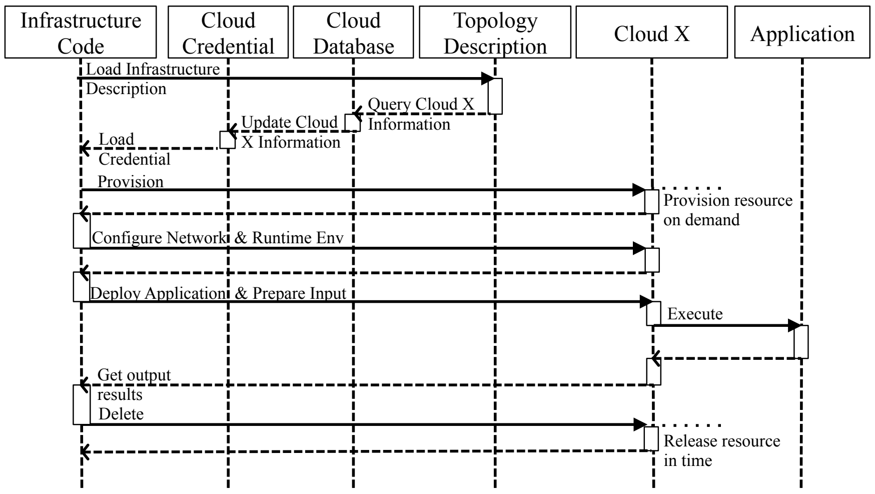
Infrastructure Code
Cloud Credential
Cloud Database
Topology Description
Cloud X
Application
Load Infrastructure Description
Query Cloud X Information
Update Cloud X Information
Load Credential
Provision
Provision resource on demand
Configure Network & Runtime Env
Deploy Application & Prepare Input
Execute
Get output results
Delete
Release resource in time

Figure 1: Sequence Diagram of Infrastructure Code Execution Model

configured. For the next step, application developers can choose where and how to deploy their applications. In addition, the input data can also be prepared. Via the "Infrastructure Code", application developers also define when and how to execute their applications with the input data. Afterwards, application developers are able to leverage "Infrastructure Code" to download the output results from Clouds. Finally, resources can be deleted to release them in time.

Applications are therefore able to efficiently leverage Clouds' computing capability 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 the resource the more you need to pay. Hence, our programmable infrastructure framework is an efficient way to use the Cloud without wasting budget on reservation in advance or occupying the resource for a longer duration. Detailed syntax of the relevant descriptions used by CloudsStorm are presented in following subsections.

3.2 Topology Description Language

Figure 2 shows an example topology to demonstrate the partition-based infrastructure management mechanism. We classify the application-defined topology description into three levels. The lowest level is the VM level. It describes the types of VMs required, mainly referring to the computing capacities, CPU, memory, etc. The level in the middle

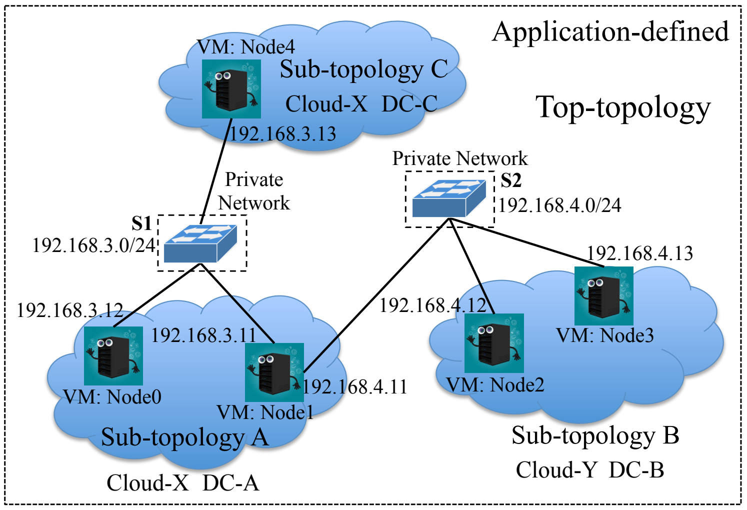

Figure 2: Partition-based Infrastructure Management Example

Syntax 1 Sub-topology Description

```
VMs:
- name: $Node
  nodeType: $Type
  OSType: $OS
  script: $Path
- *
```

is the sub-topology level. It includes descriptions of several VMs in one data center and also describes the Cloud provider from which this data center comes from. The top level is the top-topology level. It includes all the sub-topologies and describes the network connections among these VMs. Internally, the network is defined as a private network, being that it is useful for the application to define the topology as such during the design phase. The actual network and geographical information is then transparent to the application. Otherwise, the IP addresses would need to be dynamically configured after provisioning or there must be an orchestrator to broker communications among sub-topologies, because the public IP address for each VM from the public Cloud is different every time. In addition, there is no actual switch placed among VMs but demonstrating how VMs are connected. The detailed technique to provision such networked infrastructure is demonstrated in our previous work [15].

In order to describe the networked infrastructure for the application, we adopt the YAML format to define the sub-topology and top-topology description. Syntax 1 defines the sub-topology description. This is a list of “VMs”. Every element contains the VM name, such as “Node0” or “Node1” in Figure 2. “nodeType” indicates the computing capacity of the VM, such as “t2.small” or “t2.medium” for the EC2 Cloud. “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. It is worth mentioning that we use symbol ‘-’ to start with a list and ‘$*$’ to represent repeating the element above. Symbol ‘$’ expresses an application-defined variable.

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 sub-topologies defined in “topologies”. “topology” therein is the application-defined name of a certain sub-topology, such as “A”, “B” or “C” shown in Figure 2. “cloudProvider” and “domain” specify the concrete data center where this sub-topology is hosted. For example, there is a $\$DC = California$ data center from $\$Cloud = EC2$. “status” indicates the status of this sub-topology. They are used for the resources lifecycle management. Another list in top-topology definition is “subnets”, including all the private subnets required by the application in the top-topology. In each subnet, there is 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 sub-topology name and the node name itself. For instance, there are two subnets shown in Figure 2, with the “name” of “S1” and “S2”. Taking the example of subnet “S1”, it has $\$subnet = 192.168.3.0$ with $\$netmask = 24$ and it contains three members “A.Node0”, “A.Node1” and “C.Node4” with the corresponding private addresses. This

Syntax 2 Top-topology Description

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

topology description is compatible with the TOSCA standard, the difference being that TOSCA mainly describes service dependencies. The following infrastructure code definition is out of scope for TOSCA however.

3.3 Infrastructure Code

Based on the above application-defined topology description, application developers can further develop the infrastructure code to execute and run their applications on Clouds. The infrastructure 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 types of code. 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 for a certain time 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' or 'get'. They are specified in the field "Operation". Field "Command" is optional. When the operation is 'provision' or 'delete', which means provisioning or deleting Cloud resources, this field is not necessary. When the operation is 'execute', the $\$String$ of "Command" is the specific command to be executed on the VM. When the operation is 'put' or 'get', which means uploading or downloading some files from remote resources, $\$String$ is expressed as 'src:=$\$Path_s$::dst:=$\$Path_d$'. It refers to the file path from the source to the destination according to whether it is uploading or downloading. Field "ObjectType" indicates whether this operation is operated on a 'SubTopology' or on a individual 'VM'. "Objects" then refers to the objects set. To define this set, we adopt the symbol '‖' from parallel lambda-calculus [16] to express parallel operation, such that all "Objects" are operated in parallel, improving the operation efficiency. In summary, 'provision' and 'delete' operations are leveraged to acquire and release Cloud resources; 'execute' operation is used to execute the application; 'put' and 'get' operations are mainly used to prepare data and retrieve results.

Syntax 3 "SEQ" Infrastructure Code

```
- CodeType: 'SEQ'
  OpCode:
    Operation: 'provision | delete | execute | put | get'
    [Command: $String]
    ObjectType: 'SubTopology | VM'
    Objects: $Object_1 [‖$Object_2]...
```

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 condition for exiting a loop. "Count" is defined as 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. It is represented in Unix timestamp. There must be at least one condition defined for a "LOOP" code. If there are several defined, then the loop is ended when one of the conditions is met.

Syntax 4 "LOOP" Infrastructure Code

```
- CodeType: 'LOOP'
  [Count: $num]
  [Duration: $time_1]
  [Deadline: $time_2]
  OpCodes:
  - Operation: 'provision | delete | execute | put | get'
    [Command: $String]
    ObjectType: 'SubTopology | VM'
    Objects: $Object_1 [‖$Object_2]...
  - *
```

The final infrastructure code is therefore an ordered combination of these codes. We omit the detailed operation definitions and define the rest as follows:

```
- CodeType: 'SEQ | LOOP'
- *
```

An example application scenario to demonstrate this infrastructure code is provided in Section 5.

4 Implementation and Back End

Apart from the front-end syntax definitions, there must be a back-end engine to support interpreting the application-defined infrastructure code and operating on its networked virtual infrastructure. Hence, we implement TSV-Engine to be the fundamental execution engine in our programmable infrastructure framework. It is realized according to our partition-based infrastructure management mechanism mentioned in Section 3.2. For choosing implementation language, we adopt Java, which is a platform independent language. Benefiting from this, our infrastructure code is portable and able to be executed on different operating systems. Afterwards, we describe TSV-Engine, as well as some necessary required information, which act as libraries. They include Cloud credential information and the Cloud database. Last but not least, we also explain the essential logging component.

4.1 TSV-Engine

TSV-Engine is the elementary engine used to complete the execution procedure demonstrated in Figure 1. (It has also been integrated in the software release of the EU project SWITCH [17]). It is responsible for interpreting the infrastructure code, provisioning the application-defined virtual infrastructure among Clouds. Especially, it includes provisioning the corresponding private network. "TSV" is short for "Top-topology", "Sub-topology" and "VM" as shown in Figure 3. 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 sub-topologies. T-Engine takes the interpreted operations as input requests to provision or delete the corresponding Cloud resource. According to the execution model in Section 3.1, 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, for example, "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 credential is provided by the application. 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 our framework, it is also responsible for building up the Virtual Network Functions (VNFs) on each VM to provision the networked infrastructure required by the application. Currently, we implement VNFs based on the tunnelling technique. This was proposed by our previous work [15] 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. Different customized V-Engines can be inherited from the basic V-Engine class depending on the VM's features—for example "V-Engine-ubuntu" for an Ubuntu VM. If the application has specific operations on some VM, it can customize its own V-Engine.

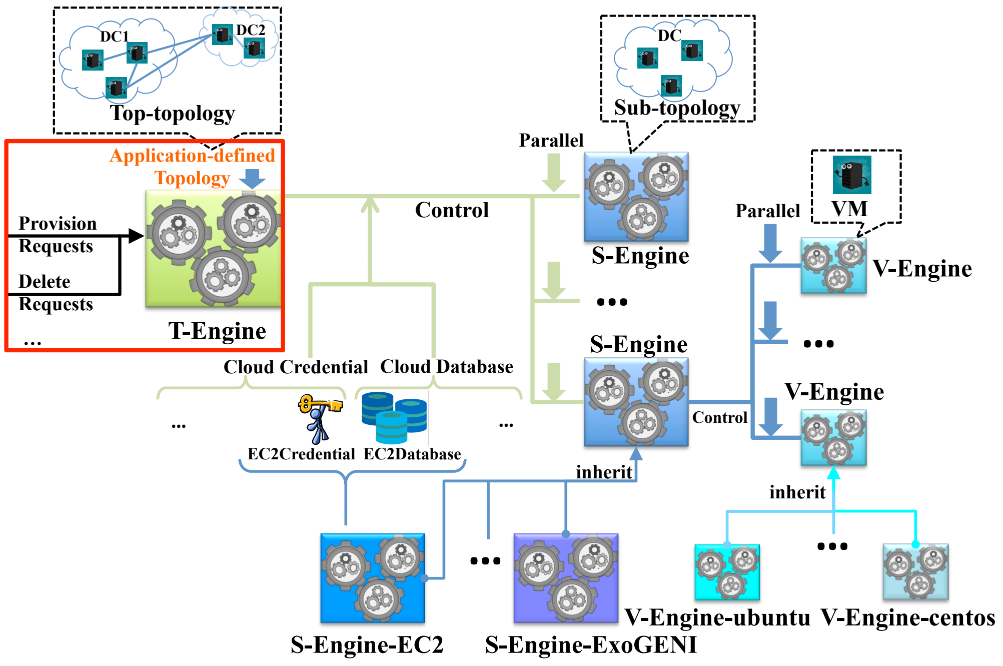

Figure 3: Architecture of TSV-Engine

In addition, a new Cloud or the application's own cluster can also be supported through deriving its own V-Engine. Our programmable infrastructure framework can therefore be easily extended to support different Clouds.

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 sub-topologies 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: the operations on all the VMs in one sub-topology can proceed simultaneously. This mechanism is leveraged to realize the parallel symbol '||' in multiple "Objects" definition in Section 3.3. In other words, TSV-Engine can accelerates operations to reduce the total time needed by an application utilizing the Cloud, reducing the application's total cost as paid to Cloud providers.

4.2 Relevant Libraries and Logging

As discussed above, there are two essential "libraries" needed for executing infrastructure code. One of them is the Cloud database. It contains detailed information about data centers for selected Clouds. This information includes: i) Geographic positioning of each data center, which can be leveraged to do locality-aware or data-aware provisioning. ii) Endpoint information, which describes a data center's controller URL needed for actual provisioning. iii) VM types (CPU, memory) supported in each data center and their characteristic data (e.g. 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 secure tokens needed to access a Cloud or the file paths for Cloud credential files. For instance, two tokens, "accessKey" and "secretKey", are needed for accessing EC2. Hence, the credentials are given by the application during runtime. It avoids the privacy issue with sending Cloud credentials to a third proxy broker to do provisioning. Both of these two libraries are organized in YAML format.

Finally, we build a logging component in our programmable infrastructure framework. The log file is also organized in YAML format. For each operation defined in the infrastructure 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 application's output results.

Due to space considerations, the detailed syntaxes of above components are not explained. They can be checked from the application scenario results on GitHub mentioned in the following Section 5.

5 Application Scenario and Evaluation

Our experiments are conducted on real Clouds instead of simulators and indeed there are some Cloud performance uncertainty issues that affect the QoS of scientific applications. Hence, we have not directly migrated any scientific applications onto Clouds here. In order to test the functionality and performance of CloudsStorm, we assume a common task-based application, which is software testing. We assume in this scenario that there are lots of test cases that need to be repeated many times, beyond our own local computing capability. If we leverage Cloud resources, we should be able to execute the application and get the results. These characteristics satisfy common features of task-based applications mentioned in Section 1. Hence, we simulate a performance test on multiple data centers and Clouds to simulate the application scenario. We pick several data centers from two supported Clouds of CloudsStorm, EC2 and ExoGENI [9]. There are three data centers picked from ExoGENI. They are the one located at Sydney (AUS for short), the one located at University of Amsterdam (UvA for short) and the one located at Boston (BBN for short). There is one data center picked from EC2, which is the one located at California (CAL for short). The application's task is to have a VM in each data center above and leverage 'sysbench' to test each data center's CPU/memory performance. The provisioning overhead is recorded as well. For testing the network performance, we simulate a client from the ExoGENI UvA data center to test the bandwidth with the above VMs at different locations. In this assumed application scenario, these tests are required to repeat regularly. As for comparison, all the VMs have 1 virtual CPU and 1GB memory, which are 't2.micro' type for EC2 and 'XOSmall' for ExoGENI. Moreover, all relevant actual topology descriptions, codes and results can be found on GitHub[10].

Finally, the goal of this application scenario is to test: i) Whether the application can leverage our programmable framework, CloudsStorm, to program on its infrastructure; ii) whether these task-based applications can run on Clouds effectively and get the results; iii) whether Cloud resources can be provisioned on demand and released in time.

[9]http://www.exogeni.net/

[10]https://github.com/zh9314/CloudsStorm/tree/master/examples/CloudPerformanceTests

Pseudocode 1 Example Solution with Infrastructure Code

```
for a certain time period or a certain count do
    Provision 'SubTopology' from AUS||BBN||CAL||UvA
    Execute CPU test simultaneously on the 'VM'
                                  from AUS || BBN || CAL || UvA
    Execute memory test simultaneously on the 'VM'
                                  from AUS || BBN || CAL || UvA
    Perform bandwidth test to the 'VM' of UvA
                                  (with IP "192.168.10.11")
    Perform bandwidth test to the 'VM' of AUS
                                  (with IP "192.168.10.12")
    Perform bandwidth test to the 'VM' of BBN
                                  (with IP "192.168.10.13")
    Perform bandwidth test to the 'VM' of CAL
                                  (with IP "192.168.10.14")
    Get the results
    Wait for executing another round of tests
end for
```

5.1 Example Solution and Results

In order to complete the task in this application scenario, we first design our networked infrastructure topology according to Section 3.2. We define four sub-topologies with one VM in each sub-topology. All these five VMs are in the same subnet, which is "192.168.10.0/24". We then leverage the infrastructure code to define the entire process for performing tests on the selected Clouds. Pseudocode 1 is described as follows. We use the "LOOP" code to repeat the test tasks. Firstly, we provision the corresponding Cloud resources; then, we define the operations of CPU test and memory test simultaneously on all the object VMs. This is achieved by the definition of "||" in Section 3.3. In addition, we simulate another VM from UvA as a client to perform the bandwidth test with other VMs. This VM is also in the subnet and its address is "192.168.10.10". Though the public IP addresses of the other VMs cannot be determined, we can always use the defined private IP address to complete the bandwidth test. This is another advantage to provisioning a networked infrastructure on Clouds with our CloudsStorm. Finally, we get the results and sleep for a while to perform these tests again.

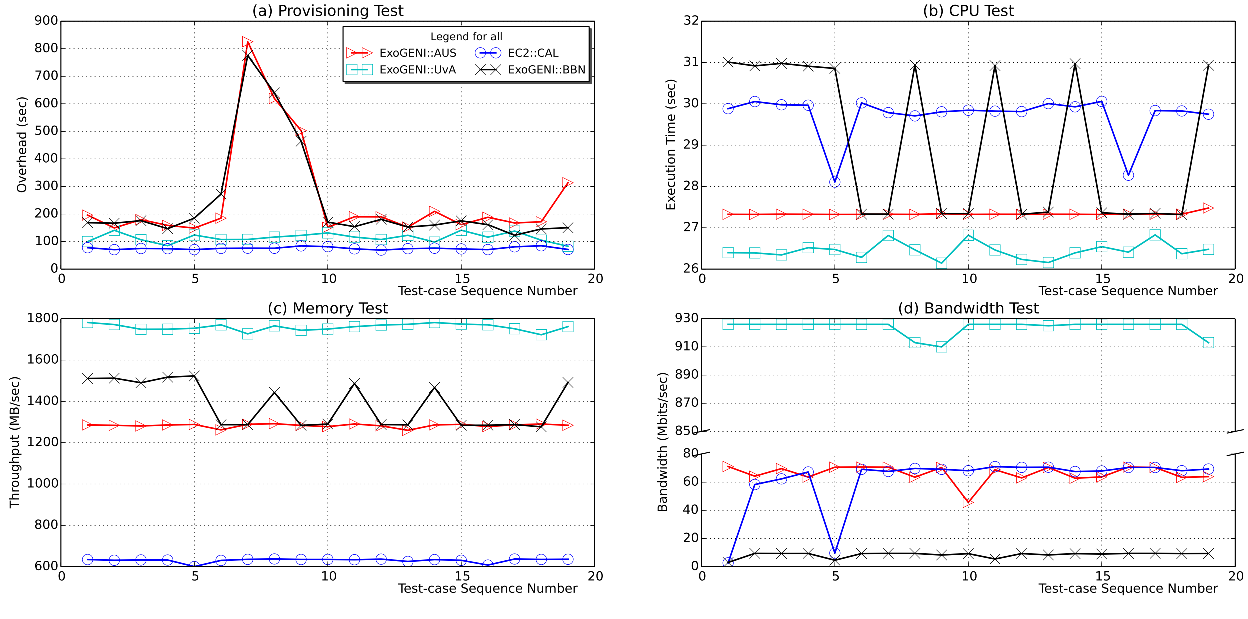

Figure 4: Results of Performance Test Tasks in this Assumed Task-based Application Scenario based on CloudsStorm

In this case, all the test results are recorded in the infrastructure code log, described in Section 4.2. Hence, we do not need explicitly to define an operation to retrieve results from remote resources. We instead extract test results from the log, analyze it and present the results in Figure 4. It lists four types of results, including the result generated by task of provisioning test, CPU test, memory test and bandwidth test. The x axis is the test-case sequence number, organized as by iteration over the total test period. For this case, we extract 19 iterations of test-cases to obtain our results; the time interval between each test-case is about 30 minutes. Hence, the entire running duration of this performance test application is about 12 hours.

Through analyzing these results, we are able to acquire some valuable insights. For instance, the provisioning overhead of commercial Cloud EC2 is relatively low compared with the community Cloud ExoGENI. On the contrary, the memory and CPU quality of resources from EC2 are not so good. There might be another workload influencing the data center of AUS and BBN from ExoGENI simultaneously as shown in Figure 4 (a) causing the provisioning overhead at both data centers to dramatically increase during the testing period. For the bandwidth test, the bandwidth to UvA is the highest, because the simulated client is in the same data center. All the above demonstrate the feasibility and functionality of our programmable infrastructure framework.

5.2 Evaluation

In this last subsection, we evaluate our framework's efficiency. Here, the efficiency refers to how much cost (money) we can save through leveraging CloudsStorm to run these task-based applications. The efficiency comes from two sources. 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 it can perform some operations in parallel to reduce the entire execution time for the application. We therefore first compare our cost with the cost of traditionally setting up Cloud resources manually. We then compare our cost with the cost of a specific script for provisioning the resource and executing the application. Due to the fact that Clouds charge based on usage time (e.g. 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 operation overhead.

The comparison result is shown in Figure 5. In one case, we set up these VMs manually, and run the application to perform tests. This is the traditional way to use Clouds. Hence, Cloud resources need to be kept during the entire application lifespan. Taking the above scenario as an example, the actual execution time for each test-case is about 10 minutes and the remaining 20 minutes is waiting. Therefore, the traditional manual method completely wastes resources during this waiting time period. The left part of Figure 5 shows that with CloudsStorm, the Cloud resources usage time is about 40.4% of the manually-controlled usage. In another case, we can develop a specific script to automate provisioning and deleting resources from some specific Cloud to reduce wasteful waiting times. However, the script cannot perform some operations in parallel. For example, in the above scenario, the CPU and memory tests can be performed in parallel. Based on the each operation overhead in the log, we obtained the conclusion that CloudsStorm still reduces the total Cloud resource usage by 45.1% compared with using a specific script as shown in right part of Figure 5.

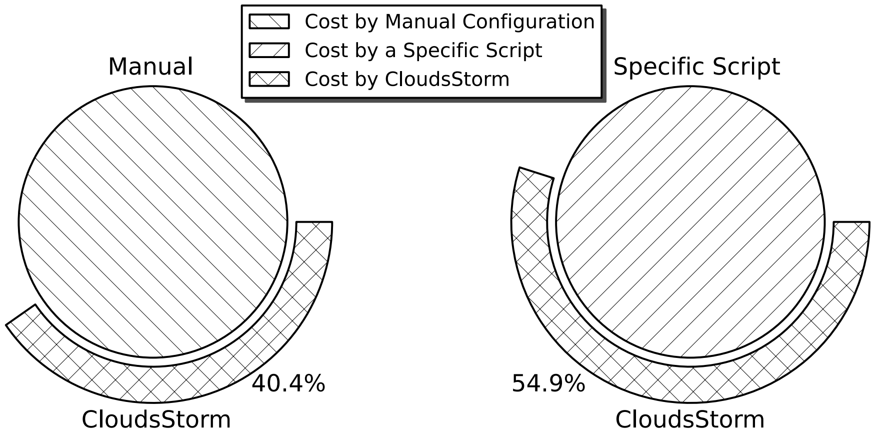

Figure 5: Cloud Resource Usage Cost Comparison

6 Conclusion and Future Work

This paper presents, CloudsStorm, a framework empowering dynamic task-based applications with agile virtual infrastructure programmability. First, we introduce its execution model and describe the relevant syntaxes of its front end. To be specific, the topology description language is leveraged to describe the application-defined networked infrastructure among Clouds and the infrastructure code is an interpreted language to execute application-defined operations. Meanwhile, all these syntaxes are based on YAML, which is human readable and easy to manage. We have also implemented a back-end TSV-Engine to support those application-defined operations. It is extensible, efficient and able to execute parallel operations. Finally, experimental results demonstrate the feasibility and functionality of CloudsStorm, in which applications can define their own networked virtual infrastructure, run on the infrastructure and get their results. Moreover, CloudsStorm achieves provisioning of resources on demand and their in-time release; CloudsStorm is therefore able to reduce the resource usage cost compared with other traditional methods. In future work, we are going to leverage CloudsStorm to execute data-centric applications on Clouds, which is a specific type of task-based applications, to realize locality-aware, data-aware, etc. It will further demonstrate the benefit of our approach.

Acknowledgment

This research has received funding from the EU Horizon 2020 research and innovation program under grant agreements 643963 (SWITCH project), 654182 (ENVRIPLUS project) and 676247 (VRE4EIC project). This work was partially supported in part by National Key R&D Program under contract (2016YFE0100300). We also would like to thank Paul Martin's contribution to this work.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[2] Zhiming Zhao, Adam Belloum, and Marian Bubak. Special section on workflow systems and applications in e-science. *Future Generation Computer Systems*, 25(5):525–527, 2009.

[3] Hao Zhu, Karel van der Veldt, Zhiming Zhao, Paola Grosso, Dimitar Pavlov, Joris Soeurt, Xiangke Liao, and Cees de Laat. A semantic enhanced power budget calculator for distributed computing using ieee 802.3 az. *Cluster Computing*, 18(1):61–77, 2015.

[4] Iman Sadooghi, Jesús Hernández Martin, Tonglin Li, Kevin Brandstatter, Ketan Maheshwari, Tiago Pais Pitta de Lacerda Ruivo, Gabriele Garzoglio, Steven Timm, Yong Zhao, and Ioan Raicu. Understanding the performance and potential of cloud computing for scientific applications. *IEEE Transactions on Cloud Computing*, 5(2):358–371, 2017.

[5] In Kee Kim, Jacob Steele, Yanjun Qi, and Marty Humphrey. Comprehensive elastic resource management to ensure predictable performance for scientific applications on public iaas clouds. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 355–362. IEEE Computer Society, 2014.

[6] Hao Qian and Daniel Andresen. Automate scientific workflow execution between local cluster and cloud. *the International Journal of Networked and Distributed Computing (IJNDC)*, 4(1):45–54, 2016.

[7] Kief Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[8] Yang Hu, Junchao Wang, Huan Zhou, Paul Martin, Arie Taal, Cees de Laat, and Zhiming Zhao. Deadline-aware deployment for time critical applications in clouds. In *European Conference on Parallel Processing*, pages 345–357. Springer, 2017.

[9] Xue Ouyang, Peter Garraghan, Bernhard Primas, David McKee, Paul Townend, and Jie Xu. Adaptive speculation for efficient internetware application execution in clouds. *ACM Transactions on Internet Technology (TOIT)*, 18(2):15, 2018.

[10] Zhiming Zhao, Paola Grosso, Jeroen Van der Ham, Ralph Koning, and Cees De Laat. An agent based network resource planner for workflow applications. *Multiagent and Grid Systems*, 7(6):187–202, 2011.

[11] Junchao Wang, Arie Taal, Paul Martin, Yang Hu, Huan Zhou, Jianmin Pang, Cees de Laat, and Zhiming Zhao. Planning virtual infrastructures for time critical applications with multiple deadline constraints. *Future Generation Computer Systems*, 75:365–375, 2017.

[12] Zhiming Zhao, Dick Van Albada, and Peter Sloot. Agent-based flow control for hla components. *Simulation*, 81(7):487–501, 2005.

[13] Joon-Myung Kang, Hadi Bannazadeh, and Alberto Leon-Garcia. Savi testbed: Control and management of converged virtual ict resources. In *Integrated Network Management, IFIP/IEEE International Symposium on*, pages 664–667, 2013.

[14] Javier Diaz-Montes, Moustafa AbdelBaky, Mengsong Zou, and Manish Parashar. Cometcloud: Enabling software-defined federations for end-to-end application workflows. *IEEE Internet Computing*, 19(1):69–73, 2015.

[15] Huan Zhou, Junchao Wang, Yang Hu, Jinshu Su, Paul Martin, Cees De Laat, and Zhiming Zhao. Fast resource co-provisioning for time critical applications based on networked infrastructures. In *Cloud Computing (CLOUD), IEEE International Conference on*, pages 802–805, 2016.

[16] Germain Faure and Alexandre Miquel. *A categorical semantics for the parallel lambda-calculus*. PhD thesis, INRIA, 2009.

[17] Zhiming Zhao, Arie Taal, Andrew Jones, Ian Taylor, Vlado Stankovski, Ignacio Garcia Vega, Francisco Jesus Hidalgo, George Suciu, Alexandre Ulisses, Pedro Ferreira, et al. A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch). In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 1181–1184. IEEE, 2015.