Seamless Infrastructure Programming and Control for Quality-critical Cloud Applications · Huan Zhou 周寰

Seamless Infrastructure *Programming* and *Control* for Quality-critical Cloud Applications

Huan Zhou

University of Amsterdam

Seamless Infrastructure Programming and Control for Quality-critical Cloud Applications

Huan Zhou

The author was sponsored by the China Scholarship Council.



The author also would like to thank ExoGENI, EGI, and DAS-5 for providing testbeds to conduct the experiments.

This research was also supported by the European projects of Horizon 2020 Programme under grant agreement No. 643963 (SWITCH), No. 825134 (ARTICONF), No. 654182 (ENVRIplus), No. 824068 (ENVRIFAIR), and No. 676247 (VRE4EIC).




Cover designed by Huan Zhou and Visual Generation
Printed by IPSKAMP, Enschede

ISBN: 978-94-028-1727-0

Seamless Infrastructure Programming and Control for Quality-critical Cloud Applications

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex
ten overstaan van een door het College voor Promoties ingestelde
commissie, in het openbaar te verdedigen in
de Agnietenkapel
op woensdag 23 oktober 2019, te 14:00 uur

door

Huan Zhou

geboren te Anhui

Promotiecommissie

Promotor:			
	Prof. dr. ir.	C.T.A.M. de Laat	Universiteit van Amsterdam
Co-promotor:			
	Dr.	Z. Zhao	Universiteit van Amsterdam
Overige leden:			
	Prof. dr. ir.	H.E. Bal	Vrij Universiteit Amsterdam
	Prof. dr.	R. Prodan	University of Klagenfurt
	Prof. dr.	J. Su	National University of Defense Technology
	Prof. dr.	R.V. van Nieuwpoort	Universiteit van Amsterdam
	Prof. dr.	P.W. Adriaans	Universiteit van Amsterdam
	Dr.	A.S.Z. Belloum	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

A hundred years ago, companies stopped generating their own power with steam engines and dynamos and plugged into the newly built electric grid. The cheap power pumped out by electric utilities didn't just change how businesses operate. It set off a chain reaction of economic and social transformations that brought the modern world into existence. Today, a similar revolution is under way. Hooked up to the Internet's global computing grid, massive information-processing plants have begun pumping data and software code into our homes and businesses. This time, it's computing that's turning into a utility.

Nicholas Carr
The Big Switch: Rewiring the World, from Edison to Google
2008

But *how* can we *effectively* utilise Cloud resources as a *utility*?

Contents

1 Introduction

Cloud computing allows developers to operate applications without maintaining physical infrastructures. The features of elasticity and pay-as-you-go provided by Cloud computing have sparked the trend to orchestrate applications on Clouds for reducing cost. During the past years, we have witnessed an explosive growth of Cloud computing in both industrial and academic fields. Figure 1.1 shows the prosperity of Cloud computing. It illustrates that the geographical distribution of data centres from well-known Cloud providers, including Amazon Elastic Compute Cloud (EC2)[1], Microsoft Azure[2], Google Compute Engine (GCE)[3], Alibaba Cloud[4], and ExoGENI Cloud[5]. The geolocations of these data centres cover most continents around the world.

Enabled by the Internet and virtualisation techniques, Cloud computing is able to deliver virtualised computing resources to remote customers over a network as services. Based on the resource stack being virtualised, those services can be typically classified as three levels: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). SaaS allows customers to directly use specific application software via the Internet without knowing the location and platform where it is actually deployed. PaaS provides customers with an online software platform to develop and execute application software. IaaS further provides customers with resource stack at the operating system level, where they can fully customise the high-level platform and software environment. IaaS leverages the hypervisor [11] to isolate specific virtual machines (VM) for multiple tenants through hardware virtualisation. From SaaS to IaaS, users can obtain more programmability and controllability of the infrastructure, which is crucial for software development and industrial innovation. Gartner[6] predicts that worldwide public Cloud service revenue will grow exponentially through 2022. The fastest-growing market segment will be IaaS, which is predicted to grow 27.5% in 2019 to reach $38.9 billion, up from $30.5 billion in 2018. It demonstrates ever more applications and services require to be migrated onto Clouds, especially with the IaaS Cloud service model.

[1] https://aws.amazon.com/ec2/

[2] https://azure.microsoft.com/

[3] https://cloud.google.com/compute/

[4] https://www.alibabacloud.com/

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

[6] https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g

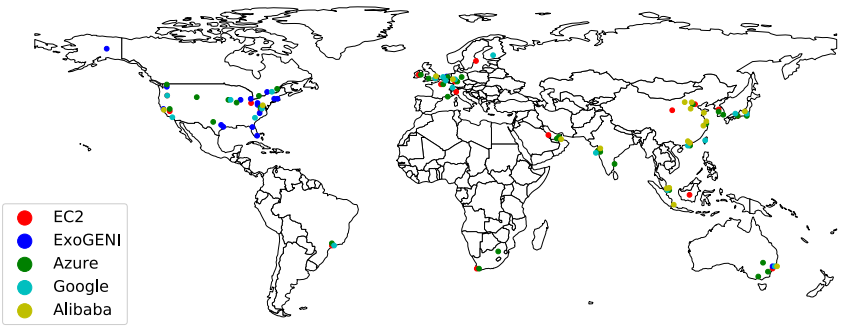


Figure 1.1: Geographical distribution of data centres from different well-known Clouds

The software Development and Operations (DevOps) lifecycle for Cloud applications is significantly different from the one for traditional systems. In the traditional model, an IT department is often divided into two teams: one development team responsible for the coding, unit testing, and debugging in the development phase; and one operation team responsible for deploying and orchestrating the application on the infrastructure in the runtime phase [54]. The operational problems met during runtime will be fed back to the development team to trigger further software development and delivery. In Cloud environments, however, a cycle of Service Level Agreement (SLA) enforcement and infrastructure provisioning is needed to connect two traditional cycles of software development and operations. Since the infrastructure is not physically owned by application developers, a provisioning step is required to apply the resources from the data centres, and also the SLA is essential to assure the virtual infrastructure can meet the quality constraints of the application. In this context, a number of challenges can be highlighted:

- **Cloud customisation and interoperability.** The diversity of the Cloud services types and interfaces makes it difficult for developers to customise the infrastructure resources from different Clouds and data centres with a unified description.
- **Infrastructure control complexity.** Most of the current Cloud infrastructures can only be controlled manually through the web console offered by the Cloud providers. An application has to provision infrastructure in advance, with limited capability for programming virtual infrastructures based on its runtime needs.
- **Vendor lock-in.** Besides the web console, each Cloud also provides its own Application Programming Interface (API) for the developer to control the infrastructure remotely, but different interfaces impede the developer to leverage the resources of various Clouds and cause the vendor lock-in problem.
- **Performance uncertainty.** The Cloud resources are shared among all the Cloud customers. Therefore, the performance uncertainty issue due to the resource contention hinders the infrastructure from satisfying the application quality requirements. Especially the unpredictable downtime of the data centres further raises the difficulty of operating applications with high-quality requirements.

The above challenges have been tackled from different perspectives. From the Cloud providers' side, the research mainly focuses on the efficiency in managing the physical infrastructure inside a data centre. For instance, Mohammad et al. [3] and Albert et al. [48] discuss how to build the data centre network to manage the traffic and make the communication salable. Software-defined Network (SDN) is also proposed as one of the data centre network solutions. Tao et al. [113] propose the optimised algorithm for dynamically assigning SDN controllers in the data centre network. Hao et al. [134] design a network management platform to reduce the data centre energy consumption based on SDN. There is also plenty of research focusing on the data centre level of job scheduling [30, 110], VM consolidation [37, 77], and energy efficiency [28, 122]. The research all above is practical to be applied to the current data centre implementation, but none of them provides the Cloud customer with the solution to expediently leverage the Cloud resources. Although some of the research discusses the Cloud application development, e.g., Dan et al. [93] exploit the high-performance data centre network to design the distributed systems, it is still based on the assumption that the developer has the total control of the data centre physical infrastructure, which is not permitted to general Cloud customers.

On the contrary, recent research taking care of the customer side mainly focuses on high-level application mapping and modelling. For example, Amelie et al. [129] and Sreekrishnan et al. [111] map the application components to proper Clouds according to the geolocation of the data centre. Kai et al. [56] and Alexey et al. [58] investigate the scaling policies in Cloud environments to satisfy the application Quality of Service (QoS). Nevertheless, programmatically leveraging these algorithms is still challenging, especially to orchestrate the applications and manage the infrastructures from different Clouds. To be specific, during the DevOps lifecycle, the gap between the application operation and the Cloud virtual infrastructure management still exists.

1.1 Research Questions

To tackle the software challenges in utilising diverse Cloud resources for applications with high-quality constraints, we thus identify the main Research Question (RQ) as:

RQ. How to seamlessly program and control the Cloud virtual infrastructure in the application DevOps lifecycle to satisfy the quality-critical constraints of the application?

We analyse this main problem in the context of Cloud application DevOps lifecycle, and address relevant challenges from different aspects: developing, provisioning, operating, and SLA assurance. We thus decompose the research question into four detailed research questions:

- **RQ1.** How can we customise and program the infrastructure according to different application quality requirements?
- **RQ2.** How can we effectively provision a networked infrastructure and enable topology partitioning across data centres or Cloud providers based on application QoS constraints?

- **RQ3.** How can an application efficiently control the virtual infrastructure at runtime, preferably without vendor lock-in?
- **RQ4.** How can we effectively handle the SLA with the provider to make the service quality assurance trustworthy?

The highly distributed Cloud data centres, as shown in Figure 1.1, provide rich choices for developers to provision virtual infrastructures for their applications, e.g., for selecting resources close to the data sources, suitable price, and performance. In many cases, resources from different providers are needed. The current description standard of Topology and Orchestration Specification for Cloud Applications (TOSCA)[7] [18] focuses more on the Cloud application components description down to the VM level, without the explicit specification of the Clouds and data centres. Moreover, the operations, such as scaling and failure recovery, performed on the infrastructure are also not defined. Especially, these operations cannot be programmatically leveraged to satisfy the application constraints. We will thus tackle those challenges in the context of **RQ1**.

In large scale distributed applications, e.g., big data infrastructures in environmental and earth sciences, data often needs to be collected and processed in different geolocation to meet (nearly) real-time requirements. The virtual infrastructure often needs to be across different data centres or providers. It is, therefore, inevitable to partition the infrastructure and provision them in different data centres. Moreover, such partitioning should be transparent for the orchestration of applications. We thus formulate those challenges as the **RQ2**.

As Cloud applications are often highly dynamic, the virtual infrastructures often need to be adapted, even though they are properly customised and provisioned at the development phase. Traditionally, the infrastructure management and the application operation are supported by separated tools, often from the Cloud providers. It is often difficult to include infrastructure control logic in the application, in particular, when across different providers. It is, therefore, essential to enable an application efficiently control the virtual infrastructure at runtime, preferably without vendor lock-in, as **RQ3**.

Since the Cloud infrastructure is remotely shared and not physically owned by application developers, the quality requirements of a Cloud application can hardly be always satisfied. The SLA is the last guarantee to economically compensate the Cloud customer, i.e., the application developer in our case. However, always the question arises who can detect service violation and determine the SLA state; i.e., who are the witness and the judge. Furthermore, it is also challenging to ensure that the customer is really able to get compensation from the provider when the violation happens. Thus, the last subquestion we will tackle is to effectively handle the SLA with the provider to make the service quality assurance trustworthy, as stated in **RQ4**.

1.2 Main Contributions

This thesis contributes models, algorithms, and prototypes to the Cloud application development and operations. For details, we classify the contributions and present them according to the implemented two prototypes.

[7] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#overview

CloudsStorm[8]: a framework for seamless Cloud virtual infrastructure programming and control

- We propose and design three types of infrastructure code to deal with functional requirements, which describe the infrastructure topology and the operations performed on the infrastructure. These types of infrastructure code are corresponding to three levels of infrastructure programmability, respectively.
  - "*Infrastructure Description Code*" provides the design-level programmability. It enables the developers to describe the infrastructure computing capability and topology for hosting their applications, notably including the specification of Clouds and data centres. Its syntax is defined in the format of YAML Ain't Markup Language (YAML), which is clear and human-readable.
  - "*Infrastructure Execution Code*" provides the infrastructure-level programmability. It empowers the developers with the ability to describe and program the infrastructure operations. Then the infrastructure can be programmatically provisioned from scratch and controlled afterwards. Specifically, it is able to describe high-level and parallel operations easily. The syntax of it is also based on YAML.
  - "*Infrastructure Embedded Code*" provides the application-level programmability. It notably allows the developers to embed the infrastructure operations into their application logic. The application can, therefore, directly control the infrastructure. Its syntax is in the form of some general-purpose language, such as the interfaces implemented in Java[9].
- We propose a programming model by abstracting infrastructure operations as functions. Then we model the basic Cloud Virtual Infrastructure Functions (VIFs), which are commonly provided by different Clouds. Based on these basic functions, we then construct high-level infrastructure operation functions, e.g., horizontal scaling, vertical scaling, and failure recovery. We demonstrate the feasibility to construct infrastructure operations from these basic functions and simplify the expression to identify parallel operations.
- We propose and implement an infrastructure runtime control model with two types of control modes.
  - Passive Mode, in which, the infrastructure is passively controlled according to the monitoring information. We define the YAML based "*Runtime Control Policy*" for developers to address non-functional requirements, which describes controlling operations with specific predefined conditions.
  - Active Mode, in which the infrastructure can be actively controlled by the application. It is benefit from the "*Infrastructure Embedded Code*". The advantage of it is to adjust the infrastructure before actual influences happening because of the varying workloads.

[8]https://github.com/zh9314/CloudsStorm
[9]https://github.com/zh9314/CloudsStormREST

- We design and implement the CloudsStorm framework for developing and orchestrating applications on Clouds. To be specific, we implement the "*Infrastructure Execution Engine*"[8] to realise extensible and efficient Cloud virtual infrastructure control based on our programming model. Besides, the "*Control Agent*"[10] is implemented to monitor and perform the control operation at runtime. It also provides users with a web-based user interface for interaction.
- We achieve empirical results of orchestrating quality-critical applications on the Cloud virtual infrastructure, through performing case studies using CloudsStorm on real Clouds.

A blockchain based solution[11] for trustworthy Cloud SLA enforcement

- We implement a blockchain based solution to automate the SLA lifecycle between the Cloud provider and the customer. The provider and the customer are, therefore, ensured to get the corresponding service fee and the compensation fee if a violation happens.
- We design a witness model for incentivising blockchain participants to perform the violation detection and enforce the SLA between the Cloud provider and the customer. The payoff function for rewarding the witness is carefully designed to drive the witness to tell the truth about the violation. The trustworthiness is proved through using the Nash Equilibrium principle of game theory.
- We propose an unbiased random sortition algorithm based on the randomness of the blockchain itself. This algorithm can be leveraged to select several elements from a set randomly. No single entity can determine the results beforehand. It is implemented in the smart contract for the witness sortition in our prototype.
- We propose the witness auditing mechanism to analyse the behaviour of the witnesses. Since all the witnesses' behaviours are recorded on the blockchain, we adopt this reputation management mechanism to filter out the irrational or malicious witnesses from the system. Specifically, we propose three types of malicious behaviours and quantitative indicators to audit.

1.3 Thesis Overview

Figure 1.2 illustrates the overview of this thesis, which includes the relationship among the chapters, contributions, and research questions. The thesis contains eight chapters in total. Chapter 1 is the introduction to the problem scope and research questions of this thesis. Chapter 2 briefly discusses some background knowledge and challenges from four aspects: Cloud virtual infrastructure, quality-critical Cloud applications, the state-of-the-art DevOps of Clouds, and blockchain as well as smart contracts.

Following three chapters address the research questions, **RQ1**, **RQ2**, and **RQ3**, respectively. Each chapter focuses on a particular phase in the DevOps lifecycle for developing and operating applications based on Cloud virtual infrastructures.

[10] https://github.com/zh9314/CloudsStormCA
[11] https://github.com/zh9314/SmartContract4SLA

- For the development phase, Chapter 3 focuses on Cloud virtual infrastructure programmability design with the infrastructure operation programming model and the syntax of the infrastructure code.
- For the provisioning phase, Chapter 4 investigates the fast provisioning mechanism to provide a networked infrastructure among the federated Cloud virtual infrastructure.
- For the runtime phase, Chapter 5 depicts the Cloud virtual infrastructure controllability implementation and the infrastructure control model with two modes.

All the above three chapters together also present the CloudsStorm framework in detail, including the framework design and overview in Chapter 3, the overlay network configuration in Chapter 4, and the framework implementation in Chapter 5.

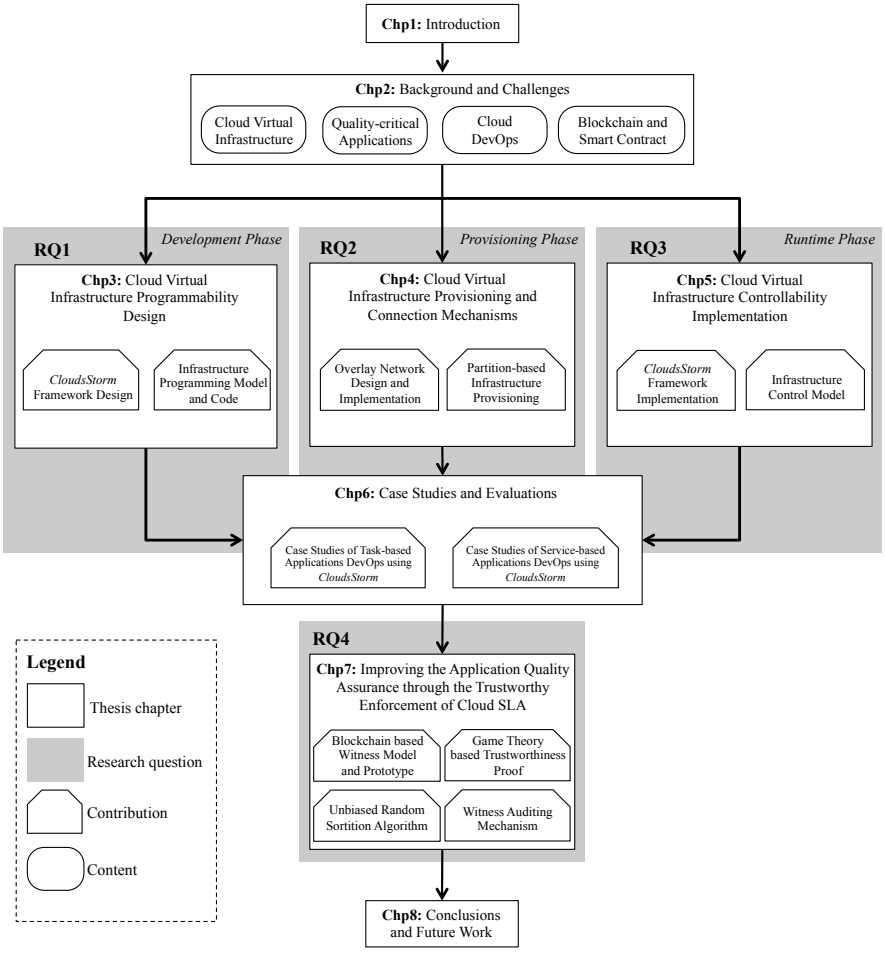


Figure 1.2: The overview of the thesis, including the chapters, the research questions, and the contributions

For demonstration, Chapter 6 conducts several experiments using CloudsStorm on real Clouds, which are generally classified as two types of case studies: the task-based application, and the service-based application. The example solutions with CloudsStorm to consider the quality constraints and the experimental results are presented. All the first three research questions, i.e., **RQ1**, **RQ2**, and **RQ3**, are somehow demonstrated in this chapter.

Then, Chapter 7 tackles the research question, **RQ4**, to further improve the application quality assurance. We propose the witness model based on blockchain and smart contract to better handle the SLA between the Cloud provider and the customer. The quality constraints of the applications are, therefore, more assured because of the enhanced SLA enforcement.

Finally, Chapter 8 concludes the thesis and discusses the prospect of future work.

1.4 Publications

All the chapters tackling research questions of this thesis have been published in peer-reviewed journals and conferences. The complete list of 21 publications is shown at the end of this thesis as achievements. In this section, we highlight all the publications which are closely related to this thesis. Then we explain how each chapter is associated with the highlighted publications.

1. **Zhou, H.**, Hu, Y., Ouyang, X., Su, J., Koulouzis, S., de Laat, C., Zhao, Z., "CloudsStorm: A Framework for Seamlessly Programming and Controlling Virtual Infrastructure Functions during the DevOps Lifecycle of Cloud Applications", Journal of *Software: Practice and Experience*. Wiley, 2019.

2. **Zhou, H.**, Ouyang, X., Su, J., de Laat, C., Zhao, Z., "Enforcing Trustworthy Cloud SLA with Witnesses: A Game Theory based Model using Smart Contracts", Journal of *Concurrency and Computation: Practice and Experience*. e5511. Wiley, 2019.

3. Koulouzis, S., Martin, P., **Zhou, H.**, Hu, Y., Wang, J., Carval, T., Grenier, B., Heikkinen, J., de Laat, C., Zhao, Z., "Time-critical data management in clouds: Challenges and a Dynamic Real-Time Infrastructure Planner (DRIP) solution", Journal of *Concurrency and Computation: Practice and Experience*, e5269. Wiley, 2019. (as co-first author)

4. **Zhou, H.**, Ouyang, X., Ren, Z., Su, J., de Laat, C., Zhao, Z., "A Blockchain based Witness Model for Trustworthy Cloud Service Level Agreement Enforcement" In *IEEE International Conference on Computer Communications* (INFOCOM), pp. 1567-1575. IEEE, 2019.

5. **Zhou, H.**, de Laat, C., Zhao, Z., "Trustworthy Cloud Service Level Agreement Enforcement with Blockchain Based Smart Contract" In *IEEE International Conference on Cloud Computing Technology and Science* (CloudCom), *workshop on resource brokering with blockchain* (RBChain), pp. 255-260. IEEE, 2018.

6. **Zhou, H.**, Koulouzis, S., Hu, Y., Wang, J., de Laat, C., Ulisses, A., Zhao, Z., "Migrating Live Streaming Applications onto Clouds: Challenges and a CloudsStorm Solution", In *11th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (UCC Companion), *workshop on Cloud-Native Applications Design and Experience* (CNAX), pp. 321-326. IEEE, 2018.

7. **Zhou, H.**, Taal, A., Koulouzis, S., Wang, J., Hu, Y., Suciu, G., Poenaru, V., de Laat, C., Zhao, Z., "Dynamic Real-Time Infrastructure Planning and Deployment for Disaster Early Warning Systems", In *International Conference on Computational Science, workshop on Data, Modeling, and Computation in IoT and Smart Systems*, pp. 644-654. Springer, Cham, 2018.

8. **Zhou, H.**, Hu, Y., Su, J., Chi, M., de Laat, C., Zhao, Z., "Empowering Dynamic Task-Based Applications with Agile Virtual Infrastructure Programmability", In *IEEE 11th International Conference on Cloud Computing* (CLOUD), pp. 484-491. IEEE, 2018.

9. **Zhou, H.**, Hu, Y., Su, J., de Laat, C., Zhao, Z., "CloudsStorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures", In *International Conference on Cloud Computing*, pp. 265-280. Springer, Cham, 2018.

10. **Zhou, H.**, Martin, P., Su, J., de Laat, C., Zhao, Z., "A Flexible Inter-locale Virtual Cloud For Nearly Real-time Big Data Application", In *IEEE Real Time System Symposium* (RTSS), *International workshop on Interoperable infrastructures for interdisciplinary big data sciences* (IT4RIs), 2016.

11. **Zhou, H.**, Wang, J., Hu, Y., Su, J., Martin, P., de Laat, C., Zhao, Z., "Fast resource co-provisioning for time critical applications based on networked infrastructures", In *IEEE 9th International Conference on Cloud Computing* (CLOUD), pp. 802-805. IEEE, 2016.

12. **Zhou, H.**, Hu, Y., Wang, J., Martin, P., de Laat, C., Zhao, Z., "Fast and dynamic resource provisioning for quality critical cloud applications", In *IEEE 19th International Symposium on Real-Time Distributed Computing* (ISORC), pp. 92-99. IEEE, 2016.

The first author has made significant contributions to the above publications with designing, implementing, and conducting the experiments. All authors have made contributions of proofreading and commenting.

Chapter 3 (Cloud virtual infrastructure programmability) and Chapter 5 (Cloud virtual infrastructure controllability) are based on Publication 1 and Publication 9. Chapter 4 (Cloud virtual infrastructure provisioning and overlay network mechanisms) is based on Publication 11 and Publication 12. The majority part of Chapter 6 (case studies of task-based and service-based applications using CloudsStorm) is based on Publication 8 and Publication 10. Chapter 6 is also partially related to Publication 1, Publication 3, Publication 6, and Publication 7. Chapter 7 (blockchain enhanced trustworthy SLA enforcement) is based on Publication 2, Publication 4, and Publication 5.

2 Background and Challenges

In this chapter, we introduce the concepts and background knowledge within the context of this thesis. We start with an introduction to the Cloud virtual infrastructure (in Section 2.1), and then discuss quality-critical Cloud applications (in Section 2.2), specifically on the terminology of quality-critical constraints and Cloud applications. Afterwards, we review the state of the art of the software Development and Operations (DevOps) technologies (in Section 2.3), introduce the background knowledge of blockchain and smart contract technology (in Section 2.4). From those reviews, we identify the research challenges.

2.1 Cloud Virtual Infrastructure

The infrastructure mentioned in this thesis mainly refers to the virtualised infrastructure provided by the Cloud IaaS. We shall first introduce the technical basis of Cloud virtualisation, then discuss the basic Cloud service models, and compare the programmability and controllability they offer to Cloud users.

2.1.1 Cloud Virtualisation Stack

From the Cloud providers' point of view, Cloud computing delivers various levels of resources from the computing stack (between hardware and high-level software) to the end user via a service-oriented architecture. By leveraging virtualisation techniques, a provider allows end users to share the usage of physical resources within data centres in an isolated way. We shall review three different levels of virtualisation techniques.

Hardware Virtualisation, the lowest level of virtualisation, partitions the hardware resources, such as CPU and memory, of a single server, exploits partitioned slices as different virtual machines (VMs). It emulates multiple guest Operating Systems (OS) with separated kernels on the physical server. The hypervisor [11] is used to manage all the VMs. There are two types of hypervisor implementations [1]: hardware-based, e.g., Xen, and software-based, e.g., Kernel-based Virtual Machine (KVM) [102]. Both types of the hypervisor are widely used.

The hardware virtualisation has high security and performance isolation, since the real hardware resources are partitioned and mapped to the VMs; however, it often has high overhead due to its large size of OS image, and is less portable.

Operating System Virtualisation facilitates the host OS kernel and isolates the resources with some user-space instances; a typical implementation is containerisation [101]. Running in a shared OS as processes, containers are lightweight and portable compared to VMs, but with less security and performance isolation. Docker [40] is an efficient tool to containerise software packaged and becomes the trend of orchestrating services on Clouds.

Application Virtualisation is implemented in the OS user-space and provides an abstraction layer to separate the application from the actual underlying OS. It enables the portability of an application across diverse underlying computing environments, e.g., Java Virtual Machine (JVM). However, the programming language used to develop the application is also limited. Meanwhile, the abstraction layer hinders the application to leverage specific hardware drive to utilise their features. This technology is very effective to deliver high-level software services.

2.1.2 Cloud Service Models

From the customers' point of view, the remote Cloud resources can be ordered and consumed as *services* via specific contracts or agreements. Customers pay the provider for what they have consumed. Due to the service delivered from different levels of the computing stack (i.e., application, platform, and infrastructure), the Cloud service model can be further classified as three models [16]: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS), as we briefly mentioned in Chapter 1.

The *Software-as-a-Service (SaaS)* model only delivers software hosted on the Cloud to customers as a service. A user can utilise functionalities of the software but with limited ability for re-configuring or customising the software. This model usually adopts the subscription business model, which requires the customer to pay monthly or yearly. The typical examples of SaaS include Dropbox[1], Google Doc[2], and Microsoft OneDrive[3]. These software tools can be developed through the application virtualisation or orchestrated using the OS virtualisation technique.

The *Platform-as-a-Service (PaaS)* model delivers a software platform to customers as a service, where the customers can build or execute their own applications on top of the platform. The consumer can use the platform, often via the Application Programming Interface (API) from the provider, for developing applications, but cannot fully control the OS and the hardware under the platform. For example, Google AppEngine[4] is a PaaS whose programming interface is given in Python or Java. Other examples include Apache Spark[5], Mesos[6].

The *Infrastructure-as-a-Service (IaaS)* offers the entire computing stack down to the hardware virtualisation level, where the customers have access to the resources of computing, storage, network, and OS, in the form of VMs. Comparing to PaaS,

[1] https://www.dropbox.com/
[2] https://www.google.com/docs/
[3] https://onedrive.live.com/
[4] https://cloud.google.com/appengine/
[5] https://spark.apache.org/
[6] http://mesos.apache.org/

IaaS customers gain more programmability and controllability on Cloud resources. Furthermore, customers can customise the network among VMs based on geolocation (determined by the data centre), the capacity (determined by the vCPU and memory), and the network configuration of the VMs. Current well-known IaaS providers include Amazon Elastic Compute Cloud (EC2)[7], Microsoft Azure[8], and Google Compute Engine (GCE)[9].

2.1.3 The Programmability and Controllability Comparison

The programmability and controllability provided by the above models are crucial for the Cloud application developers. To better discuss those issues, we highlight two aspects for comparison: application and infrastructure perspective.

- The *application programmability* refers to the ability for a developer to program the application software, including choosing the programming language.
- The *application controllability* refers to the ability for the application to control its components or processes running on a specific part of the infrastructure. This ability is often pre-programmed by the developer.
- The *infrastructure programmability* refers to the ability for a developer to 1) customise the infrastructure, such as the geolocation distribution, the computing capacity, and network connections, 2) define operations performed on the infrastructure, such as failure recovery and auto-scaling.
- The *infrastructure controllability* refers to the ability for the application to dynamically control the infrastructure and adapt the quality constraints of the application at runtime. This ability should be programmed and empowered by the developer.

As Figure 2.1 indicates, more programmability and controllability are provided when the provider offers the lower level service in the computing stack.

The SaaS model offers nearly no programmability and controllability at the application level. Users can only access the SaaS application via the given interface and they are not able to control the underlying resources, which are hidden by the provider.

The PaaS model provides limited application programmability via the platform. The application developers have to use the programming language required by the platform, e.g., Java for some Hadoop platforms. The application controllability is also restricted by the platform, since the resource allocation and scheduling are dominated by most of the platforms. It is impossible for a PaaS customer to customise and adjust the underlying infrastructure at runtime. For example, Kubernetes[10] platform is based on a predefined and fixed infrastructure, which can be a cluster of VMs. The application on the platform is running as a container and cannot adjust the underlying VM to adapt the computing capacity.

[7] https://aws.amazon.com/ec2/
[8] https://azure.microsoft.com/
[9] https://cloud.google.com/compute/
[10] https://kubernetes.io/

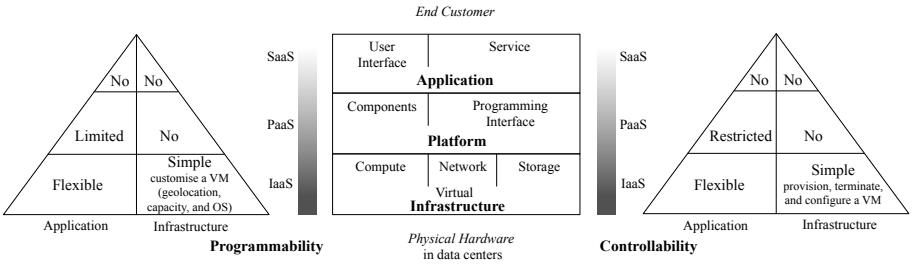


Figure 2.1: Cloud service models and the comparison of the programmability and controllability

The IaaS model allows the customer to manipulate the resources at the hardware virtualisation level, including the entire OS level, which empowers the complete application programmability and controllability. On the other hand, the IaaS model allows a customer to customise the computing capacity (CPU and memory) and the OS of a VM, from a list of possible data centres. The customer can fully control a given VM, including the operations of provisioning, terminating, and configuration. Moreover, the user can also configure different aspects of the VM, such as network configuration, execution environment configuration, and application deployment.

We thus focus on the hardware virtualisation, i.e., VM, based IaaS Cloud service model, for investigating programmability and controllability in our research questions.

2.2 Quality-critical Cloud Applications

Quality-critical Cloud applications demand a high standard of QoS (Quality of Service, e.g., tsunami emergency response time) or QoE (Quality of Experience, e.g., smooth delivery of ultra-high definition audio and video for live events) [126]. It augments the Cloud application with quality constraints. In general, the quality constraint refers to the timing requirement for the application to finish a task or request, represented as a deadline. Meanwhile, the budget and energy consumption can be other constraints on the infrastructure when operating the application in Cloud environments. The terminology of quality-critical constraints and Cloud applications is introduced as follows.

2.2.1 The Terminology of Quality-critical Constraints

In EU SWITCH project[11], we defined a simple taxonomy for classifying quality-critical constraints, as shown in Figure 2.2 [71]. The constraints about task finishing time, i.e., time-critical constraints, is one of the major concerns for quality-critical applications. It can be further classified as speed-critical, real-time and near real-time. Here, the speed-critical constraints need to minimise the completion time (i.e., the sooner, the better), while the real-time constraints are bounded by limited response time on inputs,

[11] www.switchproject.eu

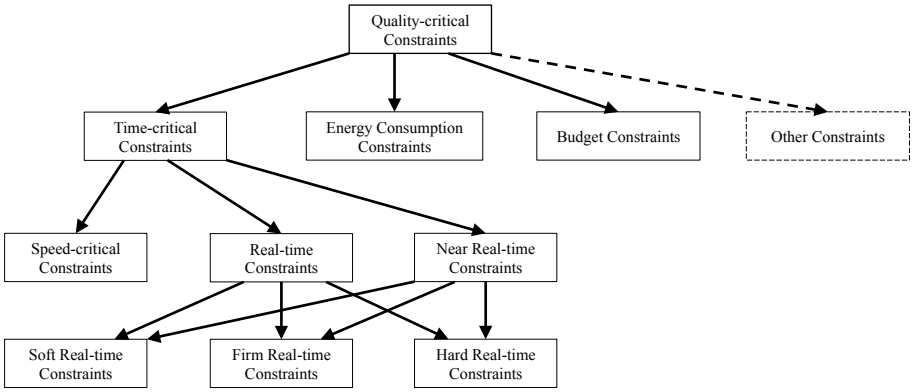


Figure 2.2: The terminology related to the quality-critical constraints

with particular consequences of failures after missing deadlines [90]. Based on those consequences, real-time constraints are further classified as hard real-time, firm real-time, and soft real-time. Hard real-time constraints cause immediate failures if any of its deadlines is missed. Firm real-time constraints can tolerate the scenario that the deadline can be missed more than once but still result in failures. Soft real-time constraints refer to ones that missing deadlines merely leads to degradation of user experience. The near real-time constraints refer to those with an intrinsic yet bounded delay introduced by data processing or transmission. Note that this does not make all near real-time constraints 'soft', such constraints can still impose hard requirements for processing within the permitted bounds.

Besides timing related issues, quality-critical constraints also include other factors, as shown in Figure 2.2, e.g., energy consumption and budget. Energy consumption is taken into considerations more from the Cloud provider side to increase infrastructure utilisation. Then the energy consumption of the physical infrastructure can be reduced within a certain threshold to orchestrate the application. On the contrary, from the Cloud customer perspective, the budget constraints limit the monetary cost to pay for the Cloud usage. Meanwhile, this cost can be reduced through reasonably exploiting and managing the resources.

In this thesis, **we mainly focus on the speed-critical, near real-time, and budget constraints, as the quality-critical constraints.**

2.2.2 The Terminology of Cloud Applications

We highlight three types of Cloud applications in this thesis. Figure 2.3 illustrates what the relationship among these types of applications is.

Task-based Applications refer to applications which mainly running in a short term [92]. This type of application takes specific inputs and is executed for a specified period. It can be software testing and performance measurement applications, which run for a short time and generate measurement results. They can also be scientific applications, e.g., scientific workflows [12] of computing tasks. Task-based applications

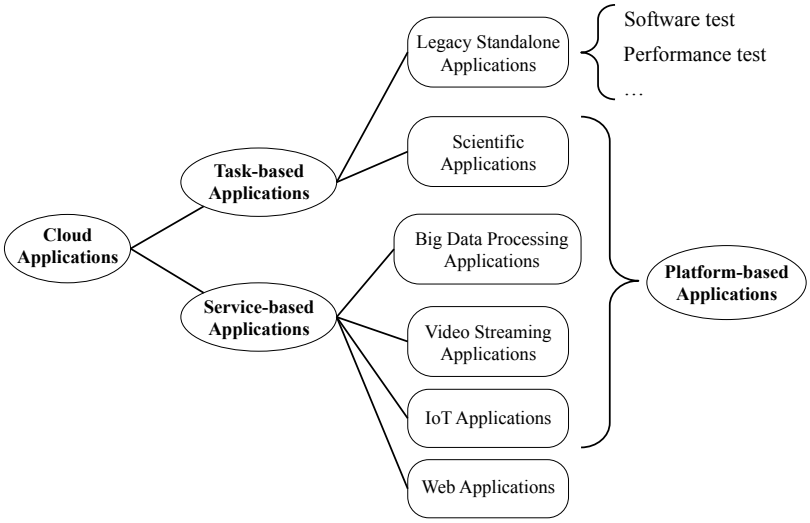


Figure 2.3: The terminology related to Cloud applications

are typically managed via schedulers over pre-defined infrastructure, e.g., computing clusters. In Cloud environments, the task-based application can either share infrastructures with the other users or customise a specialised one to satisfy the application quality requirements.

Service-based Applications refer to the applications running persistently for a long term as a service [75] . The lifecycle of the application does not end after being invoked. It often stays idle and waits for the next invocation. The traditional application of this type is the web application, which handles the request and feedback with the content. Meanwhile, big data processing, video streaming, and Internet of Things (IoT) applications are emerging.

Platform-based Applications rely on a specific platform. For example, HTCondor[12] and Pegasus[13] are popular scientific workflow execution platform. Hadoop[14] [117] is a well-known distributed platform to perform MapReduce [29] tasks for data processing. CometCloud [31] is able to support video streaming [120]. Kubernetes[10] [53] can be leveraged to develop and operate IoT applications using flexible containers. Hyperledger Fabric[15] [5] is a permissioned blockchain platform for smart contract execution. These platforms are usually operated on the top of a private cluster of VMs or provided by some Cloud providers in the manner of PaaS. The portability of such type of application is high, since the application is able to run, as long as the required platform is provided. However, it also requires the infrastructure underneath the platform to be set up in advance. The underlying infrastructure is often fixed, without the ability to scale at runtime.

[12]https://research.cs.wisc.edu/htcondor/
[13]https://pegasus.isi.edu/
[14]https://hadoop.apache.org/
[15]https://www.hyperledger.org/projects/fabric

It is worth mentioning that this thesis presents case studies with all above three types of applications, and those applications are abstracted from the use case of EU projects, e.g., the Euro-Argo[16] use case [72] of ENVRIFAIR[17] project, the eddy covariance data processing service[18] of ENVRIplus[19] and VRE4EIC[20] projects, and the disaster early warning use case [133] of SWITCH[11] project.

2.3 DevOps for Cloud Applications and Infrastructures

DevOps [13] 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 [116]. However, traditional software engineering approach for application development and operation focus on the collaborative work over the pipeline of testing, integrating and delivery during the application lifecycle. They often treat the underlying infrastructure for the application as a preset and static cluster. There are separate teams installed for software development and infrastructure operation, which often have high costs for maintaining and operating the infrastructure.

In Cloud environments, the OS level virtualisation provides a flexible way to encapsulate and deliver applications. It provides high portability for deploying application components and makes the automation of the DevOps pipeline possible. Most of the current Clouds do not directly provide virtual infrastructures at the OS virtualisation level. Thus in most cases, VM based infrastructures still need to be provisioned in advance to construct a cluster for hosting containers.

Nevertheless, at the hardware virtualisation level, the limited infrastructure programmability and controllability provided by Clouds make it possible to mitigate the gap between the application and infrastructure, and further involve the infrastructure operation into the software DevOps lifecycle.

2.3.1 Related Academic Research

The topic of Cloud infrastructure control and optimisation has attracted lots of research attention during the past years. For example, Sreekrishnan et al. [111] propose a model to deploy applications on hybrid Clouds and yield the best-fit hosting combination. Hassan et al. [136] focus on the algorithm selecting a proper Cloud to run the task according to the geographical location of the data centre. Min et al. [41] put forward eight recovery patterns for sporadic operations on public Cloud to maintain the service quality of the application. Xiaofeng et al. [114] try to optimise the makespan and the reliability of a workflow application through evaluating the resource reputation. Alexey et al. [58] conduct 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.

[16] https://www.euro-argo.eu/
[17] https://envri.eu/envri-fair/
[18] https://wiki.envri.eu/display/EC/IC_13+The+eddy+covariance+fluxes+of+GHGs
[19] https://www.envriplus.eu/
[20] https://www.vre4eic.eu/

There is existing work on enhancing applications to program and control the Cloud virtual infrastructure. CodeCloud [22] consists of an Infrastructure Manager (IM) [23]. IM provides some specific REpresentational State Transfer (REST) APIs to control each individual VM. Based on this, CodeCloud leverages Cloud Job Description Language (CJDL) to describe the application and the elasticity of the infrastructure. CloudPick [27] is a system that considers the high-level constraints of the application on the infrastructure, including deadline and budget. However, these systems work as centralised services asking users to upload their Cloud credentials, which requires trust in a third party. mOSAIC [91] is a deployable platform providing model-driven Cloud application development. CometCloud [31] provides a framework for heterogeneous Clouds to deploy several programming models, such as master/worker, map/reduce, and workflow. A video analytics system [120] is a notable application scenario of CometCloud. However, these tools need infrastructure resources provisioned in advance, without the controllability on the underlying infrastructure at runtime. Spiros et al. [70] leverage the adaptability of the network to optimise the data transfer, but the control on the switch of the data centre is required, which is not practical for public Cloud environments from the customer perspective.

2.3.2 Related Industrial Tools

From the perspective of industry, there are also many DevOps tools or frameworks proposed for the Cloud application to manage the infrastructure. Figure 2.4 illustrates the respective industrial frameworks and tools leveraged in the Cloud DevOps lifecycle for the application development and the infrastructure management, especially focusing on the programmability and controllability they provide.

As shown in Figure 2.4, with the IaaS Cloud service model, different Clouds or tools, such as OpenNebula[21] or OpenStack[22], provide different virtual infrastructure functions, i.e., APIs, to access their physical resources through the hardware virtualisation. This basic controllability of provisioning or terminating one VM is leveraged to afford controllability for the higher-level application. Tools, such as Libcloud[23] and jclouds[24], 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 a cluster of VMs, some Clouds provide a tool to describe several VMs, e.g., CloudFormation[25] of Amazon EC2[7] 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 such as Chef[26], Ansible[27], and Puppet[28], are developed to provide programmability for describing infrastructure from different Clouds. Among these tools, Chef and Ansible more focus on application deployment and configuration. They standardise the con-

[21] https://opennebula.org/
[22] https://www.openstack.org/
[23] http://libcloud.apache.org/
[24] https://jclouds.apache.org/
[25] https://aws.amazon.com/cloudformation/
[26] https://www.chef.io/
[27] https://www.ansible.com/
[28] https://puppet.com/

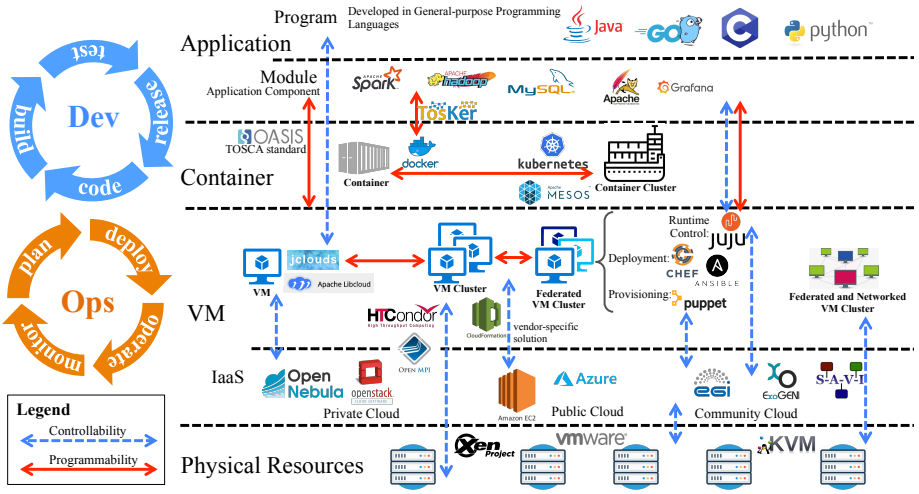


Figure 2.4: The illustration of related industrial tools considering the programmability and controllability they provide in the DevOps for Cloud applications and infrastructures

figuration 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" [83]. Nevertheless, 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[29] takes one step further to realise an application-defined infrastructure through describing application components and their hosting VMs, where the components are some typical software modules. The controllability of the infrastructure, like auto-scaling, can also be leveraged to ensure the QoS of a particular 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 [106] as the domain-specific language, which requires the developer to master the Ruby programming language.

Another aspect of work only focuses on the programmability and controllability between the container level and the application level. When treating containers as the infrastructure, containers are more flexible and lightweight to provide the infrastructure controllability and programmability compared to VMs. In particular, Docker[30] provides a VM-like environment to make the container closer to a lightweight VM. Kubernetes[10] and Mesos[6] further enhance the programmability to describe dependencies among several containers and orchestrate them to form a virtual cluster. Besides, Topology and Orchestration Specification for Cloud Applications (TOSCA)[31] [18], of which the

[29] https://jaas.ai/
[30] https://www.docker.com/
[31] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#overview

syntax is based on YAML Ain't Markup Language (YAML), is proposed to standardise the description of dependency among the application components and its underlying infrastructure, e.g., 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 [19] implements an engine, which regards Docker as the infrastructure. Though the Docker can provide a VM-like environment, it is still more close to software virtualisation, e.g., it shares the kernel of the host machine, which is the same for containers. Moreover, it usually 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 [55] can only provide the container-level scheduling, without the ability to manipulate the VM. The relation between the VM-level DevOps and container-level DevOps is also illustrated in Chapter 8 as Figure 8.1.

Other tools or frameworks, such as HTCondor[12] and Open MPI[32] [43], mainly focus on how to conduct applications on a fixed distributed system. These tools more concentrate on the application perspective, empowering the application to run in parallel and fit the distributed environment. For example, Jonghwan et al. [57] utilise a private cluster of seven physical machines to orchestrate a VoIP-based multimedia service with HTCondor. Nevertheless, they are not designed for Cloud environments, and therefore, lack infrastructure controllability. Another option to mitigate the DevOps gap between the application and the infrastructure is to build the entire computing stack from the physical hardware level, which means the infrastructure should be specifically designed. For example, SAVI (System Architecture Virtual Integration) [63, 64] builds up a test-bed for IoT. It leverages OpenFlow [80] to consider the network topology of the virtual infrastructure. Kraken [42] and SWMOA [99] also take the network into account when provisioning virtual infrastructures over multiple private data centres. However, all these solutions are applied to private data centres, where the hardware in the data centre, such as switches and routers, must be fully controlled. Hence, these solutions are not feasible in the context of public Clouds, which can only afford limited general controllability on the physical hardware infrastructure.

2.4 Blockchain and Smart Contract

The performance of the resources provided by Cloud computing is often not deterministic, as it is a multi-tenant and shared computing environment. Also considering Cloud computing is a business model, SLA plays a crucial role in building the contract between the provider and the customer to ensure the infrastructure performance. However, both parties in current SLA lack trust with each other and need a mechanism to reduce the potential risk. The blockchain technology seems a promising solution. Therefore, we introduce the basic background knowledge of blockchain and smart contract for Chapter 7.

[32] https://www.open-mpi.org/

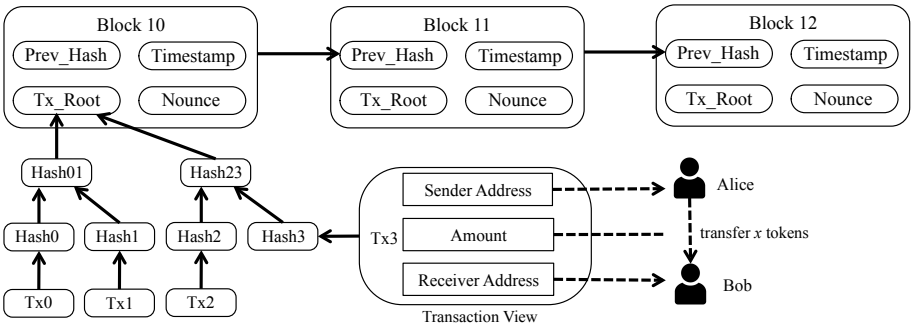


Figure 2.5: The basic structure of first generation blockchain

Blockchain [128] is an emerging technology to make every participant having trust in a decentralised ledger through the consensus algorithm, such as Proof of Work (PoW), Proof of Stake (PoS), and Practical Byzantine Fault Tolerance (PBFT). For the public blockchain, all the information stored in the ledger is public, verifiable, and immutable.

Figure 2.5 illustrates the basic structure of the first generation blockchain [85]. It demonstrates that the blockchain is literally a set of blocks chained by some special hash values. The block contains all the transactions, which identifies how many tokens are transferred from a specific wallet address to another (from Alice to Bob in the example of Figure 2.5). Here, the wallet address is derived from the public key, and only the user who keeps the private key can sign the transaction. The current balance of that wallet can be calculated from its transaction history. Then every participant in the system is able to verify the transaction. After collecting enough valid transactions, every participant has to tune the "Nounce" value in the block to find a particular hash value, which contains several zeros in the beginning. The finding process is energy-consuming and time-consuming. As long as the special value is found, the block should be published to peers immediately in order to be accepted. For the purpose of incentivation, the one who finds this value can embed a transaction in the block to announce a certain number of rewarding tokens to itself. In the end, only the longest chain in the system network would be accepted to solve the consensus issue. This design ensures that the transaction in the previous blocks cannot be modified. If the attacker would like to change the transaction value, the hash value of that block would also change, which causes the chain reaction to change all the hash values of succeeding blocks. However, the computing power of the attacker is not able to build another longest chain to replace the current one. Therefore, as long as the system is distributed enough and no one can compromise more than half of the system's computing power, all the participants in the system have trust in the transactions recorded inside the blocks.

The application of this first generation blockchain is Bitcoin[33]. The first block was generated in 2009. According to the algorithm, around every 10 minutes, a new block is calculated and found. Due to the size of the block, there is a maximum number of transactions that can be included in one block. Hence, the maximum throughput of the

[33]https://bitcoin.org/

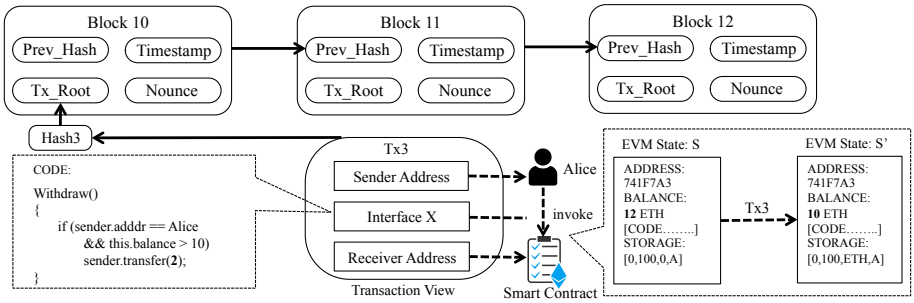


Figure 2.6: The basic structure of second generation blockchain for smart contract

Bitcoin system is seven transactions per second.

The limitation of the first generation blockchain is that the transaction can only support token transferring. In order to achieve more complicated functionality, the second generation blockchain is introduced and makes the smart contract possible. Figure 2.6 depicts the basic structure of the second generation blockchain. Its key difference from the first generation is that the wallet address can also be owned by a smart contract instead of only humans. Then the transaction between two wallet addresses can represent that someone wants to invoke a particular interface of the smart contract. For each smart contract, a corresponding byte string is exploited to represent the VM state for executing the smart contract. As long as the transaction is inserted in the blocks, the VM state for the smart contract is changed through invoking the interface specified in the transaction. The current VM state of a specific smart contract can also be validated and derived from its transaction history, as the transactions contain all the interfaces having been invoked. To be specific, these transactions are recorded in the blockchain, where the information is immutable and irreversible. Meanwhile, the interface is allowed to be programmed with specific conditions to accomplish a complex task. For instance, the example smart contract in Figure 2.6 is programmed only to allow Alice's wallet address to withdraw 2 tokens from the contract when the balance of the smart contract's wallet is above 10 tokens. In the example shown in Figure 2.6, all the conditions are met, and the transaction is valid to be committed on the blockchain. The transaction then changes the VM state, reducing its balance from 12 to 10 tokens.

Thus, blockchain provides a worldwide computer to execute the smart contract transparently, immutably, and credibly. Everyone can participate in the blockchain network and verify the transactions to execute the smart contract. We, therefore, believe that no one can stop smart contract running or fudge the results. This open and verifiable mechanism is also where the trust comes.

The typical application of this second generation blockchain is Ethereum[34] [21], starting from 2015. The VM state for the smart contract is termed as Ethereum Virtual Machine (EVM). Moreover, the PoW consensus algorithm of Ethereum allows generating a new block around 15 seconds to enhance the smart contract execution efficiency. The system throughput is, therefore, improved to around 25 transactions per second.

[34]https://www.ethereum.org/

However, it is still challengeable when the condition programmed in the interface of the smart contract relies on the off-chain event result. For instance, if the interface in Figure 2.6 is programmed as only Alice's account can withdraw 2 tokens, when the temperature is above 10 degrees. In this case, the problem is that the smart contract cannot detect the actual temperature to judge the condition in a trustworthy way, because the temperature is a real-world event. It is worth mentioning that the extension of our work in Chapter 7 partially mitigate this gap.

2.5 Challenges

We summarise this chapter as four highlighted challenges corresponding to the above four sections, respectively.

- *Insufficient models for programming and controlling Cloud virtual infrastructures.*

  Though the IaaS model can provide Cloud customers with possibilities to program and control the virtual infrastructure at the level of hardware virtualisation, i.e., the VM level, these abilities are not sufficient at the entire infrastructure level for application developers to leverage. Moreover, the diverse APIs from different Cloud providers make it difficult to enable applications across the federated Clouds. Without an effective infrastructure programming and control model, the application developer and the infrastructure operator cannot seamlessly work together by automating the pipeline of Cloud application testing, integration, provisioning and deployment in the DevOps lifecycle.

- *Short of Cloud management support to satisfy the applications' quality-critical constraints.*

  To operate distributed Cloud applications, such as IoT and live streaming applications, the factor influencing the application QoS is not only the computing capacity of the infrastructure but also the network capacity, e.g., the communication latency and bandwidth. While the Cloud data centres are nowadays continuously growing, application developers will have more options to select and to provision the infrastructure. The budget constraints can also be addressed through on-demand managing the infrastructure since the pay-as-you-go business model of Clouds. Nevertheless, most of current tools or frameworks cannot support managing different Clouds in an extensible and efficient manner.

- *Lack of a framework to seamlessly program and control the Cloud application during the DevOps lifecycle.*

  The current Cloud computing research focuses either on solutions to manage the physical infrastructure within data centres in the view of the Cloud provider, e.g., Software-defined Network (SDN), VM consolidation and migration, or on optimisation of the Cloud application performance from the Cloud user perspective. Among the various industrial tools and frameworks, some of them only concentrate on managing an individual VM, and others focus on particular steps in DevOps lifecycle, such as provisioning and deployment. These tools are not sufficient to automate the entire process for addressing the requirements of the application. In essence, there are multiple phases for operating the application on the Cloud infrastructure.

- *Difficult to convince both the Cloud provider and customer of the SLA violation.*

Currently, SLA is enforced manually and lacks trust. The blockchain technique, specifically the smart contract, provides a potential solution to automate the SLA enforcement credibly. If implemented, it is crucial to detect the SLA violation and report it to the SLA smart contract. However, it is impossible for the SLA smart contract itself to detect this off-chain event, which is still a gap mentioned in Section 2.4 for the smart contract. In essence, the smart contract is static and requires entities to invoke with the event results as inputs, e.g., whether the SLA is violated. Hence, the hurdle is to find entities who can be trusted by both sides of SLA, and keep them performing unbiased and trustworthy SLA violation detection to trigger the SLA enforcement.

3

Systematic Cloud Infrastructure Programming and CloudsStorm Framework Design

the programming phase

For the background introduction, we have discussed different Cloud service models. Among them, the emerging Infrastructure-as-a-Service (IaaS) Cloud service model allows developers to flexibly provision and terminate virtual machines (VMs) with suitable types and location specifications. However, taking an Internet of Things (IoT) application as an example, the developer requires: 1) a unified description for customising the infrastructure from any possible Cloud to be close to the mobile devices; 2) the ability to describe the infrastructure operations, such as provisioning and scaling, for programmatically adjusting the infrastructure according to the distribution of the mobile devices; and 3) identifying the non-functional requirements reacting to the fickle workload because of the mobility of the devices.

In this chapter, we first analyse the programmability requirements for infrastructure programming and introduce the state of the art. We then propose our design principle and the Cloud virtual infrastructure programming model. Afterwards, we present our CloudsStorm[1] framework and the programmability design, enabling developers to customise the infrastructure and program the infrastructure operations into their Cloud applications. Finally, we show the example code of how the description of high-level infrastructure operations is achieved.

This chapter is based on:

- **Zhou, H.**, Hu, Y., Su, J., de Laat, C., Zhao, Z., "CloudsStorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures", In *International Conference on Cloud Computing*, pp. 265-280. Springer, Cham, 2018.
- **Zhou, H.**, Hu, Y., Ouyang, X., Su, J., Koulouzis, S., de Laat, C., Zhao, Z., "CloudsStorm: A Framework for Seamlessly Programming and Controlling Virtual Infrastructure Functions during the DevOps Lifecycle of Cloud Applications", Journal of *Software: Practice and Experience*. Wiley, 2019.

[1] https://github.com/zh9314/CloudsStorm

3.1 Cloud Infrastructure Programming

Cloud computing provides an elastic approach to provisioning and terminating resources as the underlying virtual infrastructure for operating applications. However, the elasticity is not commonly used, because most Cloud virtual infrastructures are provisioned manually in advance and treated as the fixed traditional physical infrastructures. Meanwhile, developers still more focus on the application development only. It is, therefore, urgent to explore the programmability of describing and controlling the infrastructure resources, in order to programmatically utilise the resources on demand.

3.1.1 Programmability Requirements Analysis

When developing quality-critical Cloud applications, we should consider satisfying the application requirements from two aspects: functional requirements and non-functional requirements. Hence, we analyse the programmability to describe and operate the infrastructure as follows.

Functional Requirements:

For the functional requirements, the programmability is required to customise the infrastructure, such as the resource capacity, the geolocation distribution, and the network topology. Besides, the programmability is required to describe the operations performed on the infrastructure, including provisioning, terminating, and scaling. Therefore, we analyse the programmability from the following three levels.

1. **Design level.** The design-level programmability is required to describe underlying infrastructure to specify the computing resources' (VM) quantity, types, locations, and network. For instance, a live streaming application requires 2 VMs with the large type from EC2 and 4 VMs with the medium type from ExoGENI Cloud[2] [8]. All of them need to be connected with a customisable private network to transfer streaming data. It is crucial for the developer to customise this infrastructure topology with Cloud and data centres.
2. **Infrastructure level.** The infrastructure-level programmability is required to automate the process of provisioning the VMs, configuring the network, and deploying the applications. It also should provide high-level infrastructure operations, such as provisioning, scaling, and failure recovery. Meanwhile, the parallelism of the operation should be easily specified to achieve efficient control. Still, with the example of the previous one, the developer should be able to program how to provision the described infrastructure, such that the 2 VMs from EC2 can be provisioned first. Also, the scaling operation should be defined to adjust the infrastructure when the input of streaming data suddenly increases.
3. **Application level.** The application-level programmability is required to directly adjust the infrastructure to fit for the application constraints or workload. Because of the pay-as-you-go business model, the over-provisioned infrastructure results in an extra monetary cost. Taking the example of running the Hadoop based big

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

data processing application on Clouds, when there is no input data to process, the extra VM resources waste money since a traditional Hadoop platform is based on a fixed cluster. That is, the Hadoop application should be able to program and customise a required number of VM resources, according to the input workload, e.g., the data size. Then during runtime, in order to reduce the monetary cost, the Hadoop application can dynamically adjust the infrastructure resources, i.e., scaling out or in VMs, according to the input data size.

Non-functional Requirements:

For the non-functional requirements, the programmability of specifying the conditions that the application should respond to is required for adjusting the infrastructure to keep satisfying the quality-critical constraints. Therefore, the developer should be able to define the metrics as thresholds and the infrastructure operations to further respond. The detailed metrics can be classified as follows.

1. **Infrastructure Metrics.** This type of metrics is leveraged to define the system status of the infrastructure, such as availability, network connection, CPU and memory utilisation. This information is essential for the infrastructure to maintain system performance through automatic scaling and recovery. For example, a disaster early warning application requires to recover the infrastructure within a time constraint if some computing nodes are detected to be unavailable due to the Cloud failures. It is highly important for this type of quality-critical applications to keep functioning constantly.
2. **Application Metrics.** This type of metrics is leveraged for the application to define more fine-grained metrics, such as average job completion time and throughput. The specific name of the metric should be customisable for the application to define the condition in their own metrics. For instance, a Hadoop application needs to define the average job completion time as a threshold for scaling the infrastructure. If the measured data in this metric downgrades, probably because of a sudden intensive workload, the scaling operation then can be performed to bring the application back to the expected performance.

3.1.2 State of the Art

In this section, we investigate the state of the art corresponding to those aspects.

For the functional requirements, the design-level programmability currently is addressed by Topology and Orchestration Specification for Cloud Applications (TOSCA)[3] standard [18]. It unifies the syntax of describing the application components and the dependency on the infrastructure resource, e.g., two components should be put in one VM. However, it is not sufficient to further define the location (i.e., the Cloud and data centre) and the network topology of the VM cluster. It is because that TOSCA more focuses on the application components specification, instead of the detailed Cloud infrastructure description. CloudFormation[4] provides a template to describe the required VMs and resource, but it is a vendor lock-in solution that only applies to EC2 Cloud.

[3] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#overview
[4] https://aws.amazon.com/cloudformation/

The infrastructure-level programmability is partially addressed by some infrastructure management tools. For example, jclouds[5] provides a unified Application Programming Interface (API) for different Clouds. However, these API-centric [116] tools focus on the programmability on each individual VM instead of the entire infrastructure topology. In comparison, there are also some environment-centric [116] tools to help developers orchestrate their applications, which include Puppet[6] and Chef[7]. These tools more focus on cluster management, i.e., deployment and configuration, instead of provisioning and scaling automation. Some academic research also proposes architectures for developing applications on Clouds, e.g., CometCloud [31] and mOSAIC [91]. Nevertheless, 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. Specifically, none of them provides an independent syntax to describe the infrastructure operations, which does not rely on a specific programming language.

As to the application-level programmability, however, none of the current software Development and Operations (DevOps) tools supports to embed this type of infrastructure programming logic directly to the application logic, to the best of our knowledge.

For the non-functional requirements, the monitoring based solution is commonly adopted by Cloud systems [103]. Most of the tools and academic research can support specifications of different level of metrics to monitor the Cloud infrastructure. However, they are not able to program the actions according to the monitoring information. For instance, Kwon et al. [73] implement a monitoring system for Cloud infrastructures but only with a dashboard to visualise the monitoring information. The monitoring tool, Jcatascopia [108], allows the user to define the threshold and an "action" field specifying the reaction. Nevertheless, the action is simply about whether the user should be notified. This kind of action is not capable of adjusting the infrastructure as programmed.

3.2 Cloud Virtual Infrastructure Programming Model

Based on the requirement analysis, we discuss our design principles and propose the Cloud virtual infrastructure programming model based on the basic Cloud Virtual Infrastructure Function (VIF) in this section.

3.2.1 Design Principles and Programming Model

The design-level programmability can be addressed by a particular domain-specific language adding descriptions of Clouds and data centres. On the other hand, the programmability for infrastructure-level, application-level, and non-functional requirements all needs to define and program the operations, which are performed on the infrastructure. Hence, the programmability design should provide the ability to describe both the infrastructure topology and operation.

In order to be leveraged as a programming approach by developers, the syntax of the programmability design should be **human-readable and easy to learn**. Inspired by the

[5] https://jclouds.apache.org/
[6] https://puppet.com/
[7] https://www.chef.io/

TOSCA standard, we adopt the YAML Ain't Markup Language (YAML) [14] format to design the programmability for both of the infrastructure topology and operation. Besides, for the application-level programmability, we design the operations with some general-purpose programming languages, which can be more seamlessly embedded into the original application logic. The detailed syntax is demonstrated in Section 3.3.

Though both of the infrastructure topology and operation can be described, it is still challenging to program the infrastructure operations on different Clouds, since they provide different APIs to access their Cloud resources. Especially for the high-level operations, some Clouds have provided the functionality, and some others have not, e.g., EC2 Cloud provides the auto-scaling functionality through allowing the customer to define a scaling group, and ExoGENI cannot perform auto-scaling. Moreover, since the Cloud market is booming, we cannot support all the Clouds with their specific operations. Therefore, the key challenge here is how to enable the unified operation description extensible to be performed on various Clouds.

On the other side, the operation efficiency should also be considered through performing the operations in parallel. The complexity of parallel programming is another hurdle.

Inspired by the programming model of MapReduce [29] and functional programming, we propose our infrastructure **programming model based on basic Cloud VIFs**. Using MapReduce programming model, the complex data operation is decoupled into the user-defined map functions and reduce functions. Then the complicated data processing logic, including the parallelism, is handled by the framework. Our idea is to model the Cloud infrastructure operations into some basic ones, which are commonly provided by different Clouds. These basic operations for a particular Cloud can be programmed as functions given by the developer. Then the functions can be plugged into our framework to be leveraged for interpreting and realising the high-level infrastructure operations. In addition, the parallelisation is also handled by the framework.

According to the programming model, we need to model the public IaaS Cloud and the basic Cloud VIF provided by any Cloud. Based on the IssS Cloud service model, any IaaS Cloud at least provide the functionality of provisioning one VM and terminate one VM. After a VM is provisioned, the VM must be remotely accessible. Otherwise, the computing resource cannot be used. Therefore, we model the public Cloud with three basic Cloud VIFs: *VM Provisioning*, *VM Configuration*, and *VM Terminating*. These are minimal functions, which can be provided by any public IaaS Cloud.

Based on these minimal functions given by the developer, how the high-level infrastructure operations can be constructed is demonstrated in Section 3.4. Hence, the complex operations performed on the infrastructure are extensible to different Clouds. Besides, Section 5.3.1 presents how these functions are leveraged in parallel based on the implementation, which addresses the efficiency issue of the operations.

3.2.2 Basic Cloud Virtual Infrastructure Functions

Figure 3.1 illustrates the basic Cloud VIF of provisioning a single VM. According to the IaaS model, the customer should be able to remotely negotiate with the controller of a particular data centre from some Cloud. In the negotiation, the customer first needs to authenticate themselves with the key, $\mathbf{KEY}_{cloud}$. This key is the access credential given

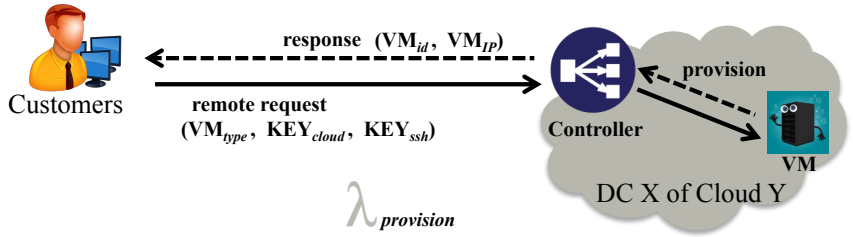


Figure 3.1: Basic Cloud Virtual Infrastructure Function: *VM Provisioning*

by the Cloud to identify the customer, which is crucial for the Cloud provider to charge the customer according to the usage. Then the customer needs to request a specific VM type supported by the Cloud. It is worth mentioning that different VM types are with various computing capacities, but also result in different prices. In addition, the customer can specify a key, $\mathbf{KEY}_{ssh}$, for accessing the provisioned VM through Secure Shell (SSH) later on. Afterwards, the controller receives the request and provisions the corresponding VM from this data centre. Finally, the customer is notified with the $\mathbf{VM}_{id}$, which is an identification string to uniquely refer the VM, and $\mathbf{VM}_{IP}$, which is a public IP for accessing the VM remotely through the Internet. Meanwhile, the controller starts billing the customer for this VM.

Figure 3.2 illustrates the VM configuring operation after provisioning. The customers are able to login the VM with the preconfigured key, $\mathbf{KEY}_{ssh}$, through the public IP address, $\mathbf{VM}_{IP}$, using SSH. It is worth mentioning that SSH is a common technology adopted by all the public IaaS providers to empower customers with the ability to access their VMs remotely and securely. Moreover, the concrete operations include the network configuration, application deployment and execution. These operations can be user-defined and mainly adjust the application execution.

Figure 3.3 illustrates the third basic Cloud VIF of terminating a single VM. This function is also a remote request from the customers to terminate a VM. $\mathbf{KEY}_{cloud}$ is still required for authentication. $\mathbf{VM}_{id}$ is used to identify the specific VM. Then the controller is able to terminate the VM and release the resources occupied by the VM. Meanwhile, the controller would stop billing the customer for this VM. In the thesis, this operation is also termed as “delete”.

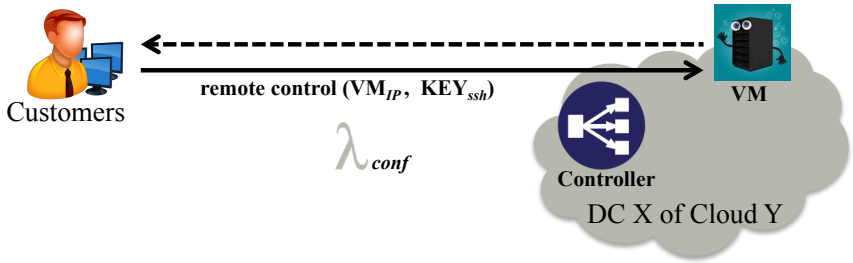


Figure 3.2: Basic Cloud Virtual Infrastructure Function: *VM Configuration*

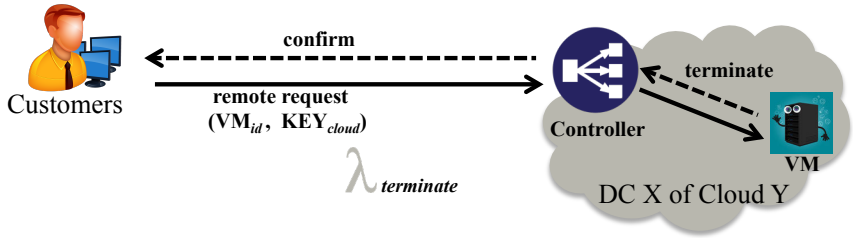


Figure 3.3: Basic Cloud Virtual Infrastructure Function: *VM Terminating*

Among these basic Cloud VIFs, the functions of *VM Provisioning* and *VM Terminating* are provided by the Cloud. These two functions are usually in the form of REpresentational State Transfer (REST) API to perform the actual remote request. Specifically, each Cloud provides a different Software Development Kit (SDK) to programmatically invoke these REST requests. The diversity of these interfaces implemented in different SDKs hinders the developer to perform the infrastructure operations on federated Clouds. In addition, some Cloud provides extra functions of starting and stopping a VM. The process is similar to the provisioning and terminating a VM, but not all Clouds support this. We, therefore, cannot include these functions as basic ones.

On the other hand, the basic function of *VM Configuration* does not rely on the Cloud specification. As shown in Figure 3.2, this operation does not need to involve the controller of the data centre. However, the commands for network configuration and application deployment are related to different operating systems (OS) of the VM. Anyhow, the diversity of OS is easier to handle, considering the set of mainstream OS is much smaller than that of Clouds.

3.3 Cloud Infrastructure Programmability Design

In this section, we introduce four types of infrastructure programming code designed by us to provide the Cloud infrastructure programmability systematically. The proposed infrastructure code can be leveraged by Cloud application developers to program the infrastructure descriptions and operations. Corresponding to the different levels of programmability based on the requirements analysis in Section 3.1.1, we propose the design of "*Infrastructure Description Code*", "*Infrastructure Execution Code*", "*Infrastructure Embedded Code*" and "*Runtime Control Policy*", respectively.

3.3.1 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 centres, we propose a partition-based infrastructure management mechanism. This mechanism adopts a hierarchical description of the infrastructure topology, which is classified into three levels, i.e., the levels of top-topology, sub-topology, and VM. Figure 3.4 shows

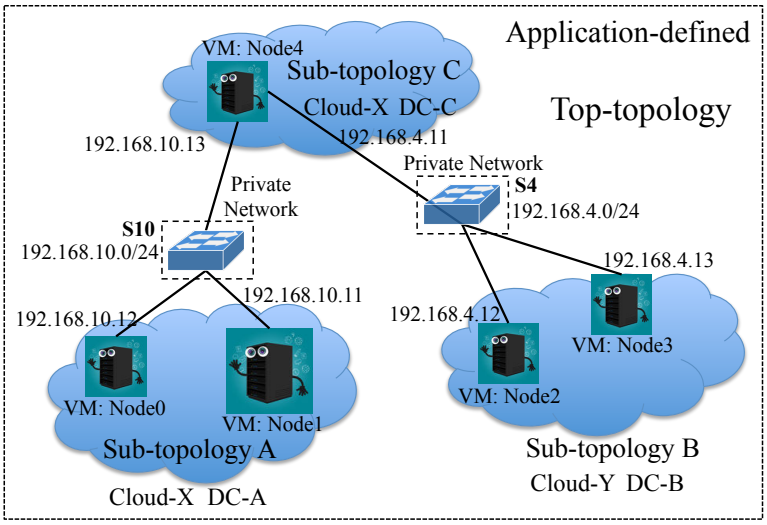


Figure 3.4: An example of partition-based infrastructure management

an example topology to demonstrate this mechanism, where DC is the acronym of Data Centre. The lowest level is the VM level. It enables the developer to customise a specific VM concerning different aspects. The level in the middle is the sub-topology level. It provides a description of several VMs provisioned in the same data centre. The top level is the top-topology level. It includes all the sub-topologies and describes the network connections among all VMs. The reason to differentiate the level is to manage the infrastructure more flexibly and efficiently. On one aspect, infrastructure operations are not intended to be applied to the entire infrastructure, i.e., the top-topology. For instance, we only want to terminate all the VMs from a specific data centre. We need the ability to specify that part of the infrastructure. On the other aspect, the operation performed on a sub-topology is applied to all the VMs inside in parallel. Hence, we only need to terminate the sub-topology from that data centre at the sub-topology level, instead of terminating the VMs one by one at the VM-level.

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: 1) the actual network is made transparent to the application. Still taking the example of operating the Hadoop application on Clouds, 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 case study of task-based applications in Section 6.1 of Chapter 6 also demonstrates this benefit; 2) the infrastructure for the application is always reproduceable regardless of the geographical information, i.e., data centres. In this case, the same network for the Hadoop cluster is provisioned every time, no matter which data centre the VMs come from, even come across different data centres; 3) get rid of the provisioning dependency. For example, we do not have to provision a database VM before a web server VM, in

Syntax 1 "*Infrastructure Description Code*" for sub-topology description

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

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 5.4.2 of Chapter 5.

In addition, the switches placed in Figure 3.4 only illustrate how VMs are connected. The technique detail to provision such networked infrastructure using overlay network mechanisms is demonstrated in Chapter 4.

In order to describe the networked infrastructure for the application, we adopt the YAML format to define the sub-topology 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 field. Symbol, '[]', is exploited to represent an optional field. Syntax 1 defines the sub-topology description, which is a list of "VMs". Every element contains the VM name, such as "Node0" or "Node1" in Figure 3.4. "nodeType" indicates the

Syntax 2 "*Infrastructure Description Code*" for 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
  - *
```

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. Thus, it can be omitted, if the following "CPU" and "Mem" are specified. The framework 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 would be interpreted as 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.

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, i.e., $\$User$, to access all the VMs, no matter which Clouds 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" here is the application-defined name of a certain sub-topology, such as "A", "B" or "C" shown in Figure 3.4. "cloudProvider" and "domain" specify the concrete data centre where this sub-topology is hosted. For example, there is a $\$DC = California$ data centre from $\$Cloud = EC2$. Actually, one Cloud provider usually has multiple data centres and they are distributed in different geolocations, e.g., current data centres of EC2 Cloud span 21 geographic regions around the world[8]. "status" indicates the status of this sub-topology. 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 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 3.4, 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.

3.3.2 Infrastructure Execution Code

The "*Infrastructure Execution Code*" is proposed as a means to provide the application developer with the infrastructure-level programmability. This type of code enables the application developer to directly perform operations on the infrastructure, such as provisioning, terminating of particular resources, and executing commands on a specific VM. It is separated with the application logic and also based on the YAML format. Comparing to the above 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 essential for the application developer to programmatically provision the infrastructure, deploy and executing their applications on Clouds.

The "*Infrastructure Execution Code*" is basically a set of operations defined sequentially in a list. In order to combine these basic operations to accomplish a complex task,

[8] https://aws.amazon.com/about-aws/global-infrastructure/

Syntax 3 "SEQ" type of "*Infrastructure Execution 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: $\$Object_1$ [||$\$Object_2$]...

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 by 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 shown in Section 3.4 with high-level infrastructure operations and in Section 6.2 with case studies. "Objects" then refers to the set of objects. To define this set, we adopt the symbol '||' from parallel λ-calculus [38] 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 terminate Cloud resources; 'start' and 'stop' can be used when the Cloud

Syntax 4 "LOOP" type of "*Infrastructure Execution 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: $\$Object_1$ [||$\$Object_2$]...
  - *

Syntax 5 Complete "*Infrastructure Execution Code*"

```
[Mode: 'LOCAL | CTRL']
InfrasCodes:
- CodeType: 'SEQ | LOOP'
- *
```

supports the operation of starting or stopping a VM; 'put' and 'get' operations are used to upload or download data to or from a particular 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 sub-topology level failure recovery.

In order to finish complex tasks, we provide the "LOOP" type of code 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 executing this loop. "Duration" is defined as the maximum amount of time for executing this loop. "Deadline" is defined as a specific timing to exit this loop, which is represented as Unix timestamp. Among these three conditions, at least one must be defined for a "LOOP" type of code. If there are several defined, then the loop is ended when any one of the conditions is met.

The complete "*Infrastructure Execution Code*" is, therefore, an ordered combination of these codes, which is shown as Syntax 5. The optional field, "Mode", indicates whether a control agent is required: value "LOCAL" is for executing this infrastructure code in local machine without a remote control agent; value "CTRL" is for executing this infrastructure code on a remote control agent, instead of the local machine, and this agent will manage the infrastructure. This "CTRL" mode is similar to executing an application in a background mode. The merit and demerit of using a control agent are discussed in Section 3.5.3. If this field is omitted, the default mode value is "CTRL".

A case study of a task-based application demonstrating how the "*Infrastructure Execution Code*" is programmed is shown in Section 6.1 of Chapter 6.

3.3.3 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, provision, terminate, and scale resources of the infrastructure. The request made by the implemented interface is finally translated as a REST request to a control agent, which should provide specific 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

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
```

able to make corresponding REST calls. The complicated controlling logic is at the control agent side. Therefore, each VM in the infrastructure does not need the heavy libraries for executing "*Infrastructure Execution Code*" to perform operations. Another advantage is that operations on the infrastructure can be performed in parallel along with the application execution, because the actual infrastructure operation is then performed by the control agent after the application makes the REST request.

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 non-blocking, 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, if 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 with the "$executionID$" 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 of Section 6.2.3 in Chapter 6.

3.3.4 Runtime Control Policy

The "*Runtime Control Policy*" is harnessed to provide the programmability of defining thresholds for maintaining non-functional requirements, which is a common manner to be enforced through the monitoring information [103]. This policy defines a set of operations, which are performed when a particular condition is met. During the runtime, this policy should be managed by a control agent, and the infrastructure is, therefore, passively controlled according to this policy. Syntax 6 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 still contains two basic elements, "ObjectType" and "Objects", for indicating the objects, to which this policy is applied. Symbol, '||', is also leveraged to represent that this policy is applied to all the objects in parallel. Besides, 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.

Syntax 6 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: ...
  - *
- *
```

The "Metrics" is a set of key-value pairs: 1) the key of the pair defines the metric type to monitor, such as "CPU" for CPU utilisation, "MEM" for memory utilisation, "ALIVE" for availability detection, and "$\$String_1$" for application-defined metrics. To address the two levels of non-functional requirements mentioned in Section 3.1.1, the first three keys are to deal with the infrastructure metrics, and the last key is for the application metrics; 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 particular threshold. Then the final condition is met when the above 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 programmed infrastructure operations will be triggered and performed.

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 auto-scaling and failure recovery, is evaluated in Section 5.4 of Chapter 5. Moreover, "BudgetPerHour" allows the developer to add a monetary constraint for infrastructure management. Its value is in dollars, and it limits the maximum Cloud resource usage per hour. This field is essential to address the budget constraint, which is one of the important factors when operating quality-critical Cloud applications, according to the analysis in Section 2.2.1.

3.4 High-level Infrastructure Operations

In this section, we discuss the programmability design of three types of high-level infrastructure operations, i.e., horizontal scaling, vertical scaling, and failure recovery. Furthermore, we demonstrate that these operations can be realised through utilising the

basic Cloud VIFs above, which is in accordance with our programming model.

3.4.1 Horizontal Scaling

In the context of this thesis, horizontal scaling refers to adapting the computing capability of the infrastructure by changing the number of VMs. Scaling out means adding more VMs to the infrastructure to increase its capability. On the contrary, scaling in means terminating VMs and removing from the infrastructure to decrease the capability.

Code 1 shows how the "*Infrastructure Execution Code*" can be leveraged to define the operation of horizontal scaling. For this example, there are two sub-topologies named "XS" and "YS" in the entire infrastructure. When we want to scale out these two sub-topologies, Code 1 is the demo code. It is based on the "*Infrastructure Execution Code*", consisting of three code elements. The first element describes the operation to scale out sub-topology "XS" from the data centre at Washington, which belongs to the Cloud "ExoGENI". It specifies that the scaled sub-topology will be named as "scaled_0" in the infrastructure description. The second element describes the similar operation

Code 1 Example code for sub-topology level horizontal scaling

```
- CodeType: "SEQ"
  OpCode:
    Operation: "hscale"
    Options:
    - ReqID: "hs_req_1"
      CP: "ExoGENI"
      DC: "GWU (Washington DC, USA)"
      OutIn: "Out"
      ScaledSTName: "scaled_0"
    ObjectType: "SubTopology"
    Objects: "XS"
- CodeType: "SEQ"
  OpCode:
    Operation: "hscale"
    Options:
    - ReqID: "hs_req_2"
      CP: "ExoGENI"
      DC: "BBN/GPO (Boston, MA USA)"
      OutIn: "Out"
    ObjectType: "SubTopology"
    Objects: "YS"
- CodeType: "SEQ"
  OpCode:
    Operation: "hscale"
    ObjectType: "REQ"
    Objects: "hs_req_1 || hs_req_2"
```

Code 2 Example code for VM level horizontal scaling

```
- CodeType: “SEQ”
  OpCode:
    Operation: “hscale”
    Options:
    - ReqID: “hs_req_1”
      CP: “ExoGENI”
      DC: “GWU (Washington DC, USA)”
    ObjectType: “VM”
    Objects: “XS.Node0 || YS.Node1”
- CodeType: “SEQ”
  OpCode:
    Operation: “hscale”
    ObjectType: “REQ”
    Objects: “hs_req_1”
```

to scale out sub-topology “YS” from another data centre at Boston of “ExoGENI”. The name of this scaled sub-topology is not explicitly specified, which will then be automatically named. Meanwhile, all the scaled sub-topology copies will keep the same network topology as the original scaling sub-topologies. The network IP addresses of the scaled VMs are picked from the private addresses pool of the original subnet. The last element indicates that the above two operations are executed in parallel. It means that the new scaled sub-topologies of “XS” and “YS” will be provisioned and configured simultaneously. This element can also be divided into two elements. Each of them only contains one “Object”. Then, the operation of parallel scaling out changes to the operation of sequential scaling out.

Compared to other programming tools, our syntax is more straightforward and human-readable. For example, if adopting jclouds to automate the scaling process, the developer needs advanced programming skills in Java using the multi-thread technique to make the operation in parallel.

Above horizontal scaling is performed at the sub-topology level. For more fine-grained scaling, the “ObjectType” can be set as “VM” to perform VM level horizontal scaling. Code 2 is the example code. The scaled part is “Node0” from sub-topology “XS” and “Node1” from sub-topology “YS”.

Moreover, we analyse and derive that the operation of horizontal scaling can be achieved by using the basic Cloud VIFs as follows. No matter at the sub-topology level or the VM level, the scaling out operation first requires the basic Cloud VIF of *VM Provisioning* to provision the individual VM or all the VMs defined in the sub-topology. Then, the basic Cloud VIF of *VM Configuration* can be leveraged to configure the network to be connected with the predefined addresses. For the scaling in operation, it first requires the basic Cloud VIF of *VM Configuration* to detach all the other VMs' network connections from the target VMs which are going to be scaled in. Afterwards, the basic Cloud VIF of *VM Terminating* is leveraged to terminate the target VMs.

3.4.2 Vertical Scaling

In the context of this thesis, vertical scaling refers to adapting the computing capability of the infrastructure through directly changing the capacity of a particular VM. For the networked infrastructure, the network connection must remain the same. Scaling up means increasing a specific VMs' capability. On the contrary, scaling down means decreasing the capability. It is worth mentioning that we only consider scaling the computing capacity without migrating the tasks in this case. However, it is still useful for platform based applications to adjust the capacity for following jobs and tasks.

Code 3 is the example that how to perform vertical scaling. In this example, we still assume there are two sub-topologies "XS" and "YS" from ExoGENI. VM "Node0" is from "XS", and VM "Node1" is from "YS". Both of these two VMs belong the type of "XOMedium". According to the node type definition[9] of ExoGENI, they both have 1 core of CPU and 3G memory. Similar with horizontal scaling, there are three elements in this example Code 3. The first two elements are vertical scaling requests. There is no explicit definition to specify whether this operation is scaling up or down. The actual scaling direction is determined by the target CPU and memory capacity. In this case, the first operation in Code 3 is scaling down. The "XOMedium" VM is scaled down to

[9] https://wiki.exogeni.net/doku.php?id=public:experimenters:resource_types:start

Code 3 Example code for VM level vertical scaling

```
- CodeType: "SEQ"
  OpCode:
    Operation: "vscale"
    Options:
    - ReqID: "vs_req_1"
      CPU: "1"
      MEM: "1"
    ObjectType: "VM"
    Objects: "XS.Node0"
- CodeType: "SEQ"
  OpCode:
    Operation: "vscale"
    Options:
    - ReqID: "vs_req_2"
      CPU: "2"
      MEM: "6"
    ObjectType: "VM"
    Objects: "YS.Node1"
- CodeType: "SEQ"
  OpCode:
    Operation: "vscale"
    ObjectType: "REQ"
    Objects: "vs_req_1 || vs_req_2"
```

"XOSmall". However, the second operation is scaling up, where the "XOMedium" VM is scaled up to "XOLarge". The last element still specifies that these two operations are performed at the same time. Besides, this operation is only at VM level.

Moreover, we analyse and derive that the operation of vertical scaling can be achieved by using the basic Cloud VIFs as follows. No matter scaling up or down, the vertical scaling operation first requires the basic Cloud VIF of *VM Terminating* to terminate the target VMs which are going to be scaled. Then, the basic Cloud VIF of *VM Configuration* is leveraged to configure the network connections of all the VMs other than the target ones and detach them from the target VMs. Afterwards, the VMs with the new capacity are provisioned by the basic Cloud VIF of *VM Provisioning*. As is done, the target VMs and others should be configured to be connected with the original private IP addresses using the basic Cloud VIF of *VM Configuration*. Finally, the operation is done and the network topology remains the same to the application.

3.4.3 Failure Recovery

In the context of this thesis, the operation of failure recovery is leveraged when a particular data centre is down and not accessible. Hence, this operation is only at the sub-topology level.

Code 4 Example code for failure recovery

```
- CodeType: "SEQ"
  OpCode:
    Operation: "recover"
    Options:
    - ReqID: "rc_req_1"
      CP: "ExoGENI"
      DC: "GWU (Washington DC, USA)"
    ObjectType: "SubTopology"
    Objects: "XS"
- CodeType: "SEQ"
  OpCode:
    Operation: "recover"
    Options:
    - ReqID: "rc_req_2"
      CP: "ExoGENI"
      DC: "BBN/GPO (Boston, MA USA)"
    ObjectType: "SubTopology"
    Objects: "YS"
- CodeType: "SEQ"
  OpCode:
    Operation: "recover"
    ObjectType: "REQ"
    Objects: "rc_req_1 || rc_req_2"
```

Example Code 4 demonstrates the scenario that recover sub-topology "XS" from Washington data centre and "YS" from Boston data centre. In addition, these two operations are performed concurrently.

Moreover, we also analyse and derive that the operation of failure recovery can be achieved by using the basic Cloud VIFs as follows. According to the description, the failure recovery operation first requires the basic Cloud VIF of *VM Configuration* to configure the network connections of all the VMs excluding the failed ones for detaching them from the failed VMs. Then, the basic Cloud VIF of *VM Provisioning* is needed to provision the VMs from another specified Cloud or data centre. Finally, the basic Cloud VIF of *VM Configuration* is leveraged again to configure the network connections among the newly provisioned VMs and others, enabling the network topology to remain the same, even from different data centres.

3.5 CloudsStorm Framework Design and Overview

In previous sections, we have presented our infrastructure programmability design and the corresponding syntax of how to describe the infrastructure topology and operation. However, in order to actually execute the code and provision the desired infrastructure, we need an engine to interpret the corresponding code. Furthermore, a programming framework is essential to seamlessly mitigate the gap between the application and the infrastructure in the DevOps lifecycle. Therefore, in this section, we first propose the infrastructure programming framework, CloudsStorm, and introduce the architecture overview. Subsequently, we describe its components in detail and demonstrate the steps of using CloudsStorm through a specific example.

3.5.1 CloudsStorm Overview

Figure 3.5 illustrates the overview of the DevOps framework we propose. With this framework, Cloud application developers are not only able to develop their own applications but also able to program the virtual infrastructures. In the development phase, not only the application logic but also the infrastructure topology and operation can be programmed. In the runtime phase, the desired infrastructure is provisioned, and predefined infrastructure operations can be performed on the infrastructure to control the Cloud resources.

As shown in Figure 3.5, first in the development phase, the application developer exploits the infrastructure programmability provided by CloudsStorm to program the Cloud infrastructure according to the application requirements systematically. To be specific, the four types of infrastructure code introduced above in Section 3.3 are leveraged to program the infrastructure along with the application. The "*Infrastructure Description Code*" and "*Infrastructure Execution Code*" are two mandatory ones, as the descriptions of the infrastructure topology and operations are the fundamental approach to provisioning the the desired infrastructure from scratch. The other two types of infrastructure code are options more for adjusting the infrastructure to satisfy the quality-critical requirements of the applications.

Secondly, the programmed "*Infrastructure Execution Code*" is interpreted by the

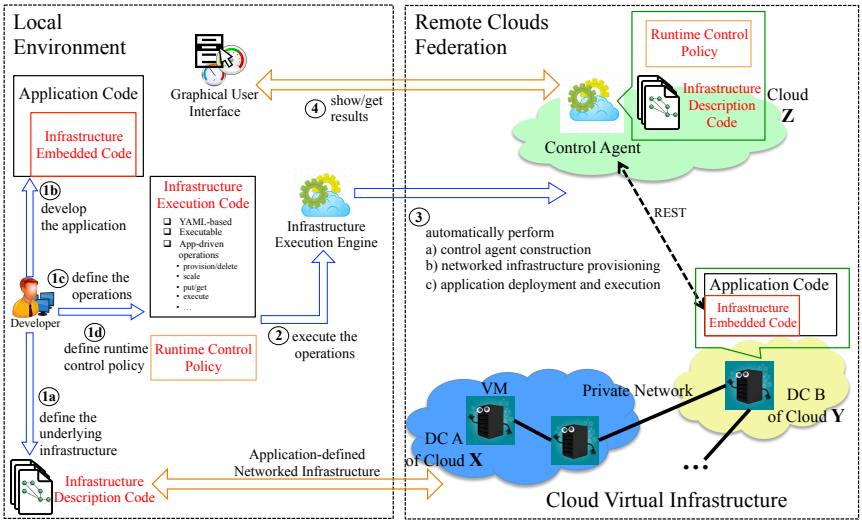


Figure 3.5: Overview of CloudsStorm framework

key engine of CloudsStorm, "*Infrastructure Execution Engine*". Still in the local environment of the developer, the engine would interpret the operations defined in the "*Infrastructure Execution Code*" to provision the corresponding infrastructure defined in the "*Infrastructure Description Code*".

Thirdly, with provisioning and executing a set of operations defined in the "*Infrastructure Execution Code*", it is entering the runtime phase of the application and the infrastructure. According to the detailed description in Section 3.3.2, there are two modes to execute the "*Infrastructure Execution Code*". For the short-term task-based applications, they do not need to keep the infrastructure for a long period. In this case, the results can be attained after finishing the execution of "*Infrastructure Execution Code*". This execution mode is demonstrated detailedly as the short-term scenario in Section 6.1 of Chapter 6. On the other hand, for the long-term task-based and service-based applications, the local machine of the developer cannot keep executing the infrastructure code for a long term or be leveraged as a server to control the infrastructure. In this case, the first operation performed by the "*Infrastructure Execution Engine*" is to provision a VM from a particular Cloud as the "*Control Agent*". Afterwards, all the programmed four types of infrastructure code are migrated to the "*Control Agent*". Via this, the "*Control Agent*" takes over the role to continue executing the "*Infrastructure Execution Code*" for provisioning the infrastructure, configuring the network, deploying and executing the application. This execution mode is demonstrated detailedly as the long-term scenario in Section 6.1 of Chapter 6.

Finally, during the runtime phase, the "*Infrastructure Embedded Code*" inside the application logic can be executed to actively adjust the infrastructure by making requests to the "*Control Agent*" using the provided REST APIs. Meanwhile, the infrastructure can also be passively controlled by the "*Control Agent*" according to the "*Rntime Control Policy*" and the monitoring information. Besides, the state of the VMs and

the connections defined in the infrastructure can be checked from the Graphical User Interface (GUI) provided by the "*Control Agent*". Especially, all the VMs can be accessed through a web terminal to show the results after executing the application. This is demonstrated by the case study in Section 6.2 and shown as Figure 6.7. The detailed syntax and usage can be checked from the online manual[10] of CloudsStorm.

3.5.2 Components Description

Based on the framework overview, we describe the detailed functionality of each component mentioned in CloudsStorm.

"*Infrastructure Description Code*"

It provides the design-level programmability of depicting the infrastructure topology. The syntax is demonstrated in Section 3.3.1. The features are as follows:

- It helps the application developer to describe how many computing resources are needed, including VM numbers, types, and specifically the customisation of Clouds and data centres.
- It adopts a partition-based approach to describe the infrastructure, including three levels of top-topology, sub-topology and VM, in order to clearly describe a federated Cloud environment.
- It allows the developer to define the network connectivity among the VMs of the infrastructure as a private network, which is essential to keep the underlying infrastructure transparent to the application.

"*Infrastructure Execution Code*"

It provides the infrastructure-level programmability of programming the infrastructure operations. The syntax is demonstrated in Section 3.3.2. The features are as follows:

- It includes the Cloud VIF of provisioning and terminating at different infrastructure levels, e.g., provisioning a particular sub-topology first or terminating it to avoid wasting resources.
- It provides the basic Cloud VIF of "*VM Configuration*", which enables the developer to program executing any command on a specific VM. This feature is essential for deploying and executing the application.
- It provides the programmability to define the operation of downloading or uploading files between the local machine and a specific VM.
- It provides the programmability of high-level infrastructure operations, such as horizontal/vertical scaling, and failure recovery.
- It adopts the symbol "||" to simplify the representation of parallel operations.

[10] https://CloudsStorm.github.io/

"*Infrastructure Embedded Code*"

It provides a more fine-grained programmability at the application level, embedding the infrastructure operation logic into the application logic. The example usage is demonstrated in Section 3.3.3. The features are as follows:

- It consists of a set of APIs written as libraries for a general-purpose programming language, such as Java, Python, and C. Thus, the developer can adopt the API in the same language used by the application code. We implement an example library to invoke the CloudsStorm REST APIs in Java[11].
- In essence, these APIs just make particular REST requests to adjust the infrastructure according to the APIs provided by the "*Control Agent*". The final infrastructure operation will be performed by the "*Control Agent*".
- In this case, infrastructure operations can be actively performed by the application at runtime.

"*Runtime Control Policy*"

It provides the programmability for specifying the non-functional requirements. The syntax is demonstrated in Section 3.3.4. The features are as follows:

- It allows developers to define the policy for how to scale or recover the infrastructure when the infrastructure resources perform insufficiently or even fail.
- It provides different infrastructure-level metrics, such as availability, CPU and memory utilisation, for the developer to determine the system status.
- It allows developers to customise their own application-level metrics as the identifications of how to adjust the infrastructure.

"*Infrastructure Execution Engine*"

It is the fundamental engine to interpret the four types of infrastructure code above and perform the actual programmed infrastructure operations. The implementation[1] details are described in Section 5.3.1 of Chapter 5. The features are as follows:

- It is developed using Java language.
- There are two ways to utilise this engine.
  - *Standalone.* It is already packaged and can be downloaded as a Java ARchive (JAR) file[12]. Then, it can be invoked through command lines.
  - *Library.* It can be leveraged as a Java library, and the corresponding engines can be extended to plug in for supporting another Cloud by the developer. "Control Agent" is an example to utilise the engine with this approach.
- The operations are performed in parallel using the multi-thread technique to interpret the parallel definition of the infrastructure operations.

[11] https://github.com/zh9314/CloudsStormREST
[12] https://github.com/CloudsStorm/Standalone/releases

"*Control Agent*"

It is responsible for collecting the monitoring information, interpreting the "*Runtime Control Policy*", and managing the infrastructure at runtime. The implementation[13] is based on the "*Infrastructure Execution Engine*". The features are as follows:

- When required, the "*Infrastructure Execution Engine*" first provisions the VM to deploy the "*Control Agent*". Afterwards, the "*Control Agent*" takes over the responsibility to manage the infrastructure.
- It provides REST APIs for the "*Infrastructure Embedded Code*" to invoke at runtime.
- It also collects the monitoring information and perform respective operations according to the "*Runtime Control Policy*".
- It utilises the "*Infrastructure Execution Engine*" as a library to perform the actual infrastructure operations.
- It provides a browser-based GUI to visualise the status of the infrastructure according to the managed "*Infrastructure Description Code*". Furthermore, the provisioned VMs can be directly accessed to check the result through the provided web terminal. An example is shown as Figure 6.7.

3.5.3 Example of Infrastructure Programming using CloudsStorm

To demonstrate the usage of CloudsStorm intuitively, we take the Hadoop application as an example to show the general process of using CloudsStorm framework. The detailed demonstration of this example is shown as a case study of service-based applications in Section 6.2 of Chapter 6. The example usage of CloudsStorm is as follows:

1. For the first step in the development phase, CloudsStorm provides four types of code for infrastructure programming, which are distinct from the original application code. The detailed syntax of these codes is defined in Section 3.3.

   (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 centre. 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 3.3.1.

   (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

[13] https://github.com/zh9314/CloudsStormCA

terminated immediately after the application finishes the processing. The key function of the application is shown as Listing 6.1 of Chapter 6.

(c) The developer programs the "*Infrastructure Execution Code*" to perform operations, such as provisioning, deployment, and application execution. In the example case, the developer programs operations: provisioning the two VMs, downloading the input data, and executing the processing application developed in the previous step. These operations are important to build 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 utilisation 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 code that has been developed. The detailed method to invoke with the command line can be checked from the online manual[10].

3. The "*Infrastructure Execution Engine*" first sets up a "*Control Agent*" from a particular data centre. Then the agent takes control of the infrastructure construction including network connections, application deployment, and execution. Via this manner, all the control operations are performed by the "*Control Agent*" to dynamically adjust the infrastructure to satisfy the requirements of the application. It is worth mentioning that this step is optional, if the runtime control is not required, e.g., for the task-based application of short-term scenario in Section 6.1. Without provisioning the "*Control Agent*", the Cloud resource usage can reduce.

4. The "*Control Agent*" provides a web-based GUI to show the current state of the infrastructure. In the example case, Figure 6.7 is a runtime snapshot showing that there are three scaled VMs in the Hadoop cluster to perform the processing task. Here, the number of scaled VMs is in line with the input data size. Besides, there are also web terminals provided for directly accessing each VM to check and get results.

3.6 Conclusion

In this chapter, we propose the framework CloudsStorm, which can be leveraged by application developers to program the IaaS Cloud virtual infrastructures. We conclude that there are three levels of programmability, and we design four types of infrastructure code dealing with functional and non-functional requirements.

Firstly, the design-level programmability, "*Infrastructure Description Code*", is proposed for developers to customise a suitable infrastructure and host their applications easily. This level of programmability is in the form of static infrastructure topology description. Secondly, the infrastructure-level programmability, "*Infrastructure Execution Code*", is further designed to depict the operations performed on the infrastructure

for control. We are inspired by the idea of functional programming and MapReduce framework in data processing to propose the infrastructure programming model based on the basic Cloud VIFs. CloudsStorm leverages the basic function commonly provided by public IaaS Clouds and constructs complex operations to achieve high-level controllability on the entire distributed infrastructure. The basic Cloud VIFs are modelled, and the feasibility of constructing high-level operations based on these basic functions is demonstrated. However, they are not sufficient for more fine-grained programming. We, therefore, propose the third level of programmability at the application level, "*Infrastructure Embedded Code*", based on general-purpose programming languages. This application-level programmability can be embedded in the application code to describe the infrastructure operations along with the application logic. The infrastructure can then be better adjusted according to the application requirements. Finally, to address the non-functional requirements of the application, "*Runtime Control Policy*" is proposed, which allows developers to define the quality-critical constraints of the applications through identifying specific monitoring metrics. The further operations reacting to the defined conditions can also be programmed. It is worth mentioning that we design most of the syntax based on the YAML format, which is human-readable and easy to learn.

In summary, CloudsStorm is an extensible and open framework to support infrastructure programming for different IaaS Clouds. The Cloud virtual infrastructure can be leveraged from federated Clouds and programmed without the vendor lock-in issue. Moreover, the partition-based infrastructure management and the simplified parallel expression allow developers to program the parallel operations effectively. Specifically, the multi-level programmability design provides developers with a more dimensional and systematic view to program the Cloud virtual infrastructure.

4

Distributed Cloud Infrastructure Provisioning and CloudsStorm Overlay Networks

the provisioning phase

In the previous chapter, we propose the infrastructure programmability design to describe the infrastructure topology and program the infrastructure operations in the development phase. Afterwards, in the provisioning phase, the key challenge is how to provision the customised Cloud virtual infrastructure effectively, especially dealing with the networked infrastructure across data centres and even Clouds. Taking the example of live streaming applications, the challenges for provisioning are that: 1) it is difficult to provision in a short time when the underlying infrastructure is large-scale; 2) the provisioned infrastructure should be resilient to the failures of data centres, i.e., the live streaming keeps functioning even failures happen in the infrastructure; and 3) the customised private network should also be provisioned to make the underlying infrastructure transparent to the application since the infrastructure for running live streaming applications is normally distributed to connect the cameras and audiences.

In this chapter, we analyse the requirements of the infrastructure provisioning and the network connections in the provisioning phase. Then, we introduce the related work of fast provisioning techniques and networked infrastructures. To tackle the issue, we first clarify the research context and problem. Then we propose two overlay network mechanisms to connect the distributed infrastructures, and therefore, the entire infrastructure can be partitioned and provisioned in parallel to increase efficiency. Finally, we evaluate the connectivity and provisioning overhead of our approaches.

This chapter is based on:

- **Zhou, H.**, Wang, J., Hu, Y., Su, J., Martin, P., de Laat, C., Zhao, Z., "Fast resource co-provisioning for time critical applications based on networked infrastructures", In *IEEE 9th International Conference on Cloud Computing* (CLOUD), pp. 802-805. IEEE, 2016.
- **Zhou, H.**, Hu, Y., Wang, J., Martin, P., de Laat, C., Zhao, Z., "Fast and dynamic resource provisioning for quality critical cloud applications", In *IEEE 19th International Symposium on Real-Time Distributed Computing* (ISORC), pp. 92-99. IEEE, 2016.

4.1 Cloud Infrastructure Provisioning

The provisioning phase is the key to making the Cloud DevOps different from the traditional DevOps, where the infrastructure is physical and no need to provision. In this case, the provisioning overhead is crucial to deploy and operate the quality-critical applications on Clouds, especially in the case of recovery from failures or scaling due to a burst of inputs. The more efficient of the provisioning operation is, the more possibility for the application to achieve the quality requirements. Meanwhile, network connectivity is essential for the application components to communicate running on a distributed infrastructure in the Cloud environment. First of all, we analyse the requirements of the provisioning and the network configuration as follows.

4.1.1 Provisioning and Network Requirements Analysis

Many quality-critical applications are migrating to Clouds to get better quality assurance [126]. During the migration, resource provisioning is the key step to provide virtual infrastructure for deploying applications. Meanwhile, provisioning is also an elementary step for the infrastructure adaption, including failure recovery and auto-scaling. Hence, a fast and dynamic provisioning mechanism to maintain the infrastructure is essential for satisfying the quality requirements of the applications. In fact, many quality-critical applications are required to be implemented on Clouds, such as disaster early warning systems [9, 133], video broadcasting [132]. In addition, most of these applications are network-centric, meaning that the network should be considered when designing the infrastructure for the application. Therefore, networked virtual infrastructures are important for optimising dynamic critical constraints of these applications. Besides, most public Cloud providers have limitations on the number of available virtual machines (VMs) for each customer, according to the data centre (region or domain) [79]. Hence, it is also essential to dynamically provision resources from multiple data centre or Clouds. We conclude that from the perspective of provisioning and network, the requirements on the infrastructure for migrating quality-critical applications are as follows.

- **Efficiency.** When dealing with a large scale of infrastructure, which can be a cluster of hundreds of VMs, an efficient mechanism to operate these VMs are needed. Especially at the provisioning phase, if the entire infrastructure comes from one data centre, the congestion to provision such amount of VMs are high and would cause significant overhead.
- **Resiliency.** Cloud computing can only provide a remote and uncertain environment, the availability of which is unpredictable. Therefore, the entire infrastructure should be resilient to Cloud failures of particular parts. It is obvious that putting the entire infrastructure in one data centre is highly risky to result in the situation that the service of the application totally crashes.
- **Transparency.** With the trend of Internet of Things (IoT) and big data processing applications, the infrastructure should also be distributed to get close to the devices and data. So it is essential to keep the network transparent to the application, even the infrastructure resources are distributed to different geolocations with different public IP addresses for access. It is helpful to get rid of the dependencies

among the application components because of the infrastructure distribution. Then the infrastructure provisioning is able to be independent to the application communication configuration.

4.1.2 State of the Art

Resource provisioning is a hot topic in Cloud research. There are existing studies done on fast provisioning to accelerate the process of VM startup. To the best of our knowledge, most innovations are developed on the side of Cloud providers. They mainly focus on particular aspects of the problem. FVD [105] modifies the image format to make the startup process shorter. However, the Cloud provider needs to modify the hypervisor accordingly, which is complicated to apply. SnowFlock [74] and Twinkle [135] adopt the method of directly forking from a running VM to get rid of the startup process. This method can even reduce the time of configuring the execution environment for the applications, because the new VM is cloned from the original one. However, it has its own constraints insomuch as that conflicts may arise when multiple processes within the same VM invoke VM forking concurrently. Moreover, it also requires special implementations from the Cloud providers' side to provide this method. Another way of fast provisioning is to accelerate the downloading process for images. Romain et al. [115] propose a Peer to Peer (P2P) method for image downloading. Later, Zhaoning et al. [123, 124] introduce two solutions, VMThunder and VMThunder+, both of which can provision hundreds of VMs in a remarkably short period. It mainly achieves further optimisation on previous P2P methods. All these methods are based on the Cloud provider, who has to adopt these methods explicitly before their benefits can be realised. They cannot be directly used by the customer.

As for provisioning networked infrastructures, previous related research mainly includes network embedding and inter-cloud architecture to satisfy the requirements. Network embedding is mapping the virtual network topology to the actual physical network infrastructure. Yufeng et al. [119] propose an efficient heuristic algorithm design for embedding virtual topologies within a Global Environment for Network Innovations (GENI) control framework. Finally, a server uses Virtual Local Area Network (VLAN) to connect each part. Hence, it also depends on the Cloud provider. Ting et al. [112] focus on the problem of how to decide which physical resources to allocate for a particular virtual network with multiple data centres. The solution is based on the view of the data centre. The purpose of network embedding is to provide an efficient virtual network topology that crosses multiple sites from the benefit of the provider. Inter-cloud or inter-domain provisioning is another research aspect. Nikolay et al. [49] investigate the inter-cloud architectures. They provide a taxonomy for the framework and describe how provisioned resources collaborate with each other. Nelson et al. [89] propose an Inter-cloud Resource Provisioning System (IRPS) to describe the resource semantically and to provision resources across Clouds. It defines a set of resource ontologies to work from, due to there being different resource descriptions and management policies for different Clouds. However, the target of their solutions is not for network-centric applications with quality-critical constraints. They only need to provision resources without considering the network. It remains a problem to address how provisioned resources should communicate with each other if networking

is required.

In conclusion, most of their innovations focus on the Cloud provides' side. Another problem is how to preserve the original network topology when dynamically provisioning infrastructure from different domains or Clouds. Therefore, it is a challenging problem to provision infrastructure transparently both to Cloud providers and customers, especially considering the networking.

4.2 Research Context and Problem Statement

In this section, we describe problem boundaries to specify the context of the this research. We use ExoGENI[1] Cloud to demonstrate that the provisioning problem really exists, and then we model the provisioning overhead.

4.2.1 Research Context

In this chapter, we mainly focus on quality-critical Cloud applications. Three typical examples [126] of these applications are: 1) a collaborative real-time business communication platform; 2) an elastic disaster early warning system; and 3) a Cloud studio for directing and broadcasting live events. There are three properties that can be derived from all these three applications. First, they all need high availability and responsiveness, especially recovering from sudden failures. Hence, the resources for these applications should be provisioned as soon as possible to satisfy the quality requirements. Second, most of them need a large number of VMs. For example, the elastic disaster early warning system requires a lot of VMs to collect data from sensors deployed in different regions. Therefore, it must be scalable, requiring the infrastructure to be dynamically provisioned. Finally, they are network-centric. For instance, the video broadcasting application should specify the path of data transmission and have particular constraints on the bandwidth of links among components. These applications require networked infrastructure to connect the VMs with each other.

As mentioned above, the network connection requirements drive us to consider Networked Infrastructure-as-a-Service (NIaaS), which was first proposed by GENI project [15]. NIaaS means that the topology of connections between the VMs can be predefined explicitly with private network connections. There is more emphasis put on networking than by conventional Infrastructure-as-a-Service (IaaS). Currently, most IaaS Clouds just provide VM nodes to customers without specifying how they are connected. At least, there is no network specification for the inter-datacentre and inter-cloud infrastructures. However, network-centric applications require a NIaaS platform to describe network links. Although we choose a new GENI testbed, ExoGENI [8], as our experimental Cloud, which is a widely distributed NIaaS Cloud computing platform geared towards experimentation and computational tasks [7], we only use its basic functions as a public IaaS Cloud. The networked part is what we are going to build based on it. This is consistent with our goal of dealing with federated Clouds, instead of specific NIaaS Clouds from the Cloud providers' view. For experimenting, we adopt the Infrastructure and Network Description Language (INDL) [45] to describe

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

the infrastructure information and submit the request to ExoGENI, but without using it for network description. It is worth mentioning that the infrastructure description in the request is named as one slice in this Chapter. In addition, this functionality has been integrated into the CloudsStorm framework to support ExoGENI Cloud. The INDL description submitted to ExoGENI is transferred from the programmed "*Infrastructure Description Code*". In this chapter, the **slice description** is corresponding to the topology description of other chapters, i.e., sub-topology or top-topology. The **domain** is corresponding to the data centre. The **node** in one slice refers to the VM.

4.2.2 Problem Statement

In this part, we demonstrate the existence of the resource provisioning problem. Ming et al. [79] have taken measurements of provisioning time on Amazon Elastic Compute Cloud (EC2). It takes from 3 to 30 minutes to get a 1GB compressed VM image to start up. The provisioning on Azure takes even more time than the provisioning on EC2. In addition, the overhead rises as the number of VMs increases on both platforms.

As for ExoGENI Cloud, the customer needs to use INDL to describe the infrastructure resources as slices. Every node in the slice description has its own VM type. The customer then submits the slice as a request to one domain, i.e., data centre, of ExoGENI to make the resources provisioned. When all nodes are active, i.e., they are accessible via SSH, the controller of the data centre informs the customer. Hence, we define the provisioning overhead as the time duration from the moment the customer submits the request to the moment getting a success notification, from the Cloud customers' view. Here, we omit the time cost on uploading the slice description and receiving the success message, because their size cannot be larger than a few hundred kilobytes and

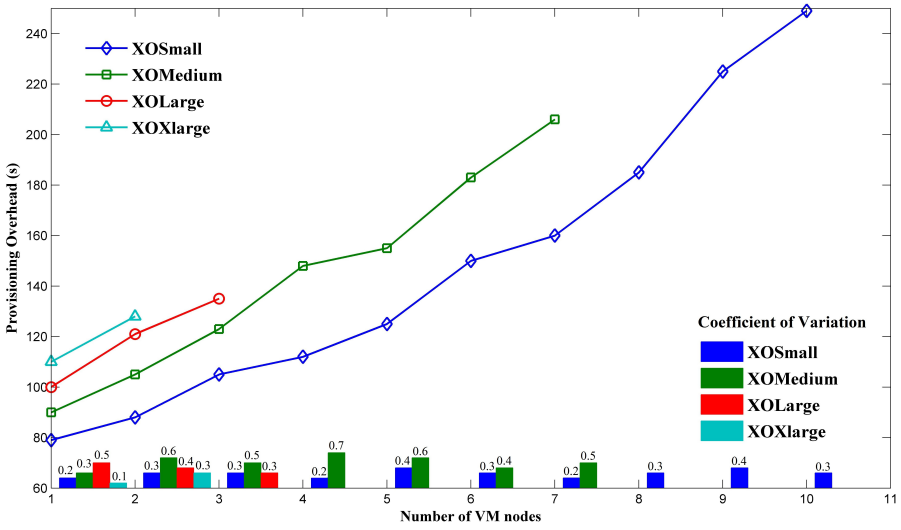


Figure 4.1: Measurements of provisioning overhead on ExoGENI

can be uploaded in milliseconds in our experimental environment. For the accuracy of our experiments, we submit slices to one domain and measure their provisioning overhead with different VM types[2] and numbers. For the consistency of comparison, the Operating System (OS) of all VM images in this chapter is set to "Ubuntu 14.04". The experimental results are shown in Figure 4.1.

The horizontal axis demonstrates the number of the VM nodes in one submitted slice. Different types of points in the figure mean different types of VM nodes created in one slice. The vertical axis demonstrates the average provisioning overhead of slices which are submitted to different domains. The lower portion of the figure denotes the coefficient of variation of this data set. If all the VMs in one slice cannot all be activated, the average provisioning overhead is not denoted here. For the slice only with "XOSmall" type of nodes, the maximum number of nodes in one slice that can be activated is 10. However, the maximum number for the type of "XOXlarge" is just 2. Actually, ExoGENI is not a commercial Cloud. Its capacity is relatively small, but it also demonstrates that the capacity of a particular domain, i.e., data centre, is limited. Besides, if the type of VM nodes is the same, the provisioning overhead depends on the number of nodes in the slice. A similar conclusion is drawn by Mao [79] based on measurements of public Clouds.

Based on the observed measurements, we propose a reasonable provisioning overhead model to approximately describe provisioning overhead. To simplify discussion, we only consider the situation in which all the nodes are the same type in each slice.

$$W_{total} = W_{single} + \eta \times W_{single} \times (n-1) \tag{4.1}$$

In Equation 4.1, W_{single} represents the provisioning overhead of a single node: n is the number of nodes in one slice, where $n > 0$, and W_{total} is the total provisioning overhead of the whole slice. In addition, η represents the delay between initialising the provisioning of one node and beginning the provisioning of the next, assuming a fixed provisioning pipeline. This delay may be caused by fetching the images of the nodes one by one and booting them sequentially. Thus, the more efficient this pipeline, the smaller η is. Previous fast provisioning methods work on the Cloud providers' side to reduce η. Thus, η can be treated as an optimisation degree of the Cloud platform. However, it cannot be zero, and therefore, the more VM nodes in one slice, the longer the slice needs to be provisioned. Mao et al. [79] draw a similar conclusion.

4.3 A Fast and Dynamic Provisioning Mechanism

Figure 4.1 shows that the more VM nodes in one slice, the more time the slice has to take to be provisioned. The basic idea comes from our observations and the overhead model, i.e., if we cut the slice into multiple smaller slices and dynamically provision them from different data centres or domains in parallel, the provisioning overhead of every slice can be reduced, as described in Equation 4.1. In other words, the idea is to partition the original topology description into different sub-topologies and provision from different data centres simultaneously. It is crucial, however, not to influence the customer's

[2] https://wiki.exogeni.net/doku.php?id=public:experimenters:resource_types:start

original network connection design, which is usually defined as a private network. It is, therefore, necessary to provide private network connections over the public network, i.e., the Internet. In order to tackle these issues, our mechanism addresses three aspects of the problem. These aspects are described in detail in the following three subsections. It is worth mentioning that the mechanism is described based on ExoGENI Cloud, but the mechanism is not limited to that Cloud.

4.3.1 Connectivity of Network

In this part, we describe two overlay network mechanisms to settle the problem of connectivity among partitioned slices. These network connection mechanisms are crucial for provisioning infrastructure and keeping the original network topology.

First, we need to understand the principles of VM internally connections and how a VM inside one ExoGENI data centre connects to the outside public network. Figure 4.2 illustrates how ExoGENI manages its IP addresses to form connections internally and externally, substantiated by real testing. In one data centre, each VM node has at least one virtual Ethernet interface, called $eth0$. If the VM node connects to other nodes, it may have other virtual Ethernet interfaces, such as $eth1$ and $eth2$, depending on the links established with the node. In the same data centre, most Clouds allow the customer to customise the link, e.g., VM_1 and VM_2 are connected with rIP_1 and rIP_2. Here, rIP is a fixed private IP address, which can be predefined. For external connections, it is realised through the management Ethernet interface, $eth0$. It is assigned a private IP address when provisioned by the controller of the data centre. The $eth0$ of each node is then bound to a public address. The source address of outward packets is replaced by this public address. It is just the opposite for inward packets. The VM is then able to communicate with the Internet using the corresponding public IP address, uIP.

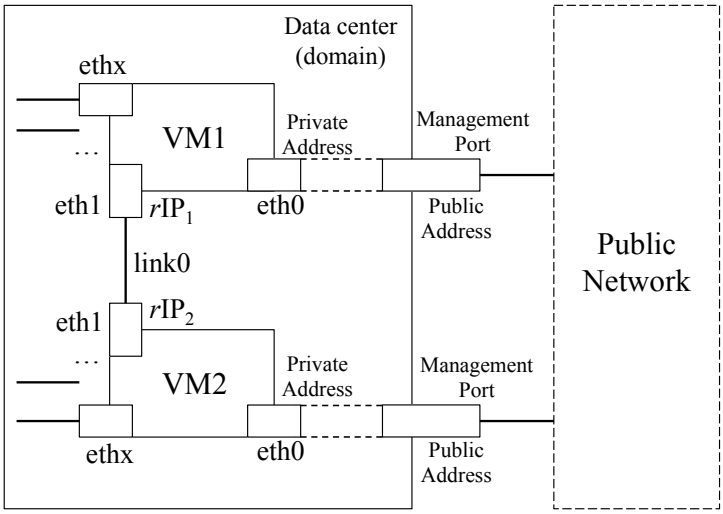


Figure 4.2: The illustration of how the VMs inside one ExoGENI data centre connect internally and externally to the outside public network

Besides, Figure 4.2 shows a typical topology which is suitable for our mechanism. VM_1 and VM_2 are two nodes in one slice. They are connected to each other with private IP addresses rIP_1 and rIP_2. These addresses are assigned by the customer. It means that the customer needs to deploy some components on VM_1 and VM_2. They communicate with each other using the specific private IP addresses rIP_1 and rIP_2. They may also connect to other parts of the topology. In this scenario, considering as the entire top-topology, this slice of two VMs can be divided into two sub-topologies by the link between VM_1 and VM_2. However, the customer, who operate applications on this infrastructure, should not be aware of this partitioning. From the application perspective, these two nodes should still be connected via this private network link, even though they need to be provisioned from different data centres. Hence, our solution is to put a proxy node in each partitioned sub-topology, as shown in Figure 4.3. The whole slice is divided into sub-topology A and sub-topology B, which are going to be provisioned from data centre A and B, respectively. Here, VM_a and VM_b are proxy nodes in each sub-topology. The proxy node can be configured with any minimal VM image that supports "iptables" for using Network Address Translation (NAT) technique. This technique can change the source and destination addresses of the packets. Especially, it provides options to change the address before routing or after routing. Therefore, the packet flow can be controlled with proper configurations.

Figure 4.3 illustrates how one packet travels through the public network between two sub-topologies. Here, VM_a in sub-topology A works as a mirror of VM_2 in sub-topology B. VM_b is similar. Hence, the interface of VM_a which is connected to VM_1 is configured with IP address rIP_2. Consequently, packets sent to VM_2 are forwarded to VM_a instead. Before routing of VM_a, the destination address is changed from rIP_2 to uIP_b, where uIP_b is the public address of VM_b in sub-topology B. VM_a forwards the packet from $eth0$ when routing. Otherwise, the packet is processed by VM_a. It is also essential to change the source address rIP_1 to rIP_a, which is the private address assigned to $eth0$. The packet can then be sent to the public network with a proper public IP address uIP_a. The source address is changed from rIP_a to uIP_a. The address translation here is done by the ExoGENI data centre itself. Hence, the packet can be normally transferred on the public network with two proper public IP addresses. Sub-topology B then receives the packets from the Ethernet with public IP address uIP_b. Packets are automatically transferred to the $eth0$ of VM_b node by ExoGENI, as the private address rIP_b is bound to that public address uIP_b. The configuration on VM_b is similar. Before routing, the destination address is changed from rIP_b to the original

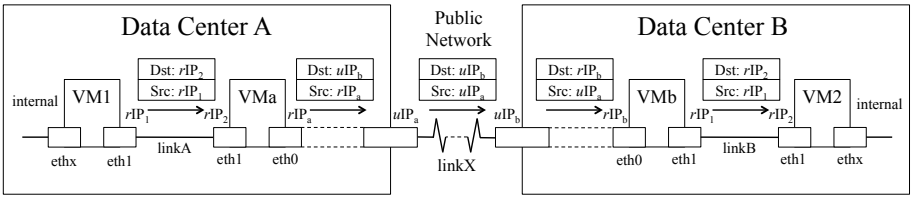


Figure 4.3: Connection mechanism among partitioned sub-topologies through NAT technique

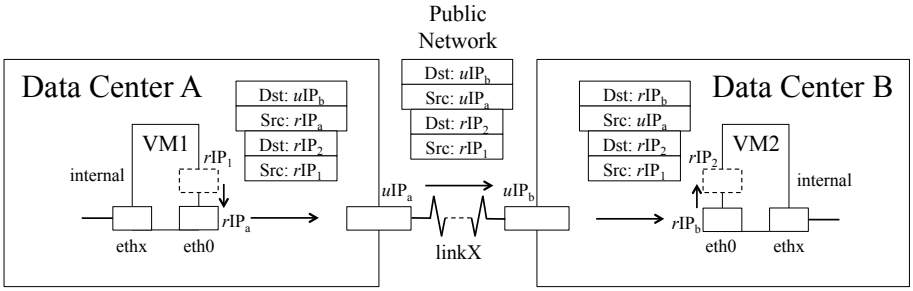


Figure 4.4: Connection mechanism among partitioned sub-topologies through tunnelling technique

private address rIP_2. After routing, the source address is changed to rIP_1. Hence, the IP packet is converted back to its original state, as sent from VM_1. From the view of the application running in VM_2, it receives the packet directly from VM_1 through the customised private network with addresses of rIP_1 and rIP_2. In this manner, the private network packets are able to be forwarded via a public network.

Moreover, we propose another overlay network mechanism to connect these partitioned sub-topologies using the tunnelling technique. This mechanism is shown in Figure 4.4. Each VM in different data centres should be first configured to add a virtual Ethernet interface with the customised private IP addresses. Then with the IP tunnelling technique, the original packet with the private network addresses given by the application running in the VM can be wrapped in another packet. Via this packet, the original packet can be delivered through the public network transparently to the customer. As shown in Figure 4.4, from the view of the application running in VM_2, it receives the packet, of which the source is the originally defined private IP address rIP_1, after stripping the header of the outside packet. Therefore, this mechanism can also achieve the VMs in different data centres communicating with the predefined private addresses.

Comparing these two mechanisms, they both allow the customer/developer to customise the network of the infrastructure with predefined IP addresses. As shown in both Figure 4.3 and 4.4, rIP_1 and rIP_2 are private IP addresses set by the customer/developer to connect VM_1 and VM_2. However, when these two VMs are not in the same data centre, they cannot directly communicate using these private IP addresses. On the other hand, rIP_a, rIP_b, uIP_a and uIP_b are assigned by the data centre and cannot be easily customised. These addresses change every time the infrastructure is provisioned. However, with above two connection mechanisms, these two VMs can always communicate with rIP_1 and rIP_2, no matter they are in the same data centre or not. Therefore, the networked infrastructure is transparent to applications.

The advantage of the tunnel-based mechanism is without bringing in the extra overhead of establishing proxy VM nodes. Hence, it gives more flexibility to partition the slice while still adhering to the quality requirements. Nonetheless, if the original VMs, e.g., VM_1 and VM_2 in Figure 4.4, are customised to use the OS of "Windows", the tunnel-based mechanism cannot directly work because only some versions of "Linux" support IP tunnelling by default. On the contrary, there is no this kind of

limitations for NAT-based mechanism. For example, the OS of the proxy VMs, VM_a and VM_b in Figure 4.3, can always be customised as "Ubuntu" which is supported by all Clouds, because the proxy VMs do not need to host applications. In this case, the NAT-based mechanism is easy to configure using the configuration of Listing 4.1. Another disadvantage of the tunnel-based mechanism is that we need to make reconfiguration over the original VMs, where the application is hosted. This configuration makes the infrastructure not at the same level of transparency to the application as that of the NAT-based mechanism. We, therefore, discuss how to partition the infrastructure to accelerate the provisioning process, according to the NAT-based mechanism in the following sections.

4.3.2 Virtual Infrastructure Partitioning Algorithm

In this part, we describe how to partition the original slice requested by the customer. Basically, the original slice, i.e., top-topology, can be treated as a graph with nodes, i.e., VMs, and links, i.e., network connections. Although we investigate specific graph partition algorithms [10, 20] and tools [52, 76], they are not applicable in our context. As mentioned above, the basic idea is to partition the entire infrastructure and provision them in parallel from different data centres to reduce the overhead. However, with the NAT-based connection mechanism, every link we cut adds another proxy node for each partitioned slice, i.e., sub-topology. Therefore, we propose an infrastructure partitioning algorithm that minimises the number of extra nodes required. To be specific, we first consider partitioning the entire infrastructure into two equal parts.

First of all, we define the weight of the VM nodes in one slice. As mentioned in Section 4.2.2, the provisioning overhead mainly depends on the number of VMs. The weight of the VM node denotes the provisioning overhead. It is determined by the type of VM and its OS image [79]. For standardisation, we define the weight of the node with the type and image which has least provisioning overhead as unit one. The weight of other VM nodes can be defined as multiples of unit one based on measurements. In our experiment, the weight of the node with "XOSmall" and "Ubuntu 14.04" is unit one. In addition, Figure 4.1 shows that the provisioning overhead of the slice has an approximately linear relationship with the number of the nodes. Therefore, we define the weight of one slice as being the weight summation of all the nodes in that slice.

Before partitioning, we abstract the link information, $LinkCollections[m][i]$, from the infrastructure slice description, which is a two-dimensional array input for Algorithm 1. Each element in the array contains a set of links, $LinkCollections[m][i].links$, by which the slice can be totally divided into two parts. The row number plus one indicates that this row stores all the link collections which can divide the slice by that amount of links. For example, each element in the first row of the array, i.e., $LinkCollections[0][i].links$, always contains one link, by which the slice can be totally divided. Each element in the second row, i.e., $LinkCollections[1][i].links$, always contains two links, by which the slice can be totally divided. Here, the link collection, $link$, is represented as a set of sequence numbers to denote links. Besides the property of the link collection, each element in the array has another two properties. One is $LinkCollections[m][i].leftweight$, i.e., the weight of one part of the slice. This part is one of the divided sub-slices, i.e., sub-topologies, after the original slice is

partitioned by this collection of links. Second is $LinkCollections[m][i].rightweight$, i.e., the weight of the other sub-slice.

Algorithm 1 then presents the partitioning algorithm to select the best two sub-slices to minimise the provisioning overhead. There are two inputs: the weight WA of the original slice A and the array of $LinkCollections$ described above. m in Algorithm 1 denotes how many links we are considering to cut across when dividing the slice. We start with considering dividing the slice across a single link ($m = 1$). After dividing the slice across the links in the element $LinkCollections[m-1][i]$, the actual weight of the partitioned part increases a little, because if you partition the slice across just one link, every partitioned part has to add one more proxy node for communication. Partitioning the slice from two links brings two more proxy nodes in each part, and so on. For reducing overhead, the type of the proxy node is set to "XOSmall", whose weight is defined as unit one. Therefore, the actual weight of each partitioned part increases by m. The bigger weight of these two partitioned sub-slices, $WeightBC$, determines the actual overhead of simultaneously provisioning these two sub-slices. If the minimum of $WeightBC$ under the condition of the same m is smaller than WA, then the slice can be partitioned in this way. Otherwise, we continue to try dividing the slice across more links. Ideally, the best way to partition the original slice is to divide it into two equal parts, whose weight is $WA/2$. If m reaches $WA/2$, the weight of each partitioned sub-slice will be at least equal to the weight of the original slice.

Algorithm 1 Infrastructure partitioning algorithm

Input:

WA, the weight of the original Slice, i.e., Top-topology, A with weight WA;
$LinkCollections[m][i]$, an array of link sets, which partition the original slice.

Output:

{Sub-slice, i.e., Sub-topology, B and C} or $\varnothing$, partitioned infrastructure.

```
1:  m ← 1
2:  while m < WA/2 do
3:      minWeight ← −1
4:      minLink ← 0
5:      for i = 0 to LinkCollections[m − 1].length do
6:          rW ← LinkCollections[m − 1][i].rightweight + m
7:          lW ← LinkCollections[m − 1][i].leftweight + m
8:          WeightBC ← max(rW, lW)
9:          if minWeight == −1 or WeightBC < minWeight then
10:             minWeight ← WeightBC
11:             minLink ← i
12:     if minWeight ≠ −1 and minWeight < WA then
13:         {B, C} ← LinkCollections[m − 1][minLink].links
14:         return {B and C}
15:     else
16:         m ← m + 1
17: return ∅
```

In this case, the partitioned topology cannot reduce the provisioning overhead. Thus, the algorithm stops at this point and returns an empty set meaning the slice cannot be divided. Moreover, it is worth mentioning that this algorithm can be leveraged to further partition the sub-slice, if the original slice requires to be provisioned faster or to be divided into more than two parts.

4.3.3 Multi-thread Provisioning

With the connection mechanism and partitioning algorithm, we can partition the original topology of the customer's application into linked sub-topologies. Basically, there are three steps to take before submitting the final topology of the customer's request to a Cloud provider: 1) create a userkey for authentication; 2) instantiate the client provided by the specific Cloud provider's SDK; and 3) call the remote method of this client to submit the request. This process is actually realised in the CloudsStorm framework as specific engines for ExoGENI demonstrated in Section 5.3.1 of Chapter 5.

Most Clouds commonly use this process for customers to request the provisioning of virtual resources, demonstrated as the basic Cloud Virtual Infrastructure Function (VIF) of provisioning in Section 3.2.2. However, if we submit the sub-slices, i.e., sub-topologies one by one, through this function, the total provisioning overhead actually increases because of the extra proxy nodes. In this part, we adopt a multi-thread technique to provision the sub-topologies of the topology at the same time. The main process is as shown in Algorithm 2. This type of parallel operations is also implemented in CloudsStorm described in Chapter 5.

Every thread finally returns the information about the state of the created sub-topology, which is discussed in Section 5.3.2. If it succeeds, then the returned information contains the state of all the VM nodes in the requested slice, especially including the public addresses, which are key to the later configuration of proxy nodes. This provisioning process is mainly modelled based on that of ExoGENI Cloud. Nevertheless, the primary part of this provisioning technique is similar for other Cloud providers.

Algorithm 2 Multi-thread provisioning process

Input:

n, the number of the partitioned sub-slices, i.e., sub-topologies, need provisioning;
$SliceDes[n]$, the infrastructure description of the nth sub-slice.

Output:

$\{IP_{pub}\}$, the set of all the successfully provisioned VM's public IP.

```
1: userkey ← getUserKeyFile(keyPath)
2: xxClient ← createClient(userkey)
3: for i = 0 to n do
4:     Info[i] ← new Thread(xxClient, SliceDes[i], CREATE_SLICE)
5: for i = 0 to n do
6:     if Info[i].status == "succeed" then
7:         {IP_pub} ← Info[i].{IP_pub}
8: return {IP_pub}
```

4.4 Implementation and Evaluation

In this section, we discuss some implementation details of the previously proposed mechanism. Then we take ExoGENI as a testbed and conduct experiments using the connection mechanism to evaluate the performance. The evaluation mainly focuses on the connectivity among multiple partitioned sub-topologies and the efficiency of the provisioning technique. Especially, because of the special features of ExoGENI, we can test this mechanism with inner domain and inter-domain scenarios. Inner domain refers to the scenario that sub-topologies are in the same domain, i.e., data centre. Inter-domain refers to the scenario that sub-topologies are located in different domains.

4.4.1 Prototype Process

Figure 4.5 shows the detailed process of the mechanism we propose. $Top\text{-}topology$ describes the customer's original request for infrastructure as a single slice. First, we partition this original slice into different sub-slices, i.e., sub-topologies, with our partition algorithm if possible. Then we modify every sub-slice to add proxy nodes for connections. $Sub\text{-}topology_i$, therefore, becomes $Sub\text{-}topology_i'$. With the modified slices, we adopt the multi-thread technique to submit each sub-slice in its own thread

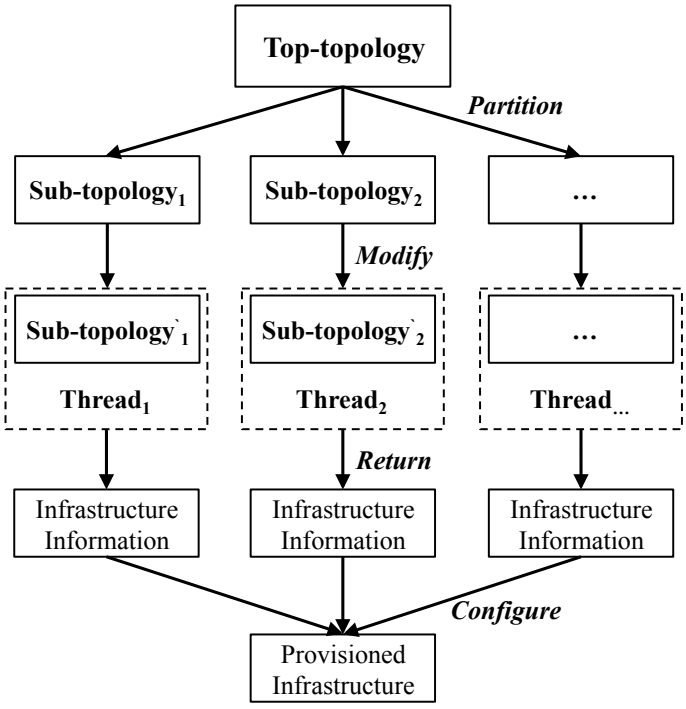


Figure 4.5: The prototype process of topology partitioning, infrastructure provisioning, and network configuration.

simultaneously. If each sub-slice is activated successfully, it should return information about the infrastructure. The key information is the public addresses of the proxy nodes. With these addresses, we can configure the network through executing shell scripts remotely on the proxy nodes. The shell scripts configure these proxy nodes to translate IP packets' addresses correctly when crossing slices. This mechanism allows VMs in different sub-slices to communicate with each other. Finally, the entire original infrastructure is provisioned, and the whole process is transparent to customers and providers. From the view of the customers' side, the application can always get the infrastructure as it requires. From the view of the providers' side, they do not need to consider how to accelerate the provisioning process further.

In the following, we detail the configuration of the shell script. For the NAT-based mechanism, we take the configuration on VM_a in Figure 4.3 as an example. Here, VM_a is customised to be with the "Ubuntu" OS. The configuration details are shown in Listing 4.1. Line 4 of the script is to get the default Ethernet interface name of the VM, which is not always 'eth0'. Then the private address rIP_a can be retrieved from the default interface. As mentioned above, rIP_1 and rIP_2 are determined by the customer. uIP_a and uIP_b are obtained from the returned information after the infrastructure are provisioned successfully. Therefore, all the IP addresses in the script are available. Line 6 to 9 of the script are to configure the VM for address translation. Here, "iptables" is a common tool provided by "Linux". To use this tool is also why we choose "Ubuntu 14.04" as the OS of the proxy node in experiments. Moreover, Line 7 specifies only to translate the destination address to rIP_1, i.e., sending to VM_1, if the source address of the packet is uIP_b, i.e., the packet comes from VM_b, because there are also other packets sending to VM_a for access and control. The destination address of these packets should not be changed. Otherwise, the proxy node cannot be accessed anymore.

```
1 #!/bin/sh
2 echo 1 > /proc/sys/net/ipv4/ip_forward
3 sysctl -p
4 ethName=`ip r show| grep "default "| cut -d " " -f 5`
5 rIPa=`ifconfig $ethName| grep "inet addr"| awk -F'[ :]' '{print $13}'`
6 iptables -t nat -A PREROUTING -d $rIP2 -j DNAT --to-destination $uIPb
7 iptables -t nat -A PREROUTING -s $uIPb -j DNAT --to-destination $rIP1
8 iptables -t nat -A POSTROUTING -d $uIPb -j SNAT --to-source $rIPa
9 iptables -t nat -A POSTROUTING -d $rIP1 -j SNAT --to-source $rIP2
```

Listing 4.1: Example script for configuring the NAT-based overlay network

For the tunnel-based mechanism, we take the configuration on VM_1 in Figure 4.4 as an example. In this example, VM_1 requires to be customised as "Ubuntu" OS, which is also the limitation of this mechanism. But in CloudsStorm framework, we implement different "V-Engine" corresponding with various operating systems to tackle this issue. The configuration details are shown in Listing 4.2. Still, all the variables in this listing is known after provisioning. In the listing, '$rIP1' and '$rIP2' are application-defined private addresses, corresponding to rIP_1 and rIP_2 in Figure 4.4. '$uIPa' and '$uIPb' are the corresponding public addresses after provisioning. '$linkX' in Line 5 and 6 can be defined by the customer. '$subnet' and '$netmaskStr' in Line 6 are the subnet and netmask of the application-defined private network, e.g., "192.168.10.0" and "255.255.255.0", which are also provided by the customer.

```
#!/bin/sh
ethName=`ip r show| grep "default "| cut -d " " -f 5`
rIPa=`ifconfig $ethName| grep "inet addr"| awk -F'[ :]' '{print $13}'`
ip tunnel add $linkX mode ipip remote $uIPb local $rIPa
ifconfig $linkX $rIP1 netmask $netmaskStr
route del -net $subnet netmask $netmaskStr dev $linkX
route add -host $rIP1 dev $linkX
```

Listing 4.2: Example script for configuring the tunnel-based overlay network

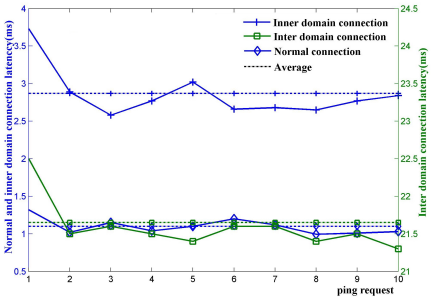
4.4.2 Evaluation of Connectivity

In this part, we design an experiment to evaluate the network performance between partitioned slices. Here, we only focus on the NAT-based mechanism, as its connection performance cannot be better than that of tunnel-based mechanism. It is because the NAT-based mechanism brings in extra links and latency. Thus, we only evaluate the NAT-based overlay network as a baseline. The basic topology in our experiment is shown in Figure 4.3. First of all, we need to demonstrate that node VM_1 and VM_2 are really connected using our technique. It is essential to prove that the packets sent by VM_1 are forwarded to VM_2 instead of just being processed by VM_a. We adopt the "ping" and "tcpdump" tools, provided by Linux. We run "tcpdump" on VM_2 to collect all the Internet Control Message Protocol (ICMP) packets heading to VM_2. Then we "ping" VM_2 from VM_1. The results show that VM_2 indeed captures the ICMP packets from VM_1. Therefore, our technique of connection is demonstrated to work properly.

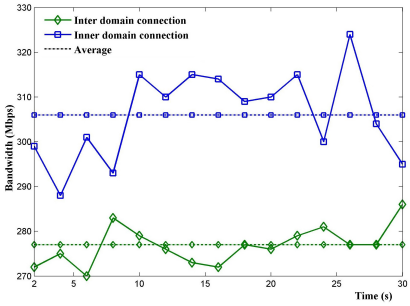
For the network performance, we use the "ping" tool to evaluate the latency and the "iperf" tool to measure the bandwidth. Figure 4.6 shows the evaluation results on network performance. As for ExoGENI, it affords the option to choose the domain of the slice, i.e., the data centre. Hence, we evaluate the performance from three aspects. One is the evaluation of normal connections in the original slice. One is the evaluation of communication between inner domain slices, and the other is of communication between inter-domain slices. In one scenario, we make slice A and slice B both within the Houston data centre. For the inter-domain scenario, we make slice A within the Washington data centre and slice B within the Chicago data centre. We also create a slice located in Houston to evaluate the original connection.

Figure 4.6(a) illustrates the evaluation of network latency. The average latency of a normal connection in one domain is 1.10 ms. The average latency of an inner domain connection is 2.87 ms. It is a little higher because of the NAT overhead between proxy nodes. The average latency of an inter-domain connection is 21.65 ms, which is much higher. It is caused by transmitting the packets over a long physical distance. However, even the latency of 21.65 ms is already fast enough for most quality-critical applications. On the other hand, we can choose to provision the infrastructure with inner domain connections, if the latency cannot be tolerated. Moreover, the latency volatility of all these scenarios appears to be small, demonstrating that our connection technique is stable. Therefore, it should satisfy most latency requirements.

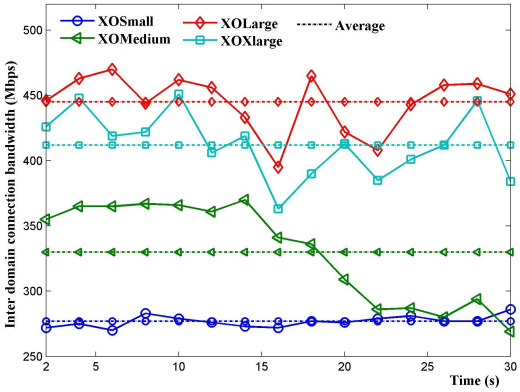
Figure 4.6(b) illustrates the evaluation of network bandwidth using "iperf". It measures the bandwidth between VM_1 and VM_2 every 2 seconds. We do not measure the bandwidth of a normal connection, because the bandwidth of the link in one slice



(a) Comparison of latency



(b) Comparison of bandwidth



(c) Bandwidth comparison on different types of proxy node

Figure 4.6: Evaluation of application-defined network connection performance

can be set within ExoGENI. We need to measure the bandwidth of the link a public network, however. In our scenarios, the bandwidth of $linkA$ and $linkB$ in Figure 4.3 should be set as large as possible as they should not be the bottleneck. In our experiment, they are set as 1 Gbps. The result shows that the bandwidth of inner and inter-domain connections are close at around 300 Mbps. The large bandwidth of inter-domain connections may be owed to the connections in place between different domains for use by ExoGENI. Anyhow, the bandwidth of 300 Mbps is sufficient, and our mechanism does not have a strong impact on the performance. The connection technique, therefore, can satisfy the bandwidth requirement of most applications.

We adopt the type of "XOSmall" as the proxy node to reduce the provisioning overhead in our design. However, we also evaluate the bandwidth with different types of proxy nodes in order to evaluate their impact. The result is shown in Figure 4.6(c). This evaluation is just in the scenario of inter-domain connections, because Figure 4.6(b) has already shown the similarity between inner domain and inter-domain connections in practice. Figure 4.6(c) shows that the bandwidth varies with the type of proxy nodes. It demonstrates that a more powerful proxy node can improve the connection bandwidth.

However, the proxy node with the type of "XOXlarge" cannot further improve the bandwidth. It is probably because the network has reached its physical limit. Therefore, there is a trade-off between connection bandwidth and provisioning overhead.

4.4.3 Evaluation of Fast Provisioning

In this section, we evaluate the overhead of our provisioning technique. We design the experiment using nodes with the type of "XOMedium" as an example. There are three scenarios: normal provisioning, inner domain and inter-domain provisioning. Among them, the inner domain and inter-domain provisioning are based on our mechanism. We assume that the customer requests a topology containing $2n$ nodes. According to the theoretic analysis in Section 4.2.2, cutting the slice just into two equal parts can reduce the provisioning overhead most. Therefore, for inner domain provisioning, we provision two slices with equal numbers of nodes within the Boston data centre. Each slice contains n nodes. As for inter-domain provisioning, we provision the same two slices to the data centres of Boston and Washington. Correspondingly, for normal provisioning, we measure the overhead of provisioning whole infrastructure in one slice from Boston. In order to compare with our provisioning mechanism, we measure two kinds of normal provisioned slices. One is to compare the overhead with each partitioned slices. It contains n nodes. The other is to compare the overhead with the whole topology. Then it contains $2n$ nodes. Here, we ignore the overhead of a proxy node, because it adopts the VM with the smallest capacity. The overhead can be omitted, especially when provisioning large-scale infrastructures. Meanwhile, there is no proxy node to increase the overhead when we adopt the tunnel-based overlay network for connecting the partitioned infrastructures.

As for the accuracy of results, we measure the provisioning overhead of the same slices three times for every scenario. In addition, we just measure provisioning overhead of up to 6 nodes due to the limitations of ExoGENI as an experimental Cloud. The result is shown in Figure 4.7.

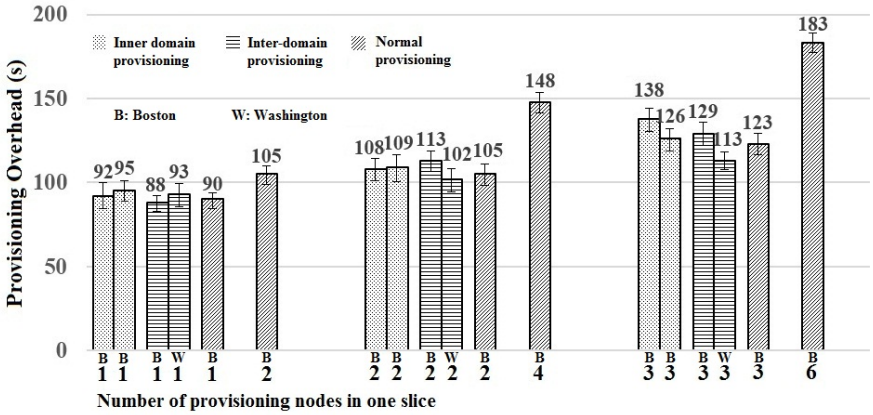


Figure 4.7: Evaluation of fast provisioning.

The experimental result demonstrates that the overhead of our provisioning technique is close to that of normal provisioning with half the number of nodes. We can derive that our provisioning technique is more efficient as the number of nodes increases. Although the total number of nodes is almost the same for normal provisioning and our provisioning mechanism, it is faster to provision them across multiple slices with our provisioning technique. It is obvious for the scenario of inter-domain provisioning, i.e., multiple smaller slices are provisioned from different data centres concurrently. However, it is worth mentioning that the overhead of inner domain provisioning is close to inter-domain provisioning. Although partitioned slices are provisioned within the same location, they may be provisioned on different racks. Hence, the experiment demonstrates that our provisioning technique indeed reduces the provisioning overhead.

On the other hand, we can evaluate our mechanism in theory with the provisioning overhead model, because we cannot test large-scale infrastructures on ExoGENI. Only considering cutting the slice into two parts, the most overhead we can save is as shown in Equation 4.2.

$$\Delta W_{save} = W_{total}(n) - W_{total}(\frac{n}{2}) = n \times \frac{\eta}{2} \times W_{single} \tag{4.2}$$

Equation 4.2 shows that when n is big enough, $\Delta W_{save} \approx W_{total}(n)/2$. It means that the provisioning overhead can be approximately reduced by half for large-scale infrastructure. In this case, the extra overhead of proxy nodes can be omitted for NAT-based mechanism. For the tunnel-based mechanism, there is no extra overhead for partitioning. And also, we do not consider the situation in which the slice can be divided into more than two parts. As described in Section 4.2.2, η represents the optimisation degree of the platform, and $\eta/2$ is shown in Equation 4.2. Hence, it is equivalent to doubling the provisioning performance of the platform from the Cloud providers' viewpoint. Considering the concrete result, we use the method of linear least squares fit to get the detailed equation for ExoGENI. According to our experiment, we choose the dataset of "XOMedium" to fit, which contains the 7 points in Figure 4.1. Then we achieve Equation 4.3.

$$W_{total} = 19.1n + 67.7 = 86.8 + 0.22 \times 86.8 \times (n - 1) \tag{4.3}$$

Compared to Equation 4.1, $W_{single} = 86.8$ and $\eta = 0.22$. Then, $\Delta W_{save} = 9.55n$. Therefore, when $n = 100$, i.e., for the infrastructure of 100 VMs, we can save about 15 minutes with our provisioning mechanism. This is a remarkable optimisation for quality-critical applications.

Finally, we do not analyse the overhead of the infrastructure partitioning algorithm. In fact, the complexity of this algorithm is high, especially when preparing the input of the two-dimensional array, $LinkCollections$, for link information. However, the result of the algorithm is reusable, when migrating the application or recovering from failures. We also do not need to calculate the algorithm again when re-provisioning a particular part of the infrastructure. Therefore, we do not think that the overhead of the partitioning process is a critical concern.

4.5 Conclusion

This chapter presents a fast and dynamic provisioning mechanism to accelerate the process of provisioning for quality-critical Cloud applications. It mainly contains three key steps: the infrastructure partitioning algorithm, network connectivity configuration, and multi-thread provisioning. With the partitioning algorithm, we can divide the original request for networked infrastructure into multiple smaller infrastructures. Then we propose two overlay network mechanisms, i.e., the NAT-based mechanism and the tunnel-based mechanism, to ensure the connectivity among sub-infrastructures. Finally, multiple infrastructures are dynamically provisioned with multiple threads.

By experimental practice and theoretical analysis, this mechanism confers three advantages. Firstly, this approach is fast and dynamic. The multiple smaller infrastructures can be provisioned with less overhead. Overhead is decided by the most significant single part, not their sum if multi-thread provisioning is applied. Meanwhile, if some part of the infrastructure crashes, we just need to re-provision the crashed part, not the whole infrastructure. It is essential for the quality-critical application recovering from failures. Secondly, there is a capacity improvement. Cloud providers often impose limitations on the scale of infrastructures. Our mechanism puts forward a way to provision large-scale infrastructure though using resources from multiple Clouds and data centres. Thirdly, our mechanism is not only transparent to Cloud providers but also to the applications operated by customers. The providers just need to afford an interface to provision resources, which is one of the basic Cloud VIFs. From the customers' viewpoint, they get the infrastructure as they design it, including IP addresses, making the application agnostic to the underlying infrastructure and without being aware of the modification. Therefore, there are no additional constraints or extra configurations required for customers to run applications on new Clouds or data centres, i.e., even provisioning the infrastructure from other Clouds or data centres, the network topology can remain the same. In order to demonstrate the feasibility of the mechanism, we tested it on ExoGENI. The experimental results in practice and model analysis in theory both show that our mechanism can potentially dramatically reduce the provisioning overhead, especially for large-scale infrastructure. The reduced provisioning overhead is crucial for the application to scale out or recover with particular quality-critical constraints.

It is worth mentioning that the fast provisioning mechanism mentioned in this chapter has already been integrated into the CloudsStorm framework. The overlay network mechanism is leveraged to connect the VMs of different sub-topologies with the application-defined network topology, no matter which data centre the sub-topology is hosted. Moreover, all the sub-topologies in CloudsStorm are provisioned in parallel by default, using the multi-thread provisioning technique in this chapter. The detailed implementation is introduced in the next chapter.

5

Seamless Cloud Infrastructure Runtime Control and CloudsStorm Framework Implementation

the runtime phase

In previous chapters, we discuss the infrastructure programmability in the development phase and the fast provisioning technique for the networked infrastructure in the provisioning phase. Still, in the runtime phase, the infrastructure should be controlled to keep satisfying the quality-critical requirements of the application, due to unexpected events, such as bursty workloads and even failures. Taking the Hadoop based big data processing application as an example, the controllability for managing the infrastructure at runtime requires that: 1) the infrastructure needs to be adjusted after a particular event happens based on the monitoring feedbacks, e.g., the processing performance decreases if certain datanodes from one data centre are not accessible, which then requires provisioning datanodes from other data centres for recovery; and 2) the infrastructure needs to be directly controlled and adapted for a specific event based on prior knowledge, e.g., the Hadoop application can measure the workload input and adjust the infrastructure to a proper capacity through scaling to suit the workload before processing.

In this chapter, after analysing the controllability requirements, we propose a control model with two modes and describe the detailed implementation of CloudsStorm framework which empowers the Cloud application with the controllability. Finally, the experimental study conducted on real Clouds, EC2 and ExoGENI, demonstrates that the controllability of CloudsStorm is efficient and outperforms other related tools.

This chapter is based on:

- **Zhou, H.**, Hu, Y., Su, J., de Laat, C., Zhao, Z., "CloudsStorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures", In *International Conference on Cloud Computing*, pp. 265-280. Springer, Cham, 2018.
- **Zhou, H.**, Hu, Y., Ouyang, X., Su, J., Koulouzis, S., de Laat, C., Zhao, Z., "CloudsStorm: A Framework for Seamlessly Programming and Controlling Virtual Infrastructure Functions during the DevOps Lifecycle of Cloud Applications", Journal of *Software: Practice and Experience*. Wiley, 2019.

5.1 Cloud Infrastructure Control

For the traditional software Development and Operations (DevOps), there is no ability empowered for the infrastructure to adjust the capacity to suit the application at runtime, as the physical infrastructure is fixed. On the contrary, the elasticity of the Cloud virtual infrastructure makes it possible to control the infrastructure in the runtime phase dynamically. Specifically, since the pay-as-you-go business model of Clouds, a seamless infrastructure control can reduce the monetary cost and satisfy the quality requirements of the application at the same time. Therefore, we analyse what the requirements are for controlling the infrastructure to react to the applications quickly and seamlessly.

5.1.1 Controllability Requirements Analysis

Traditionally, the system is adjusted because the influences of a particular event are detected. It is an essential manner for Cloud infrastructure control. However, this feedback based control is not in time, and therefore, cannot seamlessly satisfy the applications' requirements. In this section, we analyse the controllability requirements according to the example of orchestrating Hadoop applications on Clouds and conclude two types of control modes are crucial.

1. **Passive mode.** The infrastructure should be passively controlled when meeting certain predefined thresholds after a particular event happens. The thresholds can either include the infrastructure-level metrics, such as availability, CPU and memory utilisation, or the application-level metrics, e.g., the system's throughput. For instance, once one of the virtual machines (VMs) is crashed for hosting the Hadoop applications, another VM should be recovered from a particular predefined data centre and rejoin the cluster to keep satisfying the application's quality requirements. It is the same issue to scale the infrastructure according to the real-time monitoring of CPU utilisation or the data processing rate.
2. **Active mode.** In order to satisfy the application quality requirements more seamlessly and smoothly, applications also need to actively control their infrastructure before the influences of a particular event. For instance, the Hadoop application requires to scale out more VMs to be the datanodes when processing a large amount of data as inputs. Meanwhile, considering the movement of the data source, the computing resources should also be dynamically controlled to be close to the data source for reducing transmission cost. This type of control is ever urgent in the emerging Fog/Edge computing domain, where the applications even need to control their Cloud resources close to the edge nodes on demand according to the dynamic distribution of sensors [88].

5.1.2 State of the Art

For the passive control mode analysed above, the feedback based control is the de facto manner to manage Cloud systems [103]. Mahmoud et al. [2] propose a dynamic resources provisioning and monitoring system to manage the Cloud provider's resources while taking into account the customers' quality requirements. However, the view of this

work is to tackle the issue from the providers' side of monitoring the data centre states and performing the resource provisioning, which is opposite to the view of this thesis. Shicong et al. [81] introduce the concept of Monitoring-as-a-Service. They propose three approaches to analyse the monitoring information to enhance accuracy and also improve the efficiency of the monitoring service itself. Jcatascopia [108] is implemented as a tool to exploit probes to monitor the Cloud infrastructure. Nevertheless, neither of these works touches the ability to enable the infrastructure to be adapted based on the monitoring information. Cloud4sens [39] is proposed for monitoring the sensors in the Internet of Things (IoT) environment. The Cloud here is leveraged as the infrastructure to deploy the monitoring service, instead of the infrastructure to be managed by the monitoring service. From the view of the customer side to manage the infrastructure, CloudWatch[1] provides the similar functionality of the passive control mode, but it is a vendor lock-in solution only for Amazon Elastic Compute Cloud (EC2).

On the other hand, there is no systematic manner to sustain the active control mode. Alexey et al. [58] focus on the scaling policy evaluation but without considering the functionality to support for scaling operations on the Cloud infrastructure. Tools, such as Libcloud[2] and jclouds[3], are just Application Programming Interfaces (APIs) for programmatically control the infrastructure resources, which is, however, not sufficient to seamlessly combine it with the application logic.

5.2 Cloud Infrastructure Runtime Management

To realise above two controlling modes, we first describe the execution model of how to interpret the "*Infrastructure Description Code*" and "*Infrastructure Execution Code*". Then, we propose the runtime control model with two types of controlling modes, i.e., passive and active mode, to specifically demonstrate how the "*Infrastructure Embedded Code*" and "*Runtime Control Policy*" are leveraged.

5.2.1 Execution Model

Figure 5.1 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 3.5.3. After customising the infrastructure as step 1 in Section 3.5.3, 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 centres involved in running the application. 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 centres' information for these Clouds, which are required to automatically

[1] https://docs.aws.amazon.com/cloudwatch
[2] http://libcloud.apache.org/
[3] https://jclouds.apache.org/

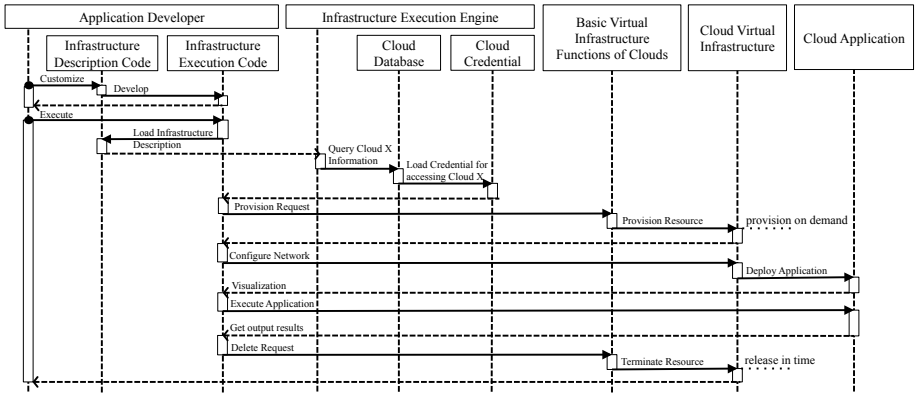


Figure 5.1: Sequence diagram for infrastructure code execution model

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 5.3.3. After loading the credential information, the "*Infrastructure Execution Engine*" is able to invoke the basic Cloud Virtual Infrastructure Function (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 customised private network connection to construct the "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 business model of Clouds, the longer occupying the Cloud resources, the more need to pay.

5.2.2 Runtime Control Model

This subsection gives a detailed description of step 3 in Section 3.5.3. 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

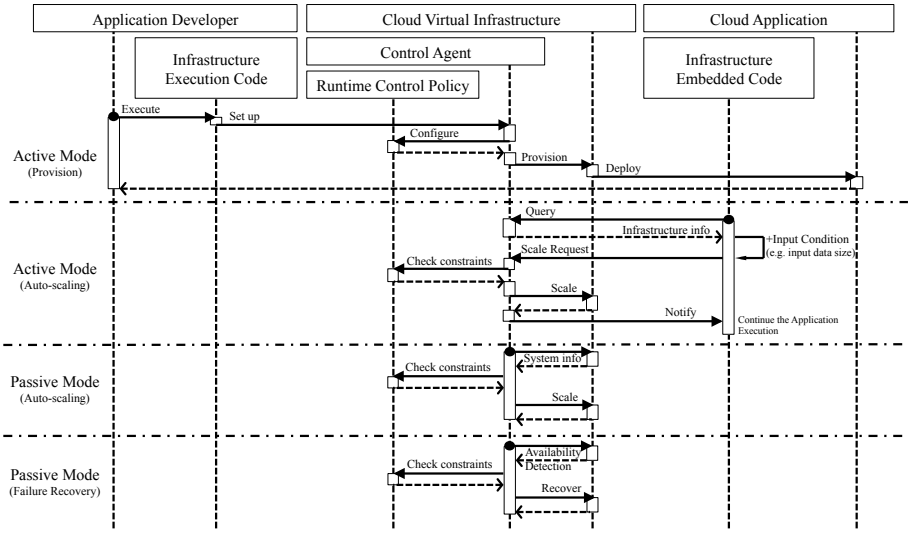


Figure 5.2: Sequence diagram for runtime control model with two modes in different scenarios

in a separate VM of "Cloud Virtual Infrastructure", and other VMs are informed about the public IP of the "*Control Agent*" for communication.

Figure 5.2 illustrates the sequence diagram for the runtime control model. It consists of two controlling modes, the active and passive mode. 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*" firstly 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 REpresentational State Transfer (REST) APIs to adjust its underlying infrastructure according to outside input conditions, e.g., the input data size. The "*Control Agent*" performs the actual operations on the Cloud infrastructure after checking constraints of the "*Runtime Control Policy*", e.g., whether the budget is enough. Therefore, the application is able to actively customise the infrastructure to fit its requirements. This is also demonstrated in the case study of Section 6.2 in Chapter 6.

The other is the passive mode. Figure 5.2 illustrates two scenarios when using passive control mode, which is auto-scaling and failure recovery. The operations performed in passive mode are dependent on the "*Runtime Control Policy*" and monitoring information. In the scenario of auto-scaling, the "*Control Agent*" analyses the performance information collected from VMs of the infrastructure. If the CPU or memory usage of certain VMs meets 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 centre 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, i.e., from which backup Cloud and data centre. 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 centre is down or the network to the data centre is not accessible. It is worth mentioning that once the operations are performed by the "*Control Agent*" onto the "Cloud Virtual Infrastructure", the "*Infrastructure Description Code*" managed by the "*Control Agent*" must be updated to keep synchronised with the actual status of the virtual infrastructure, e.g., whether the VM is terminated or not.

5.3 CloudsStorm Framework Implementation

In this section, we present the CloudsStorm controllability implementation in following aspects: the detailed infrastructure execution engine implementation for achieving parallel operations and extensibility of federated Clouds; the infrastructure status transfer for managing infrastructure descriptions among operations; the logging component and relevant supporting libraries for managing Cloud information and access credentials.

5.3.1 Infrastructure Execution Engine and Control Agent

The "*Infrastructure Execution Engine*" is implemented based on Java programming language and is the elementary engine to complete the execution procedure demonstrated in Figure 5.1. 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 illustrated in Figure 5.3.

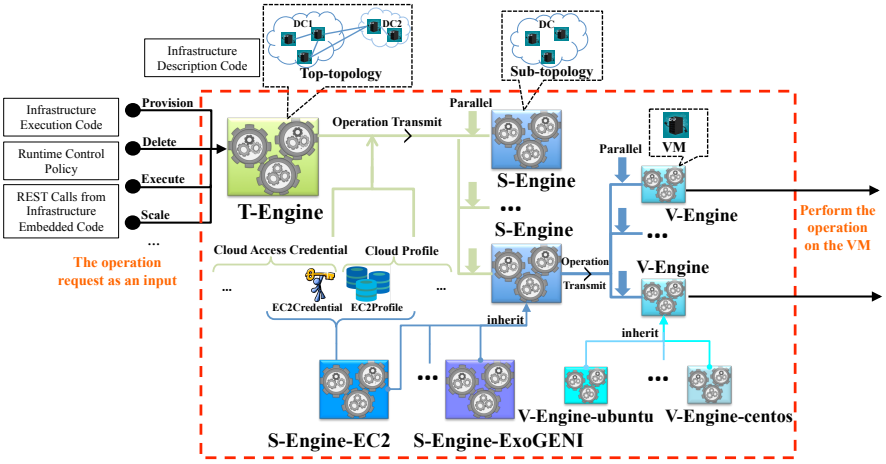


Figure 5.3: Implementation details of "*Infrastructure Execution Engine*"

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 5.2.1, 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 Cloud access credential is provided by the application developer for authentication and billing. The detailed information for organising the Cloud profiles and access credentials is discussed in Section 5.3.3, which also can be provided by CloudsStorm as libraries. The V-Engine is responsible for operations on each individual VM of the 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 overlay network on each VM to provision the networked infrastructure required by the application. Currently, we implement overlay network mechanism based on the tunnelling technique to connect the VMs from federated Clouds as a private network. The details for network configuration are described in Chapter 4. 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 customised V-Engines can be inherited from it depending on the features of the VM, for example, "V-Engine-ubuntu" for an Ubuntu VM. If the application has specific operations on some VM, it can customise 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 realised according to the factory design pattern in software engineering. Moreover, all the S-Engines and V-Engines use the multi-thread technique 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 centres, there will be no conflict among them, and they can run totally in parallel. This parallelism is the same for V-Engines: the operations on all the VMs in one sub-topology can proceed simultaneously. This mechanism is also leveraged to realise the parallel symbol '||' for multiple "Objects" defined in Section 3.3.2. In other words, "*Infrastructure Execution Engine*" can accelerate operations to reduce the total time needed by an application utilising the Cloud, reducing the application's total cost as paid to Cloud providers.

From the implementation perspective, Figure 5.4 shows the class diagram of "*Infrastructure Execution Engine*", which detailedly draws the relationship among the "T-Engine", "S-Engine", "V-Engine", and specific inheriting engines. In this class diagram, we mainly show the Cloud VIF of provisioning. So the only method of most engines is illustrated as "provision", and other methods are omitted.

At both levels of "S-Engine" and "V-Engine", the factory design pattern in software engineering is leveraged to customise an engine from the basic class according to a specific Cloud and Operating System (OS). Moreover, a Java reflection mechanism

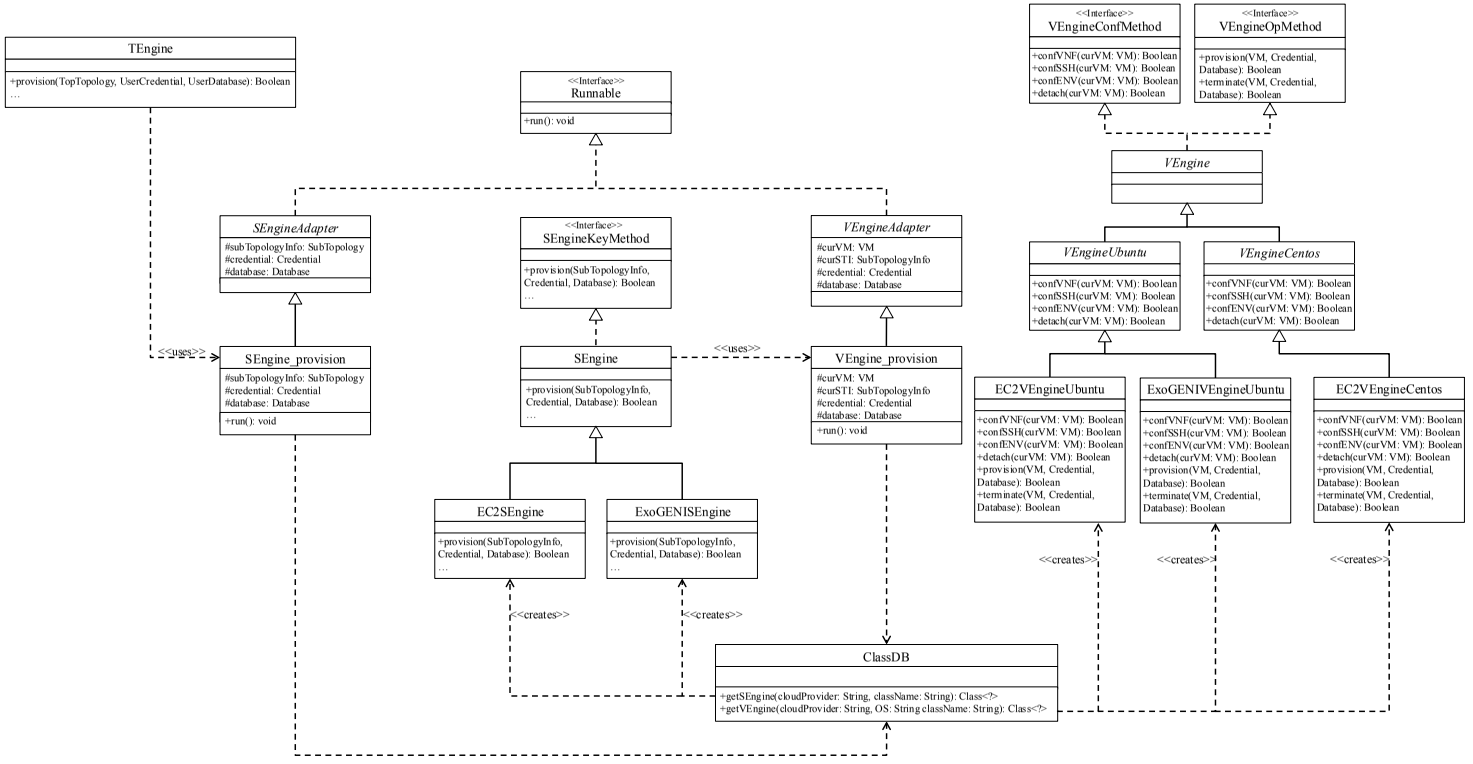


Figure 5.4: Class diagram of the "*Infrastructure Execution Engine*" implementation

is exploited to load some user-defined engine class dynamically. This mechanism is realised through the class of "ClassDB" in the figure. It provides the method of "getSEngine" and "getVEngine" to provide the corresponding class according to the Cloud provider name and OS type. Hence, this design is extensible to support different Clouds. Even the engine for some Cloud is not implemented by the framework, the developer still can program an entirely new engine for supporting a particular Cloud.

The adapter design pattern is adopted for upper-level engine controlling different types of engines at the lower level in parallel. For example, in the "provision" method of "SEngine", it leverages the class "VEngine_provision" to transfer the provisioning operation from the sub-topology level to the VM level. The class "VEngein_provision" finally invokes different "V-Engines" to perform the operation. Moreover, the adapter class "VEngineAdapter" empowers the Class "VEngine_provision" with the ability to run in a thread. Hence, this implementation mechanism realises the parallel operation, which is defined by the symbol '||' in the previous programmability design.

It is also worth mentioning that we separate the core methods of "V-Engine" as two types of interfaces, "VEngineConfMethod" and "VEngineOpMethod". They are corresponding to the basic Cloud VIFs defined in Section 3.2.2. The basic Cloud VIFs, i.e., *VM Provisioning* and *VM Terminating*, align with the methods of "provision" and "terminate" defined in the interface "VEngineOpMethod". These functions are Cloud-specific. On the other hand, the methods defined in the interface "VEngineConfMethod" all belong to the basic Cloud VIF of *VM Configuration* but are classified into more specific operations, which are OS-specific. This separation design allows the developer to only implement that two methods, "provision" and "terminate", for a new Cloud. The OS part functions can be handled by the framework, as the number of different OS types is limited, but the Cloud types are relatively more.

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 basic Cloud VIFs of a new Cloud, i.e., provisioning and terminating one VM. CloudsStorm then is able to plug in this basic VIF and afford high-level programmability and controllability of this new Cloud for the application developer. Currently, CloudsStorm has already implemented engines to support three Cloud providers, EC2, ExoGENI, and EGI[4] (European Grid Infrastructure).

Besides, even the "*Control Agent*" is key for the application to control the infrastructure during runtime, the "*Infrastructure Execution Engine*" is still the main component of the "*Control Agent*". The other component of the "*Control Agent*" provides web services, which contain a set of REST APIs to receive the infrastructure operation requests and then invoke the "*Infrastructure Execution Engine*" to perform. They also provide a web-based Graphical User Interface (GUI) to show the status of the infrastructure. The detailed implementation of the "*Control Agent*" is also open source[5].

5.3.2 Infrastructure Status Management and Transfer

In order to control infrastructures, CloudsStorm defines five statuses for the sub-topology according to the infrastructure description of sub-topology shown as Syntax 2 in Sec-

[4]https://www.egi.eu/

[5]https://github.com/zh9314/CloudsStormCA

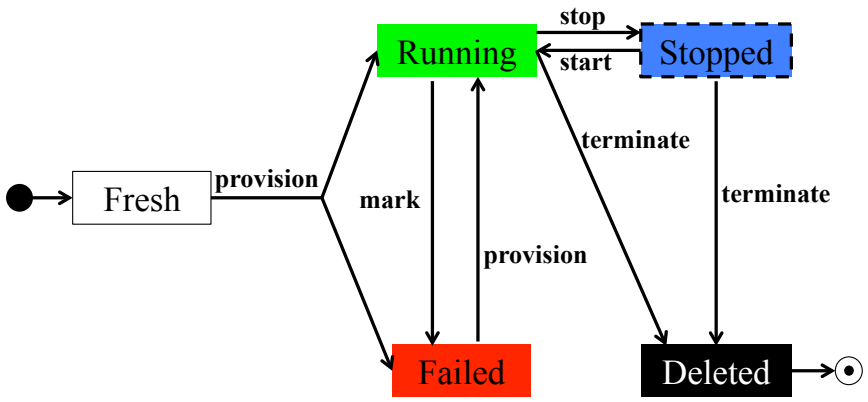


Figure 5.5: Status transition of infrastructure lifecycle in CloudsStorm

tion 3.3.1 of Chapter 3. Figure 5.5 is the status transition graph. It begins with the "Fresh" status, which means the infrastructure is in the design phase and the public IPs are not assigned. When the "T-Engine" controls the "S-Engine" to do provisioning, the status of the sub-topology infrastructure can transit into two statuses, "Running" or "Failed", depending on whether there are errors during provisioning. If becoming "Running", the public IPs of VMs in the running sub-topology must have been assigned. Afterwards, if the "Control Agent" detects some running sub-topology is not accessible or failed, the T-Engine marks its status as "Failed" and controls the corresponding "S-Engine" to identify the failed sub-topology. Meanwhile, all the sub-topologies which originally are connected with this failed one should be detached. The failed sub-topology then can be recovered within another data centre according to the recovery requests. "Stopped" status is circled with the dashed line, as some Clouds do not provide the function of stopping a VM. If the Cloud does not support this function, there is no "Stopped" status in its lifecycle. The reason for designing the status "Stopped" in the lifecycle is that the stopped VM is faster to bring up again than provisioning new one from "Fresh" when scaling up. Finally, "Deleted" is the terminated status of the lifecycle after applying the terminating function. All the "Stopped" and "Running" infrastructures can be terminated to release resources.

5.3.3 Relevant Components of 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 centres for selected Clouds. This information includes: 1) geographic positioning of each data centre, which can be leveraged to do locality-aware or data-aware provisioning; 2) endpoint information, which describes a URL of a data centre controller needed for actual provisioning; 3) VM types (CPU, memory) supported in each data centre and their characteristic data (e.g., price). This information is not application-defined but can be provided by CloudsStorm. 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, "accessKey" and "secretKey", are required for accessing EC2 Cloud. Hence, the credentials are given and managed by the application developers 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 organised in the YAML format.

Finally, a logging component is built in CloudsStorm. The log file is also organised in the 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[6].

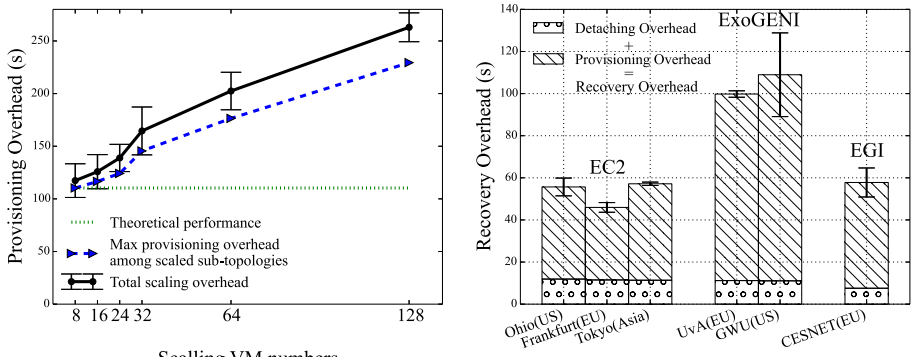
5.4 Controllability Performance Evaluation

In this section, we first evaluate the controllability performance of CloudsStorm, including auto-scaling and failure recovery.

5.4.1 Auto-scaling and Failure Recovery

Auto-scaling and failure recovery are the key controllability of the infrastructure provided by CloudsStorm. We firstly design an experiment on ExoGENI to test the auto-scaling performance. In this experiment, there are initially two sub-topologies, $subNI_1$ containing 1 VM and $subNI_2$ containing 8 "XOMedium" VMs. Each VM in $subNI_2$ is connected with the VM in $subNI_1$ via a private network link. This topology is suitable for 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 centres 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 are based on 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 5.6(a) illustrates that we scale out the 8 VMs of $subNI_2$ accordingly 1, 2, 3, 4, 8 and 16 times of scale. Each scaled $subNI_2$ is provisioned from independent data centres simultaneously. The time for provisioning each scaled sub-topology 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 centre has the same provisioning performance and there is no interference in between. It means that the time for scaling different sub-topologies is the same and a constant value t, i.e., $T_i = T_j = t$, for $\forall i, j \in \mathbb{N}$ and $1 \leq i, j \leq 16$. It is then obvious to derive that $\max_{1 \leq i \leq 16} T_i = t$. Thus,

[6]https://CloudsStorm.github.io/



(a) Horizontal scaling out performance

(b) Failure recovery performance

Figure 5.6: Controllability performance evaluation of CloudsStorm

no matter how many VMs need to be provisioned, as long as all the sub-topologies (each sub-topology contains 8 VMs) are in different data centres, the entire provisioning overhead should remain the same. However, the provisioning performances of different data centres 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 auto-scaling. In addition, most Clouds have limitations on resource allocation. For instance, ExoGENI usually allows one user to apply a maximum of 10 VMs from one data centre. The limitation for EC2 is 20. Nevertheless, with CloudsStorm, we can break through these limits to realise large-scale scaling by combining resources from different data centres and even Clouds.

Figure 5.6(b) shows the experimental result of failure recovery. In this experiment, there are still two sub-topologies in the beginning, $subNI_1$ and $subNI_2$. Each of them contains only one VM, V_1 and V_2. These two nodes are connected with a private network. We then assume the case where the data centre of $subNI_2$ is not available. CloudsStorm recovers the same sub-topology from another data centre 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 V_1 of $subNI_1$ to V_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 6 data centres from them. In order to compare, V_2 always has 2 cores and around 8G 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 centres are not stable. The performance of EC2 and EGI are close; however, most data centres of EC2 are more stable. This kind of information has reference value when deciding where to recover, to satisfy the application quality requirements, considering the recovery overhead and data centre geographic information.

5.4.2 Comparison with Related Tools

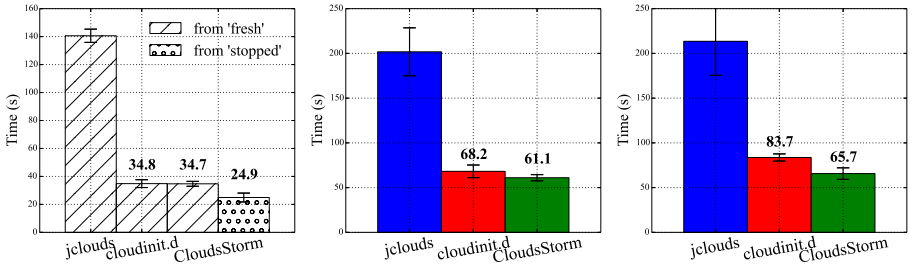
In this section, we compare CloudsStorm framework with the related tools from the performance and functionality perspective, respectively.

Performance Comparison

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 tool, such as CloudPick [27]. From the set of environment-centric tools, we pick Nimbus [66] team's cloudinit.d[7]. Other tools, e.g., Juju and IM (Infrastructure Manager), provide graphical interfaces, which make it difficult to measure performance. Both of jclouds and cloudinti.d do not support networked infrastructure. The ones who support networked infrastructure can only be applied in private data centres, where CloudsStorm cannot have the access permission, e.g., SAVI (System Architecture Virtual Integration) [63]. 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 the California data centre of EC2. However, as jclouds and cloudinit.d cannot directly support auto-scaling behaviour, we use them to provision 5 new VMs in California data centre for the assumption of this scenario. Each operation, we repeated 10 times. Figure 5.7(a) 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 Syntax 2 of Section 3.3.1) 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 auto-scaling 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 5.7(b)(c).

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 choose the California data centre to provision 5 "t2.micro" VMs and install Apache Tomcat on each of them. Each test is repeated 10 times. Figure 5.7(b) shows the results. With jclouds, the applications are installed one by one, which costs plenty of time. For

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



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

Figure 5.7: Controllability performance comparison of CloudsStorm and related tools

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, 4 out of 5 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 realise the dependency. In this scenario, the first level is the MySQL VM, and the second level contains four Tomcat VMs. The difference for CloudsStorm is that it can provision networked infrastructure. The nodes are connected with application-defined private network links. The MySQL server address is pre-defined before actual provisioning. Therefore, all the deployments can proceed simultaneously even with the dependency. Figure 5.7(c) demonstrates that the deployment dependency has smaller influences on the performance of CloudsStorm comparing to that on jclouds and cloudinit.d. We can reason out that if there are more dependencies than the current case, CloudsStorm would have a more significant advantage over others.

Functionality Comparison

Finally, we pick several tools from related ones. Table 5.1 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”, “Auto-scaling”, and “Failure Recovery” are the controllability features. “Multi-Mode” indicates whether both of the controlling modes, i.e., the active mode and the passive mode, are supported,

Table 5.1: 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-lisation
jclouds	Java	×	-	×	√	√	√	×	×	×	×
Libcloud	Python	×	-	×	√	√	√	×	×	×	×
Puppet/Chef	Ruby	-	√	×	√	√	√	-	-	×	-
Ansible	YAML	×	√	×	√	×	√	-	-	×	-
SAVI	Not Known	×	√	√	×	√	×	-	-	×	×
Juju	YAML	-	√	×	√	√	√	-	-	×	×
cloudinit.d	Python	×	√	×	√	√	√	-	-	×	×
CodeCloud	CJDL	-	√	×	√	√	√	√	×	×	×
CloudPick	GUI	√	√	×	√	√	√	-	√	×	×
CometCloud	Java	√	√	×	√	×	×	×	×	×	-
mOSAIC	Java	√	√	×	√	×	√	√	√	×	×
CloudsStorm	YAML	√	√	√	√	√	√	√	√	√	√

1. "×" indicates this functionality is not supported.
2. "√" indicates this functionality is supported.
3. "−" indicates this functionality is partially supported. For instance,
   a) "−" in "Infrastructure Description" refers the tool only provides GUI for infrastructure description instead of languages;
   b) "−" in "Auto-scaling" and "Failure Recovery" refer the tool still requires manual work to trigger the process;
   c) "−" in "Auto-scaling" also refers the tool can only on pre-existing resources instead of provisioning new ones;
   d) "−" in "Decentralisation" refers the tool requires the application developer to set up an infrastructure managing server for all the applications instead of for one application.

which are explained in Section 5.2.2. “Decentralisation” demonstrates the infrastructure management is in a decentralised way. If it is not, it means all the application developers’ infrastructures are managed by one administration, which requires everyone’s Cloud credentials. Some more details are explained in the footnote text of the table.

5.5 Conclusion

In this chapter, we propose a control model with two types of control modes. With the passive mode, the infrastructure is passively controlled through monitoring the infrastructure status. The developer can define the threshold to adjust the infrastructure through “*Runtime Control Policy*”. It is also a typical way to detect the system anomaly according to the monitoring feedback [32]. However, the monitoring based control is relatively delayed, because the action has to be taken after witnessing the result of some event. We, therefore, implement the other controlling mode, active mode. Firstly, with this mode, CloudsStorm allows the developer to actively provision and terminate virtual infrastructure resources from scratch: the infrastructure topology is defined in “*Infrastructure Description Code*”; the operations are defined by “*Infrastructure Execution Code*”. Secondly, with the active mode, CloudsStorm allows the application to actively adjust the infrastructure in advance according to the outside events, such as the input workload. This type of controllability is achieved by using “*Infrastructure Embedded Code*”. Compared to the passive mode, active control is beneficial to seamlessly adapt the infrastructure according to prior knowledge before influences actually occurring due to the varying workloads.

Besides, we describe the CloudsStorm controllability implementation in detail. Specific software design patterns are leveraged during implementation. They ensure that the infrastructure programming model based on basic Cloud VIFs proposed in Chapter 3 can be realised. Thus, the implementation achieves extensibility to support a new Cloud and efficiency of operations through multi-thread parallelisation. Finally, the experimental studies performed on the real Clouds demonstrate the infrastructure controllability implemented by CloudsStorm is efficient and outperforms others, which is essential to ensure the Quality of Service (QoS) of quality-critical applications.

6

Quality-critical Cloud Applications Development and Operation using CloudsStorm

Case studies and evaluations

In previous chapters, we have introduced the infrastructure programmability design in the development phase, the partition and network configuration mechanisms in the fast provisioning phase, and the controllability implementation in the runtime phase. Specifically, we have proposed a framework, CloudsStorm, to seamlessly and systematically handle the challenge of mitigating the gap between the Cloud infrastructure and the quality-critical Cloud application. From the Cloud DevOps perspective, CloudsStorm provides the functionalities for Cloud virtual infrastructure programming, automated networked infrastructure provisioning, and federated infrastructure runtime control. Then, the issue is how the applications can benefit from the CloudsStorm framework to operate on the Cloud virtual infrastructure considering the quality constraints.

In this chapter, we conduct experiments on real Clouds to show the case studies of using CloudsStorm to orchestrate quality-critical applications. We demonstrate four case studies from the perspective of task-based and service-based applications, respectively. For each case study, we present in detail how to leverage CloudsStorm for infrastructure programming and control. The evaluations of the case studies all show that CloudsStorm is an efficient framework to manage the distributed Cloud infrastructure for seamlessly satisfying the application quality requirements, no matter from the aspect of resource usage, i.e., budget, or from the aspect of infrastructure operation efficiency.

This chapter is mainly based on:

- **Zhou, H.**, Hu, Y., Su, J., Chi, M., de Laat, C., Zhao, Z., "Empowering Dynamic Task-Based Applications with Agile Virtual Infrastructure Programmability", In *IEEE 11th International Conference on Cloud Computing* (CLOUD), pp. 484-491. IEEE, 2018.
- **Zhou, H.**, Martin, P., Su, J., de Laat, C., Zhao, Z., "A Flexible Inter-locale Virtual Cloud For Nearly Real-time Big Data Application", In *IEEE Real Time System Symposium* (RTSS), *International workshop on Interoperable infrastructures for interdisciplinary big data sciences* (IT4RIs), 2016.

This chapter is also partially related with:

- **Zhou, H.**, Koulouzis, S., Hu, Y., Wang, J., de Laat, C., Ulisses, A., Zhao, Z., "Migrating Live Streaming Applications onto Clouds: Challenges and a CloudsStorm Solution", In *11th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (UCC Companion), *workshop on Cloud-Native Applications Design and Experience* (CNAX), pp. 321-326. IEEE, 2018.
- **Zhou, H.**, Taal, A., Koulouzis, S., Wang, J., Hu, Y., Suciu, G., Poenaru, V., de Laat, C., Zhao, Z., "Dynamic Real-Time Infrastructure Planning and Deployment for Disaster Early Warning Systems", In *International Conference on Computational Science, workshop on Data, Modeling, and Computation in IoT and Smart Systems*, pp. 644-654. Springer, 2018.
- Koulouzis, S., Martin, P., **Zhou, H.**, Hu, Y., Wang, J., Carval, T., Grenier, B., Heikkinen, J., de Laat, C., Zhao, Z., "Time-critical data management in clouds: Challenges and a Dynamic Real-Time Infrastructure Planner (DRIP) solution", Journal of *Concurrency and Computation: Practice and Experience*, e5269. Wiley, 2019. (as co-first author)
- **Zhou, H.**, Hu, Y., Ouyang, X., Su, J., Koulouzis, S., de Laat, C., Zhao, Z., "CloudsStorm: A Framework for Seamlessly Programming and Controlling Virtual Infrastructure Functions during the DevOps Lifecycle of Cloud Applications", Journal of *Software: Practice and Experience*. Wiley, 2019.

6.1 Case Study of Task-based Applications

Cloud environments provide elastic and on-demand services for running distributed applications. Typical Cloud service models include Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) [6]. Among those services, the IaaS model offers applications a rich capability for configuring virtual machines (VMs), networks, and customising 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 centres or Clouds. For service-based applications, such difficulties are not obvious, since the application mainly consists of long-term services. These applications typically only require a fixed number of VMs provisioned once. However, it becomes an urgent problem when supporting task-based applications such as on-demand processing tasks or scientific workflows, which are often highly distributed and do not run persistently in Cloud, but rather on demand.

6.1.1 Research Context and Related Work

Scientific applications are typical examples of task-based applications [125], such as earthquake prediction or genome sequence processing. These applications often share common features: 1) they all have distinct steps during runtime; 2) they do not run persistently to wait for requests; and 3) they take some data as input and output the results in the end. In order to run such applications on Cloud using SaaS or PaaS services, applications may be limited by the capability of configuring the platform's required 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. Thus, an

application-defined virtual infrastructure is often required by such applications. By using the Cloud IaaS, application developers need to provision them from particular providers, configure the required runtime environments, including network, and manipulate them at runtime. However, most of these operations are done manually.

Some infrastructure provisioning and deployment tools have been developed in past years, for instance, in supporting the software Development and Operations (DevOps) approaches of software engineering. For example, Puppet[1], Chef[2], and 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. Meanwhile, some Cloud providers provide vendor lock-in solutions, for instance, Amazon Web Service (AWS) CloudFormation[5] provided by the Amazon Elastic Compute Cloud (EC2). 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, Iman et al. [97] and Kee et al. [68] try to model the Cloud resource performance to mitigate the influence by the performance uncertainty of the Cloud, as well as to ensure the Quality of Service (QoS) for a scientific application. Hao et al. [94] leverage 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.

6.1.2 Problem Statement

Our experiments are conducted on real Clouds instead of simulators, and indeed, the Cloud performance uncertainty issue affects 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 lots of test cases 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 the common features of task-based applications mentioned in Section 6.1.1. Hence, we simulate a performance test on multiple data centres and Clouds to simulate the application scenario. The task of the application is to provision one VM in each data centre and leverage "sysbench"[6] to test the CPU and memory performance of each data centre. The provisioning overhead requires to be recorded as well. The bandwidth performance of each data centre should also be tested. In this hypothetical application scenario, these tests are required to repeat regularly. In essence, this is a common scenario for a lot of portfolio work [35, 79, 104].

[1] https://puppet.com
[2] https://www.chef.io
[3] https://www.ansible.com
[4] https://jujucharms.com
[5] https://aws.amazon.com/cloudformation
[6] http://sysbench.sourceforge.net/

Finally, the goal of this application scenario is to test: 1) whether the application developer can leverage CloudsStorm to program on the Cloud virtual infrastructure; 2) whether these task-based applications can run on Clouds effectively and get the results; and 3) whether Cloud resources can be provisioned on demand and released in time to reduce the cost as well as to satisfy the quality requirements.

6.1.3 Example Solution and Results

According to the problem statement above, we further consider two types of concrete scenarios, short-term and long-term scenario. In the short-term scenario, the performance test task does not need to be conducted for many times. In this case, we do not need "*Control Agent*", which is shown in Figure 3.5 of CloudsStorm overview. As for this short-term period of performance test, the "*Infrastructure Execution Engine*" running in the local computer is enough to conduct all the test cases without the issue of the local computer shutting down. It is corresponding to the solution that we set the field of "Mode" as "LOCAL" according to Syntax 5, when programming the "*Infrastructure Execution Code*". On the contrary, the performance test application requires to be executed for a long period, e.g., the tests need to be repeated for months. It is obvious that a regular local computer, e.g., laptop, is not able to perform a such long-term task. Then the "*Control Agent*" on Cloud is required. The detailed solutions and test results for these two specific scenarios are as follows.

Short-term scenario using "LOCAL" mode of CloudsStorm

To conduct experiments, we pick several data centres from two supported Clouds of CloudsStorm, EC2 and ExoGENI. There are three data centres 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 centre picked from EC2, which is the one located at California (CAL for short). For testing the network performance, we simulate a client from the ExoGENI UvA data centre to test the bandwidth with the above VMs at different locations. As for the comparison of different Clouds at the same level, all the VMs have 1 core and 1GB memory, which are "t2.micro" type for EC2 and "XOSmall" for ExoGENI.

In order to complete the task in this application scenario, we first design our networked infrastructure topology, according to Section 3.3.1. We define four sub-topologies with one VM in each sub-topology. All these four VMs are defined in the same subnet, which is "192.168.10.0/24". Specifically, the private addresses of the VMs are defined as follows: UvA: "192.168.10.11"; AUS: "192.168.10.12"; BBN: "192.168.10.13"; CAL: "192.168.10.14". In addition, we simulate another VM from UvA data centre 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". We then program the "*Infrastructure Execution Code*" to define the entire process for performing tests on the selected Clouds. Pseudocode 2 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 the CPU test and memory test simultaneously on all the object VMs. The parallelism is achieved by the definition of "||" in Section 3.3.2. The public IP

Pseudocode 2 Solution using "*Infrastructure Execution Code*" with "LOCAL" mode

```
Mode: "LOCAL"
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 concurrently on the 'VM' from AUS || BBN || CAL || UvA
    Perform bandwidth test to the UvA 'VM' ("192.168.10.10" → "192.168.10.11")
    Perform bandwidth test to the AUS 'VM' ("192.168.10.10" → "192.168.10.12")
    Perform bandwidth test to the BBN 'VM' ("192.168.10.10" → "192.168.10.13")
    Perform bandwidth test to the CAL 'VM' ("192.168.10.10" → "192.168.10.14")
    Get the results
    Terminate 'SubTopology' of AUS||BBN||CAL||UvA
    Wait for executing another round of tests
```

addresses of the other VMs cannot be determined and fixed, as they are assigned by Clouds dynamically. However, we can still always use the predefined private IP address to complete the bandwidth test. It is another advantage of provisioning a networked infrastructure on Clouds with our CloudsStorm. It shows the overlay network can make the infrastructure transparent to the application, no matter how the infrastructure is distributed. Finally, we get the results and wait for a while to perform these tests again. It is worth mentioning that during the interval of the test cases, all four VMs are programmed to be terminated. Besides, the "Mode" of this "*Infrastructure Execution Code*" should be set as "LOCAL". Due to the space limit, all relevant actual topology descriptions, codes, and results can be found on the GitHub repository[7].

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

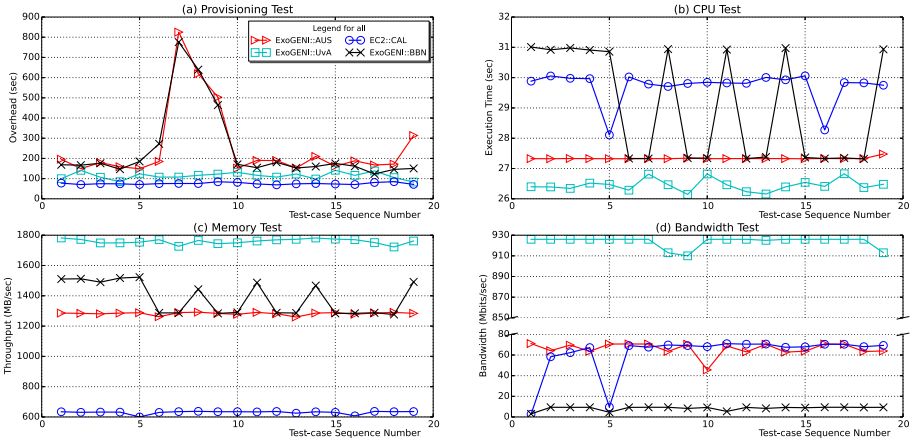


Figure 6.1: Performance test results of task-based applications in the short-term scenario (19 test cases) using CloudsStorm with the "LOCAL" mode

In this case, all the test results are recorded in the infrastructure code log, which is also shown in the repository[7]. The detailed description of this part is described in Section 5.3.3. Hence, we do not need explicitly to define an operation to retrieve results from remote VMs. We instead extract test results from the log, analyse it and present the results in Figure 6.1. It lists four types of results, including the result generated by the task of provisioning test, CPU test, memory test, and bandwidth test. The x-axis is the test case sequence number, as organised by iterations over the total test period. For this case, we only iterate 19 test cases to simulate the short-term scenario; 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 analysing these results, we can 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 resources from EC2 have not achieved high quality. There might be another workload influencing the data centre of AUS and BBN from ExoGENI simultaneously, as shown in Figure 6.1(a), causing the provisioning overhead at both data centres 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 centre. All the above demonstrate the feasibility and functionality of our programmable infrastructure framework.

Long-term scenario using "CTRL" mode of CloudsStorm

To be different from the above short-term scenario, which only performs 19 times test cases, we assume 1500 times of similar performance tests for a long-term scenario. According to the previous experiment, finishing 1500 times test cases require at least one month. Hence, developers' local computers usually cannot keep running for such a long period, i.e., it is not feasible to execute the "*Infrastructure Execution Code*" with the "LOCAL" mode.

To conduct experiments for this scenario, we pick four data centres only from ExoGENI Cloud, as ExoGENI is an experimental Cloud and free to use. These data centres are located in Sydney (in Australia, SYD for short), University of Amsterdam (in Europe, UvA for short), Boston (in the eastern US, BBN for short), and Oakland (in the western US, OSF for short). For testing the network performance, we test the bandwidth from the respective VM in the data centre of OSF, SYD, and BBN, to the VM in the UvA data centre. In this case study, all the VMs have 1 core and 3GB memory, which is the capacity of a "XOMedium" for VM in ExoGENI.

The sub-topologies and network design are similar to the solution for the short-term scenario. All these four VMs are in the same subnet, which is "192.168.10.0/24". To be specific, the private addresses of the VMs from different sub-topologies are defined as follows: UvA, "192.168.10.10"; OSF, "192.168.10.11"; SYD, "192.168.10.12"; BBN, "192.168.10.13". We then program the "*Infrastructure Execution Code*" to define the entire process for performing tests on the selected Clouds. Pseudocode 3 depicts the general procedure of how to program these operations on the infrastructure. The difference is that the mode of the "*Infrastructure Execution Code*" is set to "CTRL". In this case, a "*Control Agent*" will be first provisioned on a particular Cloud and then execute this "*Infrastructure Execution Code*" to control the Cloud resources and perform

Pseudocode 3 Solution using "*Infrastructure Execution Code*" with "CTRL" mode

```
Mode: "CTRL"
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
    Terminate 'SubTopology' of OSF||SYD||BBN||UvA
    Wait for executing another round of tests
```

the tests for a long period. For this long-term scenario, the detailed infrastructure topology descriptions and code can be downloaded via this link[8]. We are not going to show the code details, for length and simplicity reasons.

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

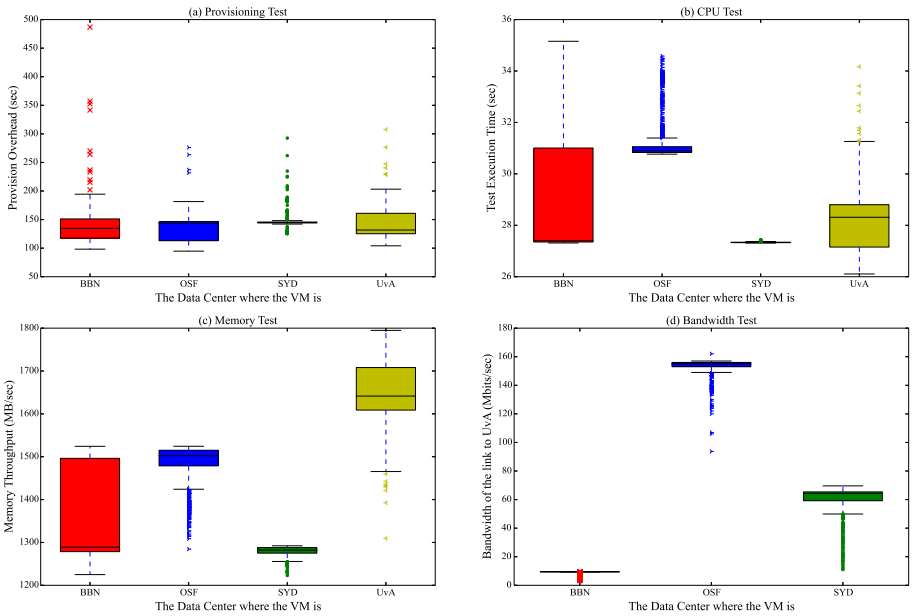


Figure 6.2: Performance test results of task-based applications in the long-term scenario (1500 test cases) using CloudsStorm with the "CTRL" mode

Similar to the short-term scenario, we still directly extract test results from the log, analyse it and present the results in Figure 6.2. 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 centres. For this case, we have performed 1500 iterations of test cases to obtain the results. From Figure 6.2, it can be derived from that the resources from different data centre perform with different qualities. The stability of the performance also varies. To conclude, such a number of performance tests are useful to characterise the performance of data centres. Notably, the "CTRL" mode of CloudsStorm plays an obbligato role for this long-term scenario here.

6.1.4 Evaluation

In this last subsection, we evaluate the efficiency of our framework for orchestrating quality-critical applications. Here, the efficiency refers to the budget constraints, i.e., how much monetary cost we can save through leveraging CloudsStorm to run these task-based applications. The efficiency 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, through the programmed operations of provisioning and terminating in the "*Infrastructure Execution Code*". The other is that it can perform some operations in parallel to reduce the entire execution time for the application. Thus, we first compare our cost with the cost of traditionally setting up Cloud resources manually. Then we 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[9], 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 infrastructure log of CloudsStorm as operation overhead.

[9]https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/

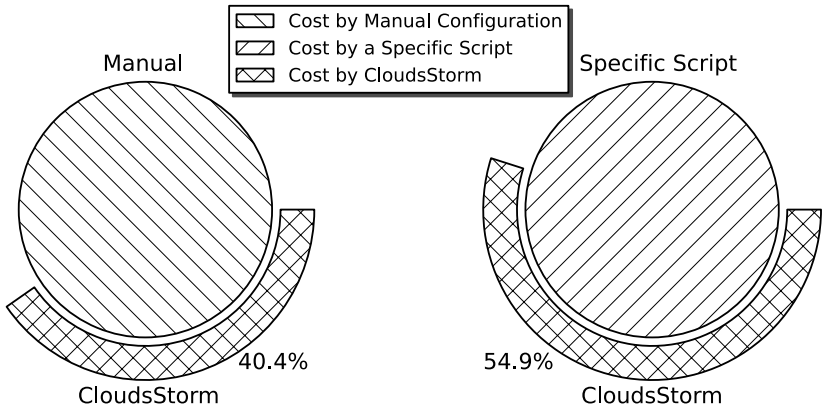


Figure 6.3: Cloud resource usage cost comparison

Table 6.1: Cloud resource usage evaluation among manual configuration, on-demand script configuration, and using CloudsStorm for the long-term scenario, being varied with the task iterations

	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%

Firstly, we analyse the result of the short-term scenario. The comparison result is shown in Figure 6.3. In one case, we set up these VMs manually, and run the application to perform tests, which 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. Anyhow, the traditional manual method completely wastes resources during this waiting period. The left part of Figure 6.3 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 terminating resources from some specific Cloud to reduce the wasteful waiting time. 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 among the VMs. Based on each operation overhead in the log, we calculate and conclude that CloudsStorm still reduces the total Cloud resource usage by 45.1% compared with using a specific script as shown in the right part of Figure 6.3.

Secondly, we analyse the result of the long-term scenario, which contains 1500 iterations of test cases. In this scenario, the actual execution time for each test case is about 5 minutes, and the remaining 10 minutes is waiting. The comparison results varying with the number of task iterations are shown in Table 6.1, where $cost$(MC) and $cost$(OC) still refer to the Cloud usage cost of manual and on-demand script configuration, respectively. In the case of MC, we still set up these VMs manually and keep them occupied during the entire execution of the tests. In the other case of OC, we still develop a specific script to automate provisioning and terminating resource from some specific Cloud to reduce the wasteful waiting time, but without parallel executions. 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 6.1 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.

Besides the manual and on-demand script usage of Cloud, the developer can also leverage the Cloud APIs to make the operations in parallel and achieve similar efficiency as CloudsStorm. However, compared with CloudsStorm, we argue that the approach of leveraging Cloud APIs requires advanced programming skills, is not extensible to support different Clouds, and cannot benefit the networked infrastructure.

6.2 Case Study of Service-based Application

Comparing to traditional simple service-based applications, such as websites hosting service and online file storage service, the emerging big data processing applications and Internet of Things (IoT) applications consist more sophisticated services and quality constraints. There are mainly two aspects of challenges to satisfy the application QoS: the latency and location requirements when distributing the infrastructure and application components; the elasticity to adjust the infrastructure according to the fickle outside events, e.g., input workload and connecting devices. Still, IaaS Clouds provide possible solutions, but without enough controllability and programmability to be leveraged by the developer. In this section, we are going to discuss the ability of CloudsStorm to tackle these issues.

6.2.1 Research Context and Related Work

Nowadays, service-based applications, like big data applications, not only consider the amount of data they can process but also consider the processing time. Data needs to be processed and fed back to users in order to inform decisions and improve runtime steering of applications. This requirement constitutes the so-called "nearly real-time" constraint on application feedback and steering. Network transmission time is an essential factor to influence the processing time. However, data collectors may be distributed in different locations for specific large-scale big data applications [95]. In some cases, the physical distance between the data collector and the data user can be significant. For distributed applications executed over the Internet, the effective processing time is often mainly determined by the network transmission time. Meanwhile, because of the emerging IoT applications, the data sources can be distributed over different geolocations. Furthermore, the fact that these data sources are mobile and need to process data near the data source [82] requires the underlying infrastructure to be location-aware and adaptive to fit the applications. Even there are regulations that the data must be processed in somewhere and is forbidden to export to other countries. For instance, the General Data Protection Regulation (GDPR)[10] is a regulation enforced recently by European Union (EU) commission to manage all the data, and the personal data should not be exported outside European Economic Area (EEA) without authorisation. The location of computing resources to process those data, therefore, should be explicitly considered according to the requirements.

On the other hand, traditional data management and processing applications require a platform and infrastructure to be configured in advance. For example, Hive [107], which is a Hadoop based database, needs to first deploy the distributed file system, Hadoop Distributed File System (HDFS) [100]. Moreover, HDFS is also deployed on a fixed infrastructure, which is either a cluster of physical machines or a set of VMs. To be specific, the entire procedure to set up a data processing service includes: 1) provision a set of computing resources; 2) configure their network to be a cluster; 3) deploy the data processing application; and 4) execute the application and provide data processing service. However, since the input data size of requests is different

[10] https://eugdpr.org/

every time, the requirement for the underlying infrastructure also varies. Therefore, it is inevitable that the fixed infrastructure could be either over provisioned to waste resources or insufficient to complete the data processing within particular deadlines.

Cloud computing can be a powerful solution for big data processing, as the computing ability of servers is constantly increasing and more data centres are being set up around the world that can be used to support these applications. It is still a problem, however, for application developers to access and manage their Cloud resources. Especially, this kind of big data application is a type of quality-critical application, which often requires customised virtual infrastructure with tailored SLAs (Service Level Agreements) when migrated into a Cloud environment [126]. There exist studies which discuss using Cloud to solve the big data application challenges: Changqing et al. [59] introduce some popular tools and Cloud platforms to do large-scale big data processing, while Rajiv [95] proposes a high-level architecture of large-scale data processing service. The underlying resource layer of the overall architecture is scalable across multiple data centres or even Clouds, but both of these studies do not mention how to help application developers manage virtual Cloud resources and achieve better performance. Moreover, some other Cloud tools, such as Amazon Web Service (AWS) Lambda[11], only focus on SaaS (Software-as-a-Service) platforms to orchestrate services, without revealing the ability for the developer to take nearly real-time constraints into account.

6.2.2 Problem Statement

Figure 6.4 illustrates a typical architecture and stages of a big data application. The data are first collected from some devices or data sources in the data acquisition phase. Some computing resources are required to perform data pre-processing. Finally, the data is analysed, and the results are stored in a database for data users to query. Traditionally, the computing resources are physical servers intensively located in some place. However, this kind of nearly real-time big data application we discuss in this chapter can be particularly large-scale. The data collectors can be distributed all around the world, especially when combining the big data application with IoT [25] or sensor networks [4].

Being faced with the two challenges of operating service-based applications on Clouds mentioned at the beginning of this section, we consider the challenges of infrastructure management as follows.

1. **Networked infrastructure.** The applications workflow becomes more complex, with many components that need to communicate with each other. Separated instances cannot complete the whole job. For instance, the components in different stages in Figure 6.4 need to communicate with each other and transfer data. The virtual infrastructure must, therefore, realise a particular network topology.

2. **Nearly real-time constraints.** Nearly real-time applications require that most task deadlines should be met over the lifetime of the application. The network transmission is a crucial part to satisfy the deadlines. Considering the distribution of the data collectors, e.g., cameras providing video of a live event, in Figure 6.4, the centralised processing resources can hardly satisfy the constraints.

[11] https://aws.amazon.com/lambda/

3. **Auto provisioning and federated Cloud.** Since these applications are complex, we need a way to provision the entire infrastructure and deploy applications automatically. Currently, some tools can only provision automatically at the VM level, e.g., jclouds[12]. On the other hand, we may need more resources from other Clouds to provision a large-scale infrastructure [124]. And also more Clouds provide more opportunities to find a better-geolocated server in the data pre-procssing phase shown in Figure 6.4. However, it is a problem to combine these resources across multiple locales.

4. **Elasticity.** Figure 6.4 illustrates a highly flexible environment. In the data acquisition phases, the data collectors like IoT devices can move around. The workload of the processing servers would also vary. The computing resource, i.e., the servers in Figure 6.4 of the infrastructure, should be elastic to the input conditions to save cost.

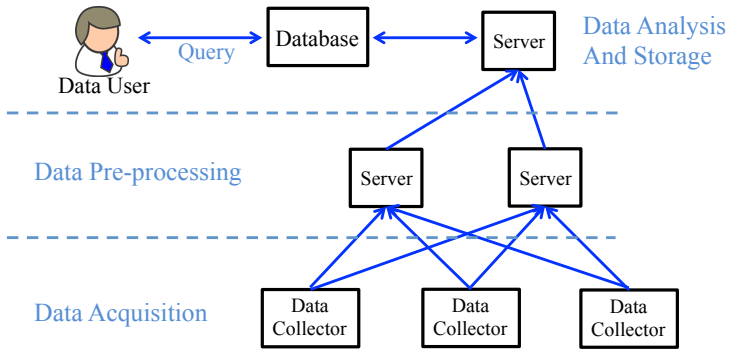


Figure 6.4: Typical architecture of big data, IoT applications

We assume two specific problem scenarios for this case study: 1) The first scenario is the nearly real-time data processing. It is abstracted from the eddy covariance data processing service[13] of ENVRIplus[14] project, which is an example of a typical nearly real-time big data application. This service measures wind and gas concentration at sites over different ecosystems. It keeps collecting these data all year round, and it needs to calculate the net exchange of gases, energy, and temperature between ecosystems and atmosphere under time constraints. In this scenario, we would like to identify whether the infrastructure geo-distribution can influence the application QoS and how CloudsStorm can help; 2) The second scenario is the data-aware processing. To emulate the scenario, we execute the word count program on a Hadoop cluster with different resource scales and different workloads as input data. We would like to demonstrate how CloudsStorm can dynamically adjust the infrastructure according to the input conditions and what the benefits are.

[12] https://jclouds.apache.org/
[13] https://wiki.envri.eu/display/EC/IC_13+The+eddy+covariance+fluxes+of+GHGs
[14] https://www.envriplus.eu/

6.2.3 Example Solutions and Evaluations

In this section, we propose solutions using CloudsStorm for the above assumed scenarios and present evaluation results, respectively.

The scenario of nearly real-time data processing

Following the typical architecture of big data application shown in Figure 6.4, we propose a solution architecture for those kinds of nearly real-time big data application using CloudsStorm, specifically for the use case of the eddy covariance data processing service as formulated within the ENVRIplus project, shown in Figure 6.5. In the ENVRIplus project use case, the data collectors are significantly spread out geographically and often do not have high-quality network access to the Internet. If the data collector is geographically far from the processing server, then the network performance will be too low to satisfy real-time requirements, e.g., to transfer a certain amount of data within a particular time limit. Junchen et al. [61] point out, however, that the emergence of private back-bones in recent years to connect globally distributed data centres can serve as a readily available infrastructure for a managed overlay network. Osama et al. [51] use Cloud-based overlays to afford a packet recovery service. We then adopt this idea of using the Cloud-based network instead of the pure Internet-based network to try and satisfy the nearly real-time requirements of the application.

Shown as Figure 6.5, CloudsStorm is a key component of the architecture, acting as an automatic provisioning agent which provisions not only the virtual computing resources but also the network connections. With the connection techniques described in Chapter 4, CloudsStorm is able to integrate resources from different data centres or Clouds as a single infrastructure. From the application developers' point of view, CloudsStorm can set up a virtual Cloud with the application-defined network. They do not need to consider where the virtual resources come. They can always use their

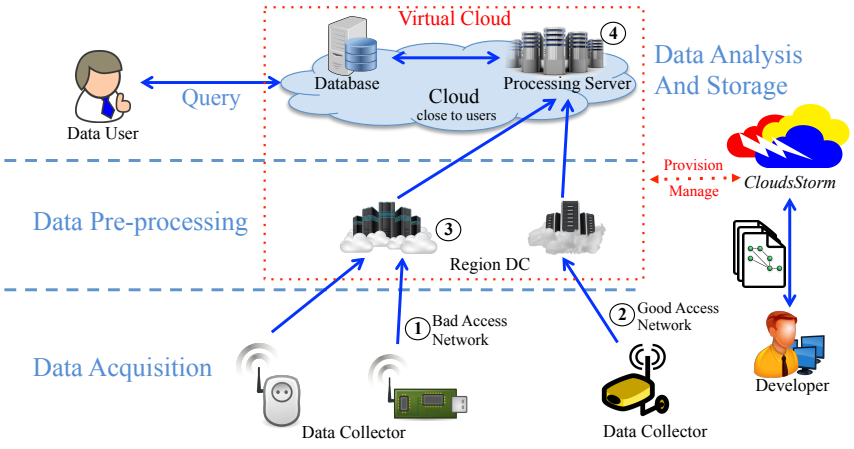


Figure 6.5: Solution architecture for nearly real-time big data applications using CloudsStorm.

own user account and unified SSH private key, which are defined in the programmed "*Infrastructure Description Code*", to access all the resources on the virtual Cloud. With this help, developers can focus on developing applications based on the networked infrastructure with customised private IP addresses. Finally, CloudsStorm can then take all the programmed infrastructure code as input to provision the distributed networked infrastructure and run the application automatically.

According to the use case, the data collectors collect data from different ecosystems. In order to satisfy the nearly real-time constraints, CloudsStorm must provision the resources from regional data centres close to data collectors. The reduced latency between the acquisition and pre-processing stages makes data collectors forward data more efficiently. The network performance between pre-processing and analysis is better than that of directly sending all data from collectors to the final processing servers. Besides, the application must keep running all year round. CloudsStorm is also able to make it recover from failures fast to satisfy another nearly real-time constraint. Moreover, the scalability across Clouds makes the infrastructure more flexible for meeting dynamic constraints at runtime, which will be discussed in the next scenario.

For evaluation, we set up experiments to test the feasibility of the solution provided by CloudsStorm. In order to emulate the real situation, we create four objects in the experiment. The detailed properties and settings of these objects are listed in Table 6.2. We use a laptop to act in the role of a data collector and put it in different network environments. For object 1, the laptop is connected with the home network via WiFi. This object is designed to simulate the situation where the data collector has a relatively poor access network connection. Object 2 is deployed within the campus network of UvA (University of Amsterdam) to emulate the situation where the data collector has a particularly good network connection. Objects 3 and 4 are two VM nodes provisioned by CloudsStorm within different locales of data centres of ExoGENI Cloud. They are connected via private IP addresses far from each other geographically. Object 3 acts in the role of virtual resources provisioned in the regional data centre in Figure 6.5, while object 4 acts in the role of remote virtual resources close to the data user. In this section, we need to compare two main experimental scenes. The first scene is the deployment of all the components in one data centre without using CloudsStorm, i.e., only leverage Object 4 to collect the data and perform processing. The second scene is to adopt our solution, which is to distribute the components on the virtual Cloud provisioned by CloudsStorm, i.e., leverage Object 3 to collect data and Object 4 to perform processing.

Table 6.2: Properties of objects in the experiment

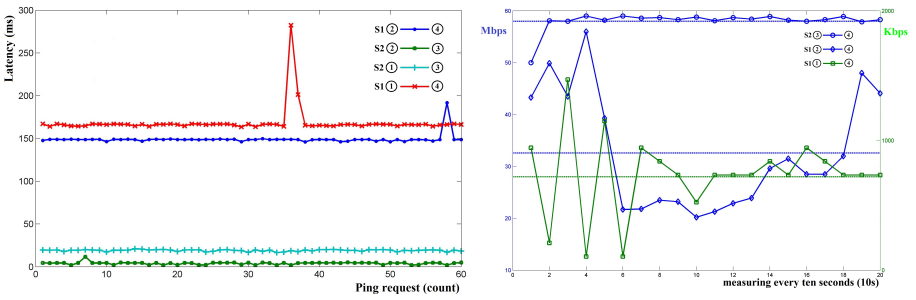
Number	Object	Access Network Properties			Geography Properties	
		Mode	Upload	Download	Cloud	Location
①	Laptop	WiFi[1]	0.94 Mbps	8.59 Mbps	-[2]	Amsterdam
②	Laptop	WiFi[3]	193 Mbps	305 Mbps	-	UvA
③	VM	Ethernet	-	-	ExoGENI	UvA
④	VM	Ethernet	-	-	ExoGENI	CA, US

[1] It is connected with the home network.
[2] '-' means not applicable.
[3] It is connected with eduroam.

We, therefore, conduct the first experiment to test the latency. The results are shown in Figure 6.6(a). We start sixty ping requests one by one between different objects of Table 6.2. The legend in Figure 6.6(a) tells the link between which two objects. Besides, "S1" preceding the legend indicates that it refers to the first experimental scene, i.e., without CloudsStorm, described above, and "S2" is for the second scene, i.e., with CloudsStorm. The experiments show that the latency is lower when the data collector is closer to the computing resources. In the first scene without our solution, even though the data collector has good access network, the average latency is still nearly ten times higher than those in the second scene. Moreover, both of the latency in scene 1 are not stable, especially when the access network has low quality, which is common for real data collectors. It is also worth pointing out that real data collectors are typically not as powerful as the laptop used in this experiment, and so the real performance may be even worse. It shows that our solution of distributing the infrastructure can reduce latency.

The second experiment is to test the bandwidth in these two scenes. Figure 6.6(b) shows the results. We measure the bandwidth continuously over 200 seconds. The corresponding y-axis of all blue lines in this figure is on the left, measured in "Mbps". The corresponding y-axis of the green line is on the right, measured in "Kbps". This figure shows that the quality of the Cloud-based network is better than the direct connection. The link between the two VMs, i.e., Objects 3 and 4, provisioned by CloudsStorm uses a Cloud-based network which exhibits superior bandwidth. If we deploy the application without our solution, data collectors are required to connect to the faraway server directly. Two lines in Figure 6.6(b) with "S1" denotes the performance. Although Object 2 is in a good access network environment, the average bandwidth is 26 Mbps less when it is directly connected to the faraway server. Moreover, it also shows that the bandwidth of the Cloud-based network is more stable than the direct connection. In addition, the green line shows that when data collectors do not have a good access network, the bandwidth is much worse.

The transmission time for data collectors can, therefore, be reduced using our solution. CloudsStorm can set up a virtual Cloud that considers the underlying network in order to better satisfy the nearly real-time requirements of the application to the extent that it is possible. This kind of consideration is essential for data collectors to



(a) Comparison of latency.

(b) Comparison of bandwidth.

Figure 6.6: Evaluation of application-defined network connection performance

work more efficiently as part of a distributed system. In this scenario, we mainly test the static nearly real-time constraints above, which is about the ability to provision the infrastructure to satisfy the network requirements. For the runtime nearly real-time constraints, we demonstrate it in the next scenario to show how the infrastructure can be adaptive to the workloads.

The scenario of data-aware processing

For another scenario, the experimental study we conduct with the CloudsStorm framework is related to 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 the input data size. The underlying infrastructure can, therefore, provide a proper amount of computing resources with network connections on demand, without over-provisioning. Via this manner, the computing resource consumption is reduced.

In order to develop this data-aware processing application, the application developer would first design the infrastructure topology. For demonstration, there are two sub-topologies 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 centre of ExoGENI. Here, sub-topology, "data_src_1", is from the "UvA" data centre located in the Netherlands to emulate the data source of this application. Sub-topology, "hadoop_1_node", is designed to be from the "UMass" data centre located in the USA. The VM defined in the sub-topology "data_src_1" is termed as "dataSrc1", and the one defined 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 realised through utilising the "*Infrastructure Description Code*" and the "*Infrastructure Execution Code*", which are similar with the case studies in Section 6.1. However, in this case study, we further leverage the "*Infrastructure Embedded Code*" of CloudsStorm to help the application achieve finer-grained controllability on the infrastructure. Listing 6.1 below shows how the function of the data-aware processing part is implemented within the CloudsStorm framework using 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, e.g., the data size returned for a 4.2 GB file is 5. Then, the developer 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 centre 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 application example, we first upload the input data onto the Hadoop Distributed File System (HDFS) in order to be processed. In this phase, no more computing resources are needed as long as the storage of the one-node cluster is enough for the input data. When the operation of uploading the input 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 "wordCountJob" is started with a scaled cluster to achieve better computing performance. After executing the task, the result is stored in the corresponding folder of HDFS. Extra VMs, which offer the computing resources, therefore, can be released to save cost.

```
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.
  //The specific scaling number is programmed according to the input 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 6.1: Infrastructure embedded code example in data-aware processing

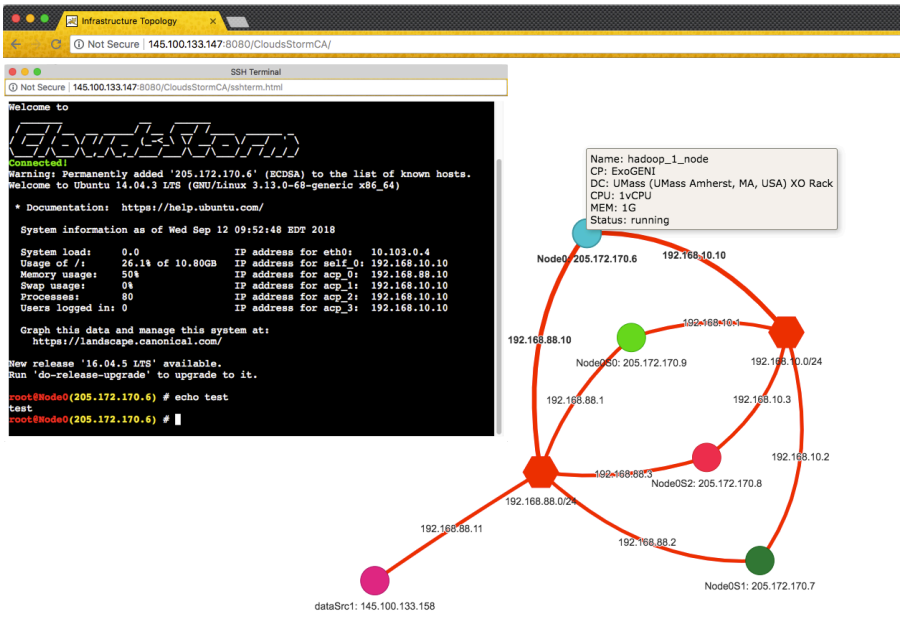
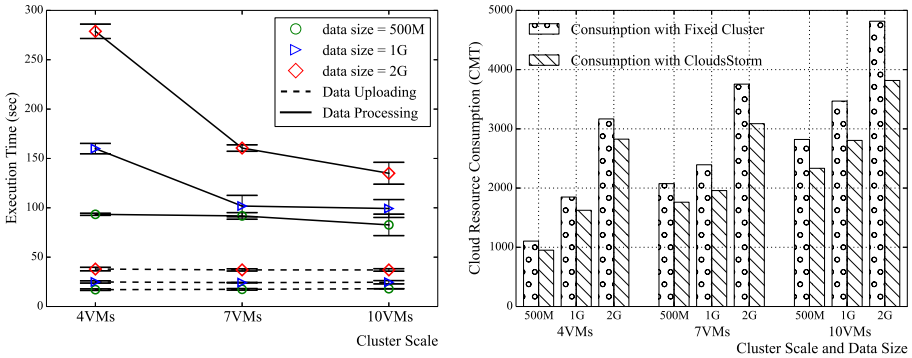


Figure 6.7: Example infrastructure topology from the CloudsStorm GUI

Figure 6.7 is a snapshot to demonstrate the web-based Graphical User Interface (GUI) offered by the "*Control Agent*" for this example. In this case, the "*Control Agent*" is also dynamically set up in the "UvA" data centre, and its public IP is "145.100.133.147". After being aware of this public IP, the developer is able to access this GUI from the browser. Figure 6.7 shows the networked infrastructure description. It illustrates how "Node0" constructs the original one-node Hadoop cluster. At the time of this snapshot, three scaled VM, "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 colour of the circle identifies the sub-topology, to which this VM belongs. The VM type and location, i.e., Cloud and data centre, information can also be popped up when hovering over the circle. It is also worth mentioning that 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 via this link[15]. Due to the space limitations of the thesis, we are not going to show the code details.

For the evaluation of this solution, we conduct several experiments on this case study using CloudsStorm. We execute the word count program on a Hadoop cluster with different scales of computing capacity and feed the program with different amount

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



(a) Execution time in different phases

(b) Cloud resource consumption comparison

Figure 6.8: Data-aware processing evaluation using the CloudsStorm framework

of input data. There are two groups of experiments. One is conducted with running a traditional word count program on a Hadoop cluster of fixed scale. The other one is conducted with the newly-developed word count program using the data-aware processing function similar to that shown in Listing 6.1. Each experiment is repeated five times.

Figure 6.8(a) 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 customise the infrastructure computing capacity properly according to the input data size. Through leveraging CloudsStorm, the infrastructure can be programmed to adjust the input data size dynamically. Compared to the traditional application case, redundant computing resources can be avoided.

Figure 6.8(b) demonstrates that the Cloud application programmed with CloudsStorm 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, i.e., cores), 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, as 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 finer-grained controllability on its virtual infrastructure.

6.3 Conclusion

This chapter presents case studies on using CloudsStorm to operate two types of quality-critical applications, i.e., task-based and service-based applications, on Clouds. Among them, the platform-based application using the Hadoop platform for data processing is also demonstrated within the CloudsStorm framework. For different specific scenarios, we detailedly explain how to leverage the infrastructure code proposed and designed by CloudsStorm framework to describe the infrastructure topology and the operations performed on the infrastructure. CloudsStorm also provides the "*Infrastructure Execution Engine*" and "*Control Agent*" to interpret the infrastructure code and perform the actual operations. With these components, it demonstrates how to build a Cloud application from scratch, i.e., without infrastructure provisioned in advance. Through experimental studies conducted on real Clouds and the comparison with other ways of operating the Cloud applications, CloudsStorm is an efficient framework and able to operate the Cloud virtual infrastructure to satisfy the application QoS, including the budget constraints.

To summarise, there are following advantages to adopt CloudsStorm in the Cloud application DevOps lifecycle: 1) CloudsStorm achieves provisioning of resources on demand and release of them in time. Thus, CloudsStorm is able to reduce the resource usage cost compared with other traditional methods. 2) CloudsStorm is able to define and perform the operation on the infrastructure in parallel to improve efficiency. 3) CloudsStorm is able to provision the customised federated Cloud infrastructure and the application-defined overlay network, according to the data distribution for satisfying the network requirements. 4) CloudsStorm allows the application to embed the infrastructure operation logic to dynamically adjust the virtual infrastructure for maintaining the application QoS, according to the outside events, e.g., various random workloads as inputs.

7

Enhancing the Cloud Application Quality Assurance through the Trustworthy Enforcement of Service Level Agreement

In previous chapters, we demonstrate that CloudsStorm framework empowers the Cloud virtual infrastructure with the programmability and controllability for the developers to seamlessly operate their applications on Clouds. The virtual infrastructure can, therefore, be better programmed and controlled to satisfy the quality requirements of the applications. However, the requirements cannot be absolutely ensured because of the essence that the Cloud resources are not physically owned by the Cloud customers, i.e., the application developers in our case. Cloud Service Level Agreement (SLA) is a general solution for this issue, which defines how the customer can get compensation from the provider if the violation of the Cloud resources happens. However, traditional Cloud SLA suffers from lacking a trustworthy platform for automatic enforcement. The emerging blockchain technique seems to provide a promising solution.

In this chapter, we first analyse the SLA enforcement requirements for quality assurance and explore the state of the art using blockchain. It is still challenging to prove that the off-chain SLA violations really happen before being recorded into the on-chain transactions. To tackle this challenge, we propose a witness model using game theory and the smart contract technique to enhance the trustworthiness. Specifically, we have prototyped the system[1] based on the smart contracts of Ethereum blockchain.

This chapter is based on:

- **Zhou, H.**, Ouyang, X., Ren, Z., Su, J., de Laat, C., Zhao, Z., "A Blockchain based Witness Model for Trustworthy Cloud Service Level Agreement Enforcement" In *IEEE International Conference on Computer Communications* (INFOCOM), pp. 1567-1575. IEEE, 2019.
- **Zhou, H.**, de Laat, C., Zhao, Z., "Trustworthy Cloud Service Level Agreement Enforcement with Blockchain Based Smart Contract" In *IEEE International Conference on Cloud Computing Technology and Science* (CloudCom), *workshop on resource brokering with blockchain* (RBChain), pp. 255-260. IEEE, 2018.
- **Zhou, H.**, Ouyang, X., Su, J., de Laat, C., Zhao, Z., "Enforcing Trustworthy Cloud SLA with Witnesses: A Game Theory based Model using Smart Contracts", Journal of *Concurrency and Computation: Practice and Experience*, e5511. Wiley, 2019.

[1] https://github.com/zh9314/SmartContract4SLA

7.1 Application Quality Assurance

Cloud SLA is an approach to driving the Cloud provider to offer the promised Cloud infrastructure service. It is also the last assurance for the application to satisfy the expected quality requirements, from an economic perspective. Thus, it is essential to analyse the challenges of current SLA when empowering the application with quality assurance. Then, we investigate the related work and state of the art, which adopts the blockchain based solutions.

7.1.1 Quality Assurance Requirements Analysis

Cloud computing is a popular business model nowadays for sharing physical resources among multiple tenants. A Cloud customer can rent and use various resources as service, including computing, storage, and network, from the provider through a network (typically via the Internet) without maintaining the physical hardware. Although convenience is conveyed, this causes the challenge of "*Cloud Performance Unpredictability*" [6] when migrating time-critical applications onto Clouds [126]. Cloud SLA is, therefore, proposed to ensure that specific service quality can be met, and in the case of violation, the customer can get the corresponding compensation from the provider. It compensates the customer's lost due to the Cloud performance uncertainty from an economic perspective.

Traditionally, SLA is a business concept which defines the contractual financial agreements between the roles who are engaging in the business activity [36]. In the context of Cloud computing, it is an agreement between the Cloud customer and provider on the quality of the Cloud service. For instance, the Infrastructure-as-a-Service (IaaS) provider, Amazon Elastic Compute Cloud (EC2), claims that the availability of its data centre is no less than 99%. If this number is not achieved, it will pay back 30% credits to its customer as compensation[2]. However, this agreement is hard to be enforced in practice. The significant challenges that hinder the conceptual SLA to be feasibly adopted in the real-life industry are:

- **The Manual verification.** An automatic mechanism to enforce the agreement is missing, especially the automatic compensation after SLA violation. The current process involves a lot of manual work, such as doing the verification through emails before compensation: the customer needs to submit a claim to the EC2 website for its SLA violation[2].
- **The fairness between different roles.** The provider has more rights in the current agreement model, as it has the right to verify the violation and decide whether to compensate the customer. On the contrary, the customer has little space to negotiate about the price or the amount of compensation. They can only choose whether to accept the SLA provided by the provider.
- **The proof of violation.** Currently, the agreement is only between the Cloud customer and the provider. It is hard for the customer to prove and convince the provider that the violation has really occurred. For example, EC2 requires

[2]https://aws.amazon.com/compute/sla/

the customer to provide detailed logs that record the date and time of each unavailability incident[2].

The blockchain [85] technology brings in new opportunities for tackling these challenges. The smart contracts in Ethereum [21] provide a feasible way to automate the service transactions and enforce the SLA via the blockchain. Hiroki et al. [86] introduce a "*Service Performance Monitor*" role to detect the violation and notify the user. However, the proposed solution lacks an analysis of the credibility on the identified violations and still faces challenges in achieving consensus on an event that happens outside the blockchain. The bridge between the events that on and outside the chain is called "*oracle*"[3]. One of the solutions is to retrieve data from "oraclize"[4], a third trusted company performing as a trustful data source. Moreover, these solutions are centralised, which suffer from single-point of failure and are easy to be compromised.

7.1.2 State of the Art

SLA is a well-discussed research topic, specifically in the context of Cloud computing. It establishes the quality of service agreement between the service provider and the customer, which ensures the customer's benefit when the agreement is violated. A typical SLA lifecycle consists of multiple enforcement phases, including negotiation, establishment, monitoring, violation report and termination [36]. Most of the research focuses on three aspects: 1) syntax definition of the SLA terms and parameters, the goal of which is to standardise the representation so that SLA can be efficiently processed online by computer systems. A set of domain-specific languages is proposed to solve the issue, such as SLA* [67], SLAC [109], CSLA [69]; 2) resource allocation techniques to ensure the SLA. The work in this aspect focuses on the algorithm to optimise resource allocation, thereby avoiding SLA violation from happening. SLA in this kind of work is typically considered as constraints, e.g., [24, 60]; and 3) systems or methods to address issues in specific phases of the SLA lifecycle. According to a systematic survey [36] on Cloud SLA, among this kind of works, 22% are focusing on negotiation and establishment phase, 73% are targeting at monitoring and deployment phase, 3% are interested in SLA violation management while 1% focus on reporting. For these phases, the goal of negotiation is to maximise either the provider's or the customer's rewards through adopting some negotiation strategy [50]. Monitoring is mainly about what to monitor and how to automate the process [46]. However, the most challenging phases, violation management and reporting, are seldom explored. In industry, Amazon CloudWatch service[5] is an example that the provider automates monitoring and notification. In this case, the customer has no choice but to trust the provider. Muller et al. [84] develop a platform named SALMonADA to deal with the SLA violation at runtime. It works as a third trusted party to perform the monitoring and violation report. All these work assumes that the violation report is trustworthy, which is, however, the most challenging part in the case of a violation.

[3] https://blog.ethereum.org/2014/07/22/ethereum-and-oracles/
[4] http://www.oraclize.it
[5] https://aws.amazon.com/cloudwatch/

Smart contract is proposed to digitally facilitate, verify and enforce a contract through a computer protocol [26]. Some explorations, e.g., Vincenzo et al. [98] combines this concept with Cloud SLA negotiation, focusing on the semantic expression of smart contract to automate the negotiation phase. However, most of them lack a trustworthy platform to execute a smart contract. This is actually essential because the smart contract relies on a strong assumption that no one can tamper its execution. Town Crier [121] and TLS-N [96] ensure the trustworthy execution and communication environment from the hardware and transmission protocol level, respectively. However, they are either centralised or require special hardware support.

Blockchain [85] is a promising technique to be the execution platform since the interactions on the chain are immutable. Especially, Ethereum [21] first realises to execute a general-purpose program on its blockchain. They design several programming languages, which make it possible to implement smart contracts. Hiroki et al. [86] leverage Ethereum and design a set of web APIs. They attempt to automate the SLA lifecycle enforcement on the blockchain. A new role called "*Service Performance Monitor*" is introduced in their paper, who is responsible for the violation management and reporting. However, whether the violation reports sent to the blockchain can be trusted is not discussed. In essence, this is still a gap for blockchain based systems. That is how to credibly record a random event happening outside of the blockchain onto the chain. It is called "*oracle*"[3], which is a party performing as a "data-carrier" for the blockchain. "oraclize"[4] is a third trusted company currently offering the service as an oracle. Nevertheless, a third trusted party can lead to a single-point failure. Relying on the centralised party also deviates from the decentralisation idea of blockchain. Hence, ChainLink [34] works on distributed oracles. Only when an agreement is achieved among the oracles, the result data of the event can be carried onto the chain or trigger a transaction. However, it has some downsides, such as no incentive for individuals to do this, requiring individuals being independent and trustworthy, and the consensus issue among oracles.

7.2 The Witness Model using Smart Contract

The roles involved in the proposed model are introduced in this section, specifically, the role of the witness. The overall system architecture for SLA enforcement using the smart contract on blockchain is illustrated afterwards, followed by a detailed description of the responsibility of the witness: service violation detection and report.

7.2.1 The Witness Role and Assumptions

There are mainly two roles in the traditional Cloud SLA lifecycle. One role is a Cloud provider, P, which offers Cloud service. The other role is a Cloud customer, C, which consumes the Cloud service and pay the service fee. To demonstrate the key contribution of our work, we take a concrete example to formulate our problem as follows.

A Cloud provider, p, is an IaaS provider. It provisions virtual machines (VMs) on demand with public addresses for customers to use. For instance, according to the request of a customer c, provider p provisions a VM with a public IP address, IP_{pub}.

During the service time, $T_{service}$, the customer, only the customer c is able to SSH and login to the VM through the corresponding address IP_{pub}. In this case, the SLA can be that, the provider p claims that during the service time the provisioned VM will always be accessible. If this is true, the customer c must pay the service fee, $F_{service}$, to the provider p after the service ends. Otherwise, the customer c can acquire a compensation fee, $F_{compensation}$. That is the customer c only needs to pay $F_{service} - F_{compensation}$ to the provider p in the end, where we assume that $F_{service} > F_{compensation}$. For the latter case, if it happens, we define it as a SLA violation event. In addition, it is worth mentioning that we should exclude the case that the inaccessibility is caused by the customer's own network problem, to be a violation event.

With only these two roles in the agreement, it is hard to ensure that the provider can get paid or the customer can get compensation paid back if the service fee is prepaid. Hence, we leverage blockchain to play as the trusted party to afford a platform for these two roles and enforce these monetary transmissions. But it is still especially difficult to convince both roles whether the violation happens and whether it is caused by the customer's own network problem. We, therefore, bring in another new role in the traditional SLA lifecycle, named as witness role, W. They are also the normal participants in the blockchain and volunteers to take part in our SLA system to gain their own rewards through offering monitoring service. In order to solve the trust issue, a set of N witnesses, $\{w_1, w_2, ..., w_N\}$, is selected to form a committee in a specific SLA lifecycle. They together report the violation event and may obtain witness fee, $F_{witness}$, as rewards from both the provider and the customer. Moreover, the wallet address of a specific role on the blockchain is denoted as a filed value of it, $x.address$. For instance, $w_k.address$ is the wallet address of witness w_k.

In this paper, we make the basic assumption on the witness role that it is always selfish and aims at maximising its own rewards.

7.2.2 Overall System Architecture

Figure 7.1 illustrates the system architecture we design for Cloud SLA enforcement. It consists of two types of smart contracts on the blockchain. One is the witness-pool smart contract, which is the fundamental smart contract of the system to manage all the registered witnesses. The other type is the SLA smart contract for a specific SLA enforcement. The responsibilities of the witness-pool smart contract include witness management, specific SLA contracts generation, and witness committee construction. Any user of the blockchain, who has a wallet address, can register its wallet address into the witness pool to be a member of witnesses. They can keep themselves online and wait to be selected for some specific SLA contract. The incentive for the witness to participant in this system is to obtain rewards. Moreover, the more witnesses participant in the system, the more reliable and trustful the system would be.

The entire SLA lifecycle then becomes as follows. Certain provider p first leverages the smart contract of witness pool to generate an SLA smart contract for itself. Prior to setting up SLA, the customer c should negotiate with the provider p about the detailed SLA terms, including $T_{service}$, $F_{service}$, $F_{compensation}$, etc. Here, one of the most important terms is to determine N, which is the number of witnesses would be involved in the enforcement of this SLA. The more witnesses involved in an SLA, the more

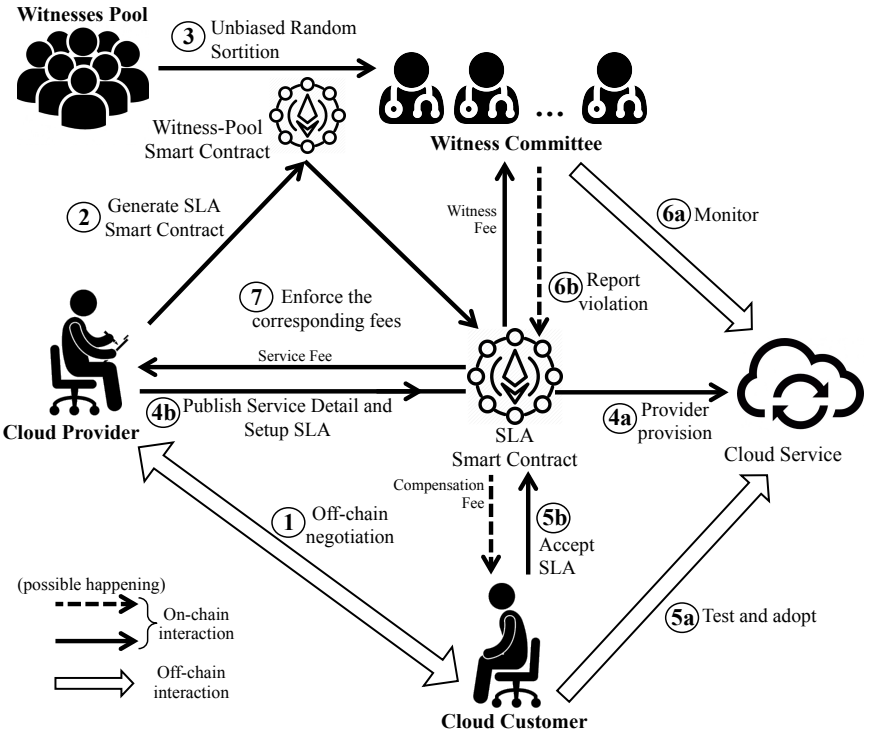


Figure 7.1: System architecture for trustworthy enforcement of Cloud SLA with the witness role

credible the violation detection results are. On the other hand, however, the more witness fee would be paid, and both the customer and the provider need to afford this fee equally. According to this negotiation, the provider can reset these parameters of the generated SLA contract. Afterwards, a set of N witness members can be selected to form a witness committee through the sortition algorithm in Section 7.3.1, which is implemented by the witness pool smart contract on the blockchain. We design the algorithm to be random and able to convince both, c and p, that most ones in the witness committee are independent and would not belong to the adverse role. After dynamically selecting the witness committee member, only the candidate witness can join the specific SLA contract. Meanwhile, the provider provisions its Cloud service for the customer to use and is able to publish its service details in the SLA contract. The witnesses from the committee, therefore, start to monitor the service. In the case of our problem assumption in Section 7.2.1, the provider p provisions a VM on demand and notify the public address IP_{pub} to all the committee members and customer c through the service detail field of the SLA contract. Therefore, the customer can use the VM, and each witness starts to "ping" the address IP_{pub} constantly. If the violation happens during the service time, i.e., the address IP_{pub} is not accessible, the witness can report this event immediately. However, this is a naive example. In real SLAs of more complex scenarios,

the service monitoring component can be negotiated and provided by the provider and customer. Besides, this component can be delivered in the form of containers, which is lightweight and portable. Then, the witness is able to firstly download the container and query the container to detect service violation. Section 7.2.3 describes how the violation state can be finally determined through these witnesses' reports. If the violation event is approved, then the customer can get back its compensation fee. All the dash lines in Figure 7.1 mean it may happen according to the actual event. Anyhow, the provider and witnesses from the committee are able to get corresponding fees.

7.2.3 Service Violation Detection and Reporting

The critical role in our SLA enforcement system is the witness. It takes the responsibility of monitoring the service and determining whether there is a violation. In this section, we present our smart contract model in a specific SLA lifecycle to demonstrate crucial functionalities of a witness: service violation detection and report.

The sequence diagram in Figure 7.2 shows how different roles interact with the smart contract, especially involving the witness in our model. After witnesses being selected, the entire lifecycle of a specific SLA begins. The provider p provisions the Cloud service and deploy the smart contract on the blockchain. In order to set up an SLA, p must prepay the corresponding fee $PF_{prepaid}$ to the smart contract first. The amount of $PF_{prepaid}$ is determined by half of the maximum witness fee. The customer c is then notified about the service and the content of the smart contract. As all the

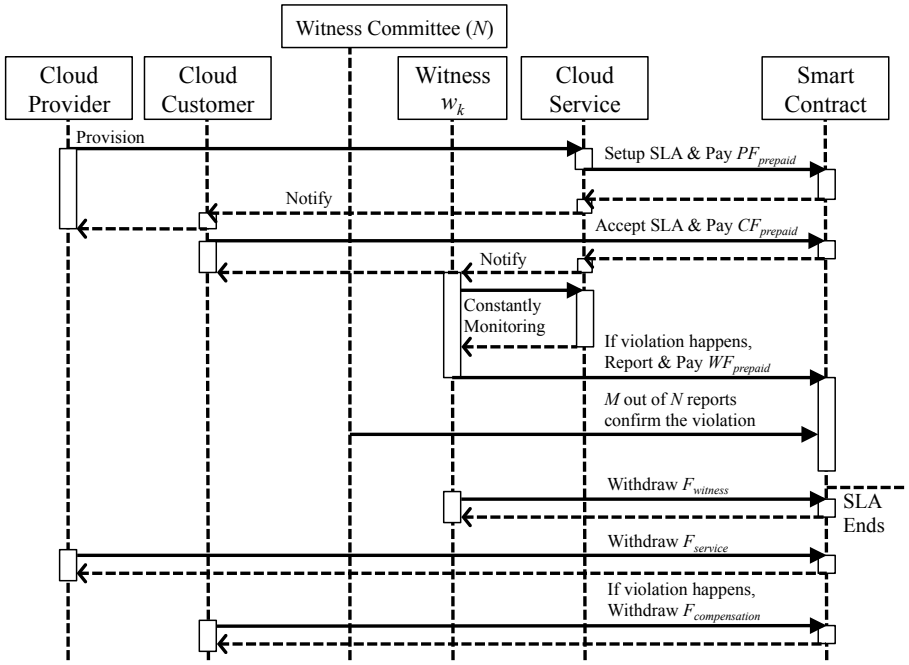


Figure 7.2: Sequential diagram of different roles in the witness model

smart contract on the blockchain is public, the customer can verify the contract and the service status to decide whether to accept the SLA in a particular time window. In order to accept the SLA, the customer also needs to transfer the prepaid fee, $CF_{prepaid}$. It includes the service fee, $F_{service}$ and the other half of the maximum witness fee. As we assume $F_{service} > F_{compensation}$, the compensation fee would be directly deducted from this part of the prepaid fee, if the violation happens. Afterwards, every witness in the committee is notified to start monitoring the service continuously.

During the service time, the witness can decide whether to report the event to the smart contract, if there is an service violation that, for instance, the VM is not accessible. We design the rule that the witness w_k also needs to transfer a small amount of fee, $WF_{prepaid}$, to endorse its report at the same time. The incentive persuading w_k to report the event is that it would gain relative more rewards as a witness fee if the violation event is finally confirmed by the smart contract. On the contrary, if the violation is not confirmed, w_k would not get back the prepaid endorsement fee, $WF_{prepaid}$, as a penalty. This design prevents w_k from reporting fake violations just for maximising its rewards. Moreover, the violation is finally confirmed by the smart contract as the explanation below.

Since the first violation report, the smart contract would start counting a time window, T_{report}. Within this time window, the smart contract accepts reports from other witnesses. When the time window T_{report} is over, the violation is automatically confirmed, if there are no less than M out of N reports from the witness committee received by the smart contract. M is also negotiated by p and c. It is then defined in the smart contract. Of course, M must be bigger than half of N. Furthermore, the bigger the M is, the more trustworthy the violation is. For example, if there are $N = 3$ witnesses in the committee, the service violation can only be confirmed when at least $M = 2$ of them report the event. Here, it is worth to mention that the smart contract is designed to receive the report only from the committee member. Besides, the second report from the same witness is refused within the same report time window. In some sense, these N independent witnesses constitute a $n\text{-}player$ game, in which each witness would like to maximise its rewards. We specially design the payoff function, shown in Section 7.3.2, and leverage the Nash Equilibrium principle of game theory to prove that the witness has to be an honest player in this game. That is, they have to report the violation according to the real event.

Finally, the SLA ends in two cases. One case is the service time $T_{service}$ is over, and there is no violation. The other case is that the SLA is violated. According to these different cases, the three roles can withdraw corresponding fees from the smart contract. Section 7.3.2 explains more details.

7.3 Key Techniques to Ensure Trustworthiness

In this section, we describe key techniques adopted in our witness model in detail. This model enables the automatic detection on the service violation, the results of which can convince both sides: the provider and customer. First, the unbiased random sortition algorithm is leveraged to guarantee that most of the witnesses selected into the committee are random and independent. It is also essential to make both sides achieve a

consensus that most of the selected witnesses would not delegate the opponent's benefit. Based on this, we give the payoff function for the witness model in Section 7.2.3. Also, through the Nash Equilibrium principle, we prove that the "player" from the witness committee have to behave honestly and tell the truth to maximise its rewards. Furthermore, we analyse some possible fraudulent behaviours from a malicious witness and propose quantitative indications to audit them from the action history.

7.3.1 Unbiased Random Sortition

It is crucial in the witness model that the witness sortition for a specific SLA contract is unbiased, i.e., neither the provider nor the customer can have advantages in the committee selection. Since Ethereum has already been introduced as a trusted party in our model, we propose a straightforward random sortition algorithm for committee selection shown as Algorithm 3, which is also implemented in the witness-pool smart contract. It is different from another sortition algorithm developed by us, whose randomness comes from participants [131].

The idea of this sortition process is to exploit the randomness of the blockchain itself. Figure 7.3 illustrates the procedure with three steps. In essence, the basic smart contract, termed as the "Witness-Pool Smart Contract", is to manage the witness pool. It affords interfaces for any blockchain user to register into the pool. Especially, the registered witness is able to turn its state to "Online" or "Offline", in order to indicate whether it can be selected. The detailed witness state management is shown in Section 7.4.2. The addresses in the witness pool are managed as a list in the registration order. Moreover, the sortition algorithm is implemented in the witness-pool smart contract.

As shown in Figure 7.3, there are two interfaces designed in the smart contract to select N witnesses from the pool. The "request" interface is firstly invoked by a specific SLA smart contract at block B_b. It means this transaction is involved in the b^{th} index of block. The hash value of this block is B_b^{hash}. After K blocks generated by the

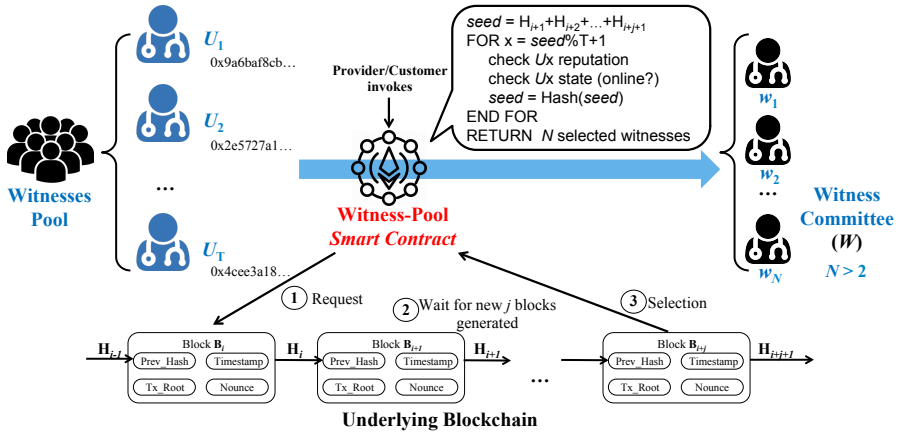


Figure 7.3: The procedure of invoking the unbiased sortition algorithm implemented in the witness-pool smart contract to select the witness committee from the witness pool

Algorithm 3 Unbiased random sortition

Input:
Registered witness set, RW, a list of addresses;
The the size of the list, $len(RW)$;
The number of online witnesses, oc;
Required number, N, of members in a witness committee;
The hash value, B_b^{hash}, of the b^{th} block B_b at request;
The block index, Id, of current block;
Following sequential, K_s, blocks;
Confirmation, K_c, blocks;
The address of the provider, $p.address$;
The address of the customer, $c.address$

Output:
Selected witness set, SW, to form a committee.

assert($Id > b + K_s + K_c$) && assert($oc >= 10 * N$)
seed $\leftarrow 0$
for all $i = 0$; $i < K_s$; $i++$ **do**
  seed$+= B_{b+1+i}^{hash}$
$SW \leftarrow \emptyset$
$j \leftarrow 0$
while $j < N$ **do**
  $index \leftarrow$ seed$\%len(RW)$
  if $RW[index].state == Online$
  && $RW[index].reputation > 0$
  && $RW[index].address != c.address$
  && $RW[index].address != p.address$ **then**
    $RW[index].state \leftarrow Candidate$
    $oc--$
    Add $RW[index] \Rightarrow SW$
    $j++$
  seed $\leftarrow hash$(seed)
return SW

blockchain, another interface "sortition" can be invoked to select N online witnesses as candidates. The sortition algorithm is shown as Algorithm 3.

It takes the hash values of the former K_s out of K blocks mentioned above as a seed. In addition, we need to wait other K_c blocks to confirm the adopted former ones, where $K_s + K_c = K$.

- Here, K_s should be chosen such that the probability of some parties sequentially generating K_s blocks is very small.
- K_c needs to be chosen such that the candidate blocks before are finally involved in the main chain with a dominant probability.
- These two values depend on the blockchain's own properties. Considering the

main net of Ethereum, Efe et al., [44] shows that the top four miners control 61% of the mining power. Thus, we recommend $K_s = 10$ so that with more than 99% probability that the seed is not manipulable and predictable even if the top four miners collude. On the other hand, it is commonly believed that K_c should be 12.

Only the "Online" witness with positive reputation can be selected by the seed from the list of the witness address pool. Anyhow, a new seed is generated based on the hash value of the previous seed. This process is repeated until the required N witnesses are selected. At the beginning of Algorithm 3, there are two assertions. We first check whether there has already been the expected number of blocks generated after invoking the "request" interface, to ensure the unbiasedness of the random seed from the hash values of these blocks. The other assertion is to make sure that there are at least 10 times of available witnesses than required N. It ensures that the number of online witnesses, i.e., the potential ones can be selected into the witness committee, is large enough to achieve the randomness among the committee members and prevent collusion. The invoker can only wait for the condition to be met if any of the assertions are violated.

It is also worth mentioning that oc is leveraged to keep recording how many witnesses are in the "Online" state currently. Shown as Figure 7.5, when the witness is selected by the sortition algorithm, and its state turns into "Candidate", demonstrated as Line 10 of Algorithm 3. The online witness number, oc, should be counted down in Line 11.

Considering the difficulty itself of generating a hash value for a block and combining the sequential K_s blocks as seed, we can prove that the sortition algorithm is random and unbiased, i.e., neither the provider nor the customer can take advantage in the committee with the assumption of trusting the security of Ethereum.

Finally, we analyse the security and trust issues in this design. First, the witness pool should be Sybil-attack-proof. This protection can be achieved by requesting a certain amount of deposit to pay to the witness-pool smart contract when the blockchain participant registers as a witness. Therefore, an entity cannot register a lot of fake witness accounts because that requires a large amount of deposit in total. Second, there is no protection against corruption after the committee candidates are determined. The provider or customer can attack the unwanted candidates to prevent them from joining the committee or bribe them when they are in. However, we argue that this kind of attack or collusion among the provider, customer, and committee members is not easy: 1) the address to identify a witness is the blockchain wallet address in Ethereum, which is just a dynamically user-generated public key. It is not linked to any real identity or IP address. Therefore, using off-chain methods, it is even difficult for the witness to convince other committee members that it is also one of the members; 2) any suspicious behaviour can be audited, because all the interactions on the blockchain are public and immutable, e.g., the possible bribery. Then, the witness's reputation value described in Section 7.4.2 can be influenced through auditing. Hence, the on-chain collusion is easy to detect and audit; and 3) when the witness pool is big enough, the unbiased sortition algorithm proposed in this section is able to ensure that most of the committee members are not known with each other before and the members change every time, since no one can determine the sortition result. Therefore, the cost for collusion is high, and the trade-off prompts the witness to perform honestly for rewards.

7.3.2 Payoff Function and Nash Equilibrium

Game theory targets to mathematically predict and capture behaviour in a strategic situation, where each player's rewards depend on the strategies of itself and also others. There is currently a wide range of applications, including economics, evolutionary biology, computer science, political science, philosophy [17] and also SLA negotiation [127].

The strategic or matrix form, of a $n\text{-}player$ game, is the most common representation of strategic interactions in game theory. The definition consists of a set of players, a set of strategy profiles and a design of payoff functions. Based on the basic type of strategic form game with complete information in game theory, we define our witness game as follow.

Definition 1. Witness Game: it is a $n\text{-}player$ game represented as a triple (SW, Σ, Π), where

- $SW = \{w_1, w_2, ..., w_n\}$ is a set of n players. Each player is a selected witness and they form the witness committee.
- $\Sigma = \Sigma_1 \times \Sigma_2 \times ... \times \Sigma_n$ is a set of strategy profiles, where Σ_k is a set of actions for the witness w_k, i.e., w_k can choose any action $\sigma_k \in \Sigma_k$. A strategy profile is, therefore, a vector, $\sigma^* = (\sigma_1^*, \sigma_2^*, ..., \sigma_n^*)$, where σ_k^* is a specific action of Σ_k, $(k = 1, 2, .., n)$.
- $\Pi = \{\pi_1, \pi_2, ..., \pi_n\}$ is a set of payoff functions, where $\pi_k : \Sigma \rightarrow R$ is the payoff function determining the rewards for witness w_k under a certain strategy, $(k = 1, 2, .., n)$. R is the corresponding rewards.

In addition, $\sigma_{-k} = \{\sigma_1, \sigma_2, ..., \sigma_{k-1}, \sigma_{k+1}, ..., \sigma_n\}$ is defined as any strategy profile σ without player k's action. The full strategy can then be written as $\sigma = \{\sigma_k, \sigma_{-k}\}$. Actually, there are only two actions in our witness game, which is $\Sigma_k = \{\sigma_k^{(r)}, \sigma_k^{(s)}\}$. $\sigma_k^{(r)}$ means Report the service violation to the smart contract. $\sigma_k^{(s)}$ means do not report and keep Silence to the smart contract. In this N-witness game, we define the set of witnesses choosing the action of Report as, W_{report}, where $\forall w_k \in W_{report}$, the $\sigma_k^* = \sigma_k^{(r)}$. Respectively, $W_{silence}$ is the set of witnesses not reporting, where $\forall w_k \in W_{silence}$, the $\sigma_k^* = \sigma_k^{(s)}$. These actions determine the final state of SLA: $SLA_{status} = V$iolated, there is a service violation; $SLA_{status} = C$ompleted, service is completed without violation. We then define the violation confirmation as Definition 2.

Definition 2. Violation Confirmation: based on the result of a strategy profile in a N-witness game, the service violation is confirmed, only when $||W_{report}|| \geq M$, where $1 < N/2 < M < N, N, M \in \mathbb{N}$. Otherwise, it is treated as there is no violation happened.

It is worth mentioning that we define $N > 2$ and $M < N$ here, in order to achieve the violation confirmation reliably and fairly. According to our witness model, the witness is designed to report the violation along with endorsement fee to the SLA smart contract. Therefore, if the violation is not confirmed, the witness cannot retrieve back its endorsement fee as a penalty. The detailed payoff function design is shown

as Definition 3 according to above definitions and analysis. Here, the value of each function is only leveraged to quantitatively represent the relative relationship among the fees. Hence, 1 represents one share of profit. 10 is ten times shares of 1. -1 means losing one share of profit.

Definition 3. Payoff Functions: the values of payoff functions are designed according to the final SLA status.

- When $SLA_{status} = V$iolated:
  $\forall w_k \in W_{report}, \pi_k(\sigma_k^{(r)}, \sigma_{-k}) = 10;$
  $\forall w_k \in W_{silence}, \pi_k(\sigma_k^{(s)}, \sigma_{-k}) = 0.$
- When $SLA_{status} = C$ompleted:
  $\forall w_k \in W_{report}, \pi_k(\sigma_k^{(r)}, \sigma_{-k}) = -1;$
  $\forall w_k \in W_{silence}, \pi_k(\sigma_k^{(s)}, \sigma_{-k}) = 1$

In a $n\text{-}player$ game, if a player knows the others' actions, it would choose a strategy to maximise its payoff. This is referred as its best response. Therefore, the best response of the witness w_k can be defined as follows.

Definition 4. Witness w_k's best response: in order to maximise its rewards, w_k's best response to a strategy profile σ_{-k}^* is a strategy $\sigma_k^* \in \Sigma_k$, such that $\pi_k(\sigma_k^*, \sigma_{-k}^*) \geq \pi_k(\sigma_k, \sigma_{-k}^*)$ for $\forall \sigma_k \in \Sigma_k (k = 1, 2, ..., n)$.

A Nash Equilibrium point [87] is, therefore, able to be defined as a stable state, where no player has an incentive to deviate from current strategy. It is actually a strategy under which every player adopts its own best response.

Definition 5. Nash Equilibrium point: it is a specific strategy profile $\sigma^* = (\sigma_k^*, \sigma_{-k}^*)$, if for every witness w_k, σ_k^* is a best response to σ_{-k}^*, i.e., $\forall w_k \in SW$ and $\forall \sigma_k \in \Sigma_k (k = 1, 2, ..., n)$, $\pi_k(\sigma_k^*, \sigma_{-k}^*) \geq \pi_k(\sigma_k, \sigma_{-k}^*)$.

Based on the Nash Equilibrium point definition and payoff functions, we can derive the theorem below.

Theorem 1. In a witness game, the only two Nash Equilibrium points are following strategy profiles:

- $\sigma^* = (\sigma_1^*, \sigma_2^*, ..., \sigma_n^*)$, of which $\forall w_k \in SW$, $\sigma_k^* = \sigma_k^{(r)}$;
- $\sigma^* = (\sigma_1^*, \sigma_2^*, ..., \sigma_n^*)$, of which $\forall w_k \in SW$, $\sigma_k^* = \sigma_k^{(s)}$.

Proof. According to Definition 1 and 2, in a N-witness game, $N \geq 3$, $N/2 < M \leq N - 1$ and $M \geq 2$, where $N, M \in \mathbb{N}$.

For the strategy profile of $\forall w_k \in SW$, $\sigma_k^* = \sigma_k^{(r)}$, which means $||W_{report}|| = N > M$. The SLA violation status is, therefore, violated, $SLA_{status} = V$. According to payoff functions design in Definition 3, for $\forall w_k$, its rewards are $\pi_k(\sigma_k^{(r)}, \sigma_{-k}^*) = 10$. If any w_k chooses the other action, Silence, instead of Report. The final status of SLA, however, would not be modified, due to $||W_{report}|| = N - 1 \geq M$. Then, w_k's rewards

are $\pi_k(\sigma_k^{(s)}, \sigma_{-k}^*) = 0 < 10 = \pi_k(\sigma_k^{(r)}, \sigma_{-k}^*)$. According to Definition 5, this strategy profile is a Nash Equilibrium point.

Analogously, for the strategy profile of $\forall w_k \in SW$, $\sigma_k^* = \sigma_k^{(s)}$, which means $||W_{report}|| = 0 < 2 \leq M$. The SLA violation status is, therefore, not violated, $SLA_{status} = C$. According to payoff functions design in Definition 3, for $\forall w_k$, its rewards are $\pi_k(\sigma_k^{(s)}, \sigma_{-k}^*) = 1$. If any w_k chooses the other action, *R*eport, instead of *S*ilence. The final status of SLA, however, would not be modified, due to $||W_{report}|| = 1 < 2 \leq M$. Then, w_k's rewards are $\pi_k(\sigma_k^{(r)}, \sigma_{-k}^*) = -1 < 1 = \pi_k(\sigma_k^{(s)}, \sigma_{-k}^*)$. According to Definition 5, this strategy profile is also a Nash Equilibrium point.

For all the other strategy profiles, they are all a mix of actions, *R*eport and *S*ilence. It means $W_{report} \neq \emptyset$ and $W_{silence} \neq \emptyset$. When $SLA_{status} = V$, i.e., $||W_{report}|| \geq M$, there always $\exists w_k \in W_{silence}$, it can change the action to *R*eport. But the SLA status would not change, because of $||W_{report}|| + 1 > M$. Hence, w_k increases its rewards, from $\pi_k(\sigma_k^{(s)}, \sigma_{-k}^*) = 0$ to $\pi_k(\sigma_k^{(r)}, \sigma_{-k}^*) = 10$. On the other hand, when $SLA_{status} = C$, i.e., $||W_{report}|| < M$, there always $\exists w_k \in W_{report}$, it can change the action to *S*ilence. But the SLA status would not change, because of $||W_{report}|| - 1 < M$. Hence, w_k increases its rewards, from $\pi_k(\sigma_k^{(r)}, \sigma_{-k}^*) = -1$ to $\pi_k(\sigma_k^{(s)}, \sigma_{-k}^*) = 1$. These counterexamples demonstrate all these strategy profiles are not Nash Equilibrium points.

Therefore, in a witness game, there are two and only two Nash Equilibrium points. They are $\sigma^* = (\sigma_1^{(r)}, \sigma_2^{(r)}, ..., \sigma_n^{(r)})$ and $\sigma^* = (\sigma_1^{(s)}, \sigma_2^{(s)}, ..., \sigma_n^{(s)})$. □

We take the basic $three$-witness game as an example, where $N = 3$. Therefore, M can only be equal to 2 based on Definition 2. Table 7.1 shows payoff functions according to our previous definitions. The value element in Table 7.1 is the vector of corresponding payoff function values. It is represented as (π_1, π_2, π_3). According to Theorem 1, Nash Equilibrium points in this game are (10, 10, 10) and (1, 1, 1), respectively.

Based on above analysis, for a rational and selfish witness, who wants to maximise its rewards through offering services, would have to behave as follows in this game. If there is a violation happening, the witness knows that most of other witnesses are more likely to report this event to gain more rewards. Hence, the higher rewards push the witness to report this event. On the contrary, if there is no violation, the witness knows that most of other witnesses are more likely to keep silence. Although the witness wants to achieve the highest rewards, it has to take a great risk to pay a penalty for its

Table 7.1: Payoff functions for a $three$-witness game

w_1	w_3			
	$\sigma_3^{(r)}$: *R*eport		$\sigma_3^{(s)}$: *S*ilence	
	w_2		w_2	
	$\sigma_2^{(r)}$: *R*eport	$\sigma_2^{(s)}$: *S*ilence	$\sigma_2^{(r)}$: *R*eport	$\sigma_2^{(s)}$: *S*ilence
$\sigma_1^{(r)}$: *R*eport	(10, 10, 10)	(10, 0, 10)	(10, 10, 0)	(-1, 1, 1)
$\sigma_1^{(s)}$: *S*ilence	(0, 10, 10)	(1, 1, -1)	(1, -1, 1)	(1, 1, 1)

fraudulent behaviour. From the global view, when there is no violation, all the witnesses prefer to keep silence in order to stay at the Nash Equilibrium point, $(\sigma_1^{(s)}, \sigma_2^{(s)}, ..., \sigma_n^{(s)})$. Then the violation acts as a signal to push them achieving another Nash Equilibrium point, $(\sigma_1^{(r)}, \sigma_2^{(r)}, ..., \sigma_n^{(r)})$, for much higher rewards. At the same time, they tell the truth about the service violation.

Therefore, making the witness tell the truth of the event is not because of honesty. Instead, the reason is that the witness only wants to maximise its rewards.

7.3.3 Witness Audit

The sortition algorithm, Algorithm 3, ensures that the selected witnesses are independent to a great extent. The payoff function design stimulates the witness telling the truth. However, an auditing mechanism is still needed to ensure that the malicious or irrational witness can be detected and kicked out from the witness pool. As all the interactions with the smart contract, i.e., transactions, are public and permanently stored on the blockchain, it is possible to audit a witness through its behaviour history. We mainly exploit the following information in the history to do auditing. It is expressed as Equation 7.1. It represents a set of events of the witness, w_k. The witness can adopt the action of silence $\sigma_k^{(s)}$ or the action of report $\sigma_k^{(r)}$. σ_k^* means the SLA event with any action adopted by w_k. To be specific, when the action is report, we can also further know its $order$ among all the witnesses' reports, first to report or not, and the reporting time T_r, relatively from the SLA starting time. In addition, $status$ expresses whether the SLA is violated or completed. Symbol, "$*$", means the event with any SLA status is counted in this set.

$$Event(w_k, \sigma_k^{(s)} || [\sigma_k^{(r)} : (order, T_r)] | \sigma_k^*, status : C|V|*) \tag{7.1}$$

Based on the information above, we analyse that there are three types of malicious witnesses: lazy witness, speculative witness, and sacrificed witness.

Lazy witness refers to the one, who prefers not to report the violation. Since there is a case that the higher incentive for reporting a violation is not enough to motivate the lazy ones, they can choose the strategy not to really monitor the service. Then they always keep silence and never report the violation. With this strategy, they would not pay the penalty, even if the final status of SLA is actually violated. Considering violation is not a regular event, the lazy witness is still able to gain some rewards via multiple "games". However, this type of lazy witness can be audited through the active rate, which is defined as follows.

Definition 6. Active rate ($\eta_{active}(w_k)$): this the metric to measure the activeness of witness w_k, when there is a violation.

$$\eta_{active}(w_k) = \frac{||Event(w_k, \sigma_k^{(r)}, status : V)||}{||Event(w_k, \sigma_k^*, status : V)||} \tag{7.2}$$

$\eta_{active}(w_k) = 0$ means that the witness w_k never reports the violation event, although there actually is one. A threshold, $\widehat{\eta}_{active}(w_k)$, therefore, can be set to

determine whether w_k is a lazy witness, i.e., still $\eta_{active}(w_k) < \widehat{\eta}_{active}(w_k)$, when w_k has already been involved into many SLA events.

Speculative witness refers to the one, who is more likely to report in a speculative way. Since all the actions are public and transparent on the blockchain, there is a possible speculative behaviour for the witness, which is only to follow others' reports. The witness does not monitor the service. Instead, it monitors transactions on the blockchain to see whether some other witnesses are reporting the violation. Then, it immediately follows and reports, trying to gain the maximum rewards. Although we can set a relatively short report time window in the model design of Section 7.2.3, its speculative reports might still be counted in the following blocks of the blockchain. Moreover, this speculative witness also needs to take the risk that the violation may not be confirmed finally. Anyhow, this type of speculative witness can be audited through the following rate.

Definition 7. Following rate ($\eta_{follow}(w_k)$): this the metric to measure the frequency that the witness w_k follows the reports of other witnesses.

$$\eta_{follow}(w_k) = \frac{||Event(w_k, \sigma_k^{(r)} : (NotFirst, T_r), status : V)||}{||Event(w_k, \sigma_k^{(r)} : (order, T_r), status : V)||} \tag{7.3}$$

Here, $NotFirst$ means that the transaction containing the report of w_k is not the first block in the reporting time window. So $\eta_{follow}(w_k) = 100\%$ means for all violated SLA events involving the witness w_k, it is never the first one to report. A threshold, $\widehat{\eta}_{follow}(w_k)$, therefore, can be set to determine whether w_k is a speculative witness through following, i.e., when $\eta_{follow}(w_k) > \widehat{\eta}_{follow}(w_k)$, if w_k is involved into many SLA events.

Sacrificed witness refers to the one, who always reports at a specific time stamp. For instance, w_k always reports the violation within one minute after the SLA starts. Though the witness may pay a lot of penalty for its malicious behaviour at the beginning, it can show other witnesses its behaviour pattern from its history later on. In some sense, it is able to imply to others that it would report at some time stamp. Then as long as others have analysed its behaviour pattern and followed, it can most likely gain the maximum rewards. Hence, it is crucial to audit this type of witness through the following fixed pattern rate.

Definition 8. Fixed pattern rate ($\eta_{fix}(w_k)$): this the metric to measure the frequency that the witness w_k report the violation at a specific time stamp, $\widehat{T}_r$.

$$\eta_{fix}(w_k) = \frac{||Event(w_k, \sigma_k^{(r)} : (order, \widehat{T}_r), status : *)||}{||Event(w_k, \sigma_k^{*}, status : *)||} \tag{7.4}$$

Here, $\eta_{fix}(w_k) = 100\%$ means for all of the SLA events, which involves the witness w_k, it always reports at time stamp, $\widehat{T}_r$. A threshold, $\widehat{\eta}_{fix}(w_k)$, therefore, can be set to determine whether w_k is a sacrificed witness through following, i.e., when $\eta_{fix}(w_k) > \widehat{\eta}_{fix}(w_k)$, even if w_k is just involved into several SLA events.

It is worth to mention that these auditing mechanisms can also be implemented in the smart contract, in order to avoid a third party to dominate the judgement. It can be combined with the reputation value of the witness, which is further explained as the witness reputation in the implementation part of Section 7.4.2. Therefore, when the witness' reputation decreases to zero, that witness would also be blocked automatically by the sortition algorithm.

7.4 Prototype Implementation and Experiments

According to the witness model and the payoff function design, we implement a prototype system based on the smart contracts of Ethereum. We leverage the programming language, Solidity[6], provided by Ethereum, to program smart contracts. Overall, there are three roles and two types of smart contracts in our SLA enforcement system. Roles include the traditional Provider and Customer, as well as the introduced Witness. The smart contracts include the witness-pool smart contract and the SLA smart contract. In this section, we firstly illustrate the state transition in that two types of smart contracts, respectively. Via this, we describe the detailed functionalities of the interfaces in the smart contracts and show how they are leveraged to transit the states. Afterwards, we show some experimental studies on the transaction cost of these interfaces on the Ethereum test net, "Rinkeby"[7].

7.4.1 Overall System Implementation

Overall, there are three roles and two types of smart contracts in our SLA enforcement system. Roles include the traditional Provider and Customer, as well as the introduced Witness. The smart contracts include the witness-pool smart contract and the SLA smart contract. Figure 7.4 illustrates the relationship among these entities through different interfaces. These interfaces are named as the text on the arrow. The format of the text is '$R_{role} \rightarrow [C_{type}::]N_{interface}$'. It means that only the role R_{role} can invoke the interface, $N_{interface}$, which is defined in the smart contract of C_{type}. The corresponding implementation is achieved by the checking mechanism, which is the property of the programming language provided by Ethereum. Therefore, the smart contract restricts that only the specified role can interact with the smart contract in a certain state. The representation of the R_{role} are P for Provider, C for Customer, SC for a generated SLA Smart Contract and X for any blockchain user. Also, for C_{type}, WP is for the witness-pool type of smart contract, and SLA is for the generated ones for SLA enforcement. In addition, this interface definition also applies to Figure 7.5 and Figure 7.6.

The witness-pool smart contract is the basis in order to set up the system. Any blockchain user can register and become the Witness role through the interface "register" provided by the witness-pool smart contract. There are also some other interfaces for the witness to invoke to change its state in the pool in order to be selected. More details are discussed in Section 7.4.2. For a specific SLA lifecycle, any user X can invoke the

[6]http://solidity.readthedocs.io
[7]https://www.rinkeby.io/

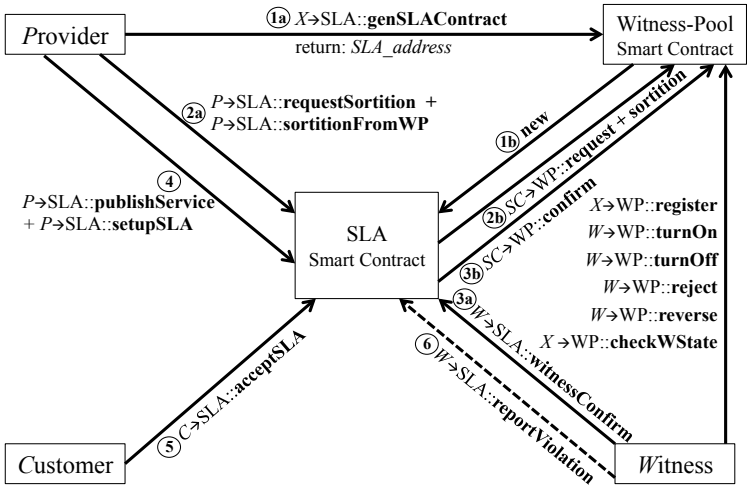


Figure 7.4: Interactions among roles and smart contracts

interface "genSLAContract" provided by the witness-pool smart contract. Afterwards, a specific SLA smart contract is generated by the witness-pool smart contract, and the contract address is returned back. This address is also recorded in the witness-pool smart contract. It ensures the validity of the SLA smart contract to interact with other roles. Meanwhile, the user X becomes the Provider role of the generated SLA smart contract. It can customise the contract, including setting the customer's address, and other negotiated contract parameters, such as service duration and witness committee scale N. It is also responsible for performing the unbiased random sortition algorithm, explained in Section 7.3.1. As this sortition algorithm is leveraged by the provider through a specific SLA contract, the online witness can know it is selected by which SLA smart contract, basically the contract address, from checking its own state. Then it can make a confirmation to the SLA contract to join the witness committee. Simultaneously, the SLA smart contract further invokes interfaces of witness-pool smart contract to acknowledge the witness management. After a proper witness committee is constructed, the provider can publish its service detail on the chain to initiate the SLA lifecycle. Details are explained in Section 7.4.3.

7.4.2 Witness-pool Smart Contract Implementation

In this part, we focus on implementation details about the witness-pool smart contract, especially the witness management. Figure 7.5 illustrates the states of a witness role defined in the smart contract. It includes four states: "Online", "Offline", "Candidate" and "Busy". The state transition of the witness is as follows.

After registration in the witness pool, only the witness itself can turn its state into "Online". It is then probably selected by a specific SLA smart contract. Hence, the witness needs to monitor its own state on the blockchain continuously. This operation is feasible, as the read-only operation does not need any transaction fee. Once it is

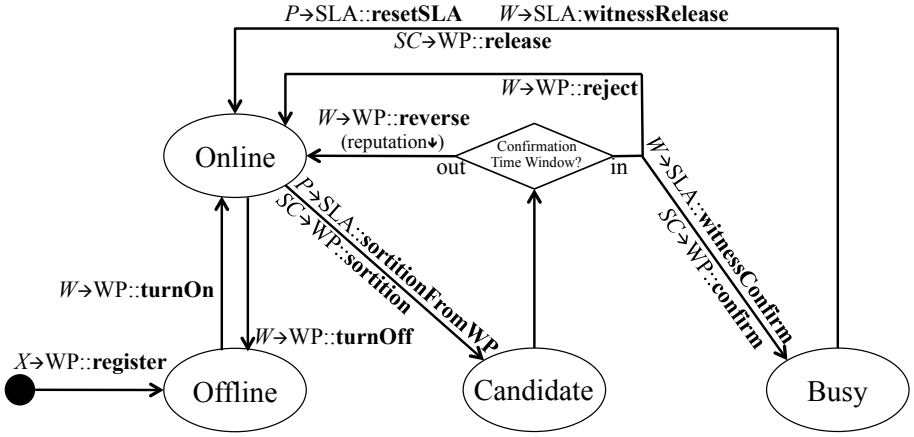


Figure 7.5: State transition diagram of a witness in the witness-pool smart contract

selected by a specific smart contract through performing "sortition" algorithm, its state turns into "Candidate". Within a confirmation time window, e.g., 2 minutes, the witness can look through the SLA smart contract, which selects it, and decide whether to confirm or reject this sortition. If it rejects, the provider of the SLA smart contract has to perform another sortition. Otherwise, its state turns into "Busy", after it invokes the interface, "witnessConfirm", of the SLA smart contract. By the end of each SLA lifecycle iteration, the witness has the right to actively leave the SLA contract by leveraging the interface "witnessRelease". On the other hand, it can also be passively released from the SLA contract if the provider invokes the interface "resetSLA" to dismiss the witness committee. Finally, the witness can "trunOff" to avoid being selected before it is not available to the Internet.

In order to prevent some malicious intentions, we bring in a reputation value for each witness to measure their behaviours. Firstly, each witness has an initial reputation value of R_{init} at registration, which could be a predetermined constant value. Then, for instance, some witness may not turn its state into "Offline", when it is not actually available or does not frequently check its state. Then, it would not be able to confirm the selection and join the SLA contract within the confirmation time window, if it is chosen. In this case, the witness would not be chosen again, since its state becomes "Candidate". To reverse back to the state "Online", in which it can be selected, the witness has to leverage the interface, "reverse". In this case, its reputation value decreases by 10. If this value becomes zero or less, it would be permanently blocked by the sortition process according to Algorithm 3. We also combine the reputation value with the auditing mechanism mentioned above in some sense. For example, the reputation of the witness, who does not report, would decrease by 1, when the violation is confirmed. It is the same with the one, who reports the violation but the violation is not finally confirmed.

It is worth mentioning that in the current implementation, we have not designed the scenario where the reputation can be heightened. The idea is to make the reputation decreasing as a soft punishment, not directly losing tokens (money). However, when

the reputation is too low, the account is blocked, and the witness has to register another account. Meanwhile, the witness would lose the deposit of the blocked account in this case. On the other hand, if the witness can heighten its reputation through performing honestly, even the increasing scale is much smaller than the decreasing scale. This design would still give witnesses the chance to balance the reputation through performing different times of honest behaviours and malicious behaviours, e.g., performing maliciously once and ten times of honest behaviours afterwards.

7.4.3 SLA Smart Contract Implementation

Figure 7.6 shows the SLA state transition to implement a specific SLA smart contract enforcement. This type of smart contract is generated by the witness-pool smart contract. All the interfaces annotated in this figure belongs to this type of SLA smart contract. Hence, we omit the definition scope C_{type}::. There are five states: "Fresh", "Init", "Active", "Violated" and "Completed", shown as circle in Figure 7.6. The dashed arrows demonstrate the state transition path when a violation happens. The three squares in the figure represent the respective roles in this smart contract. At the end of SLA, they can withdraw the rewards respectively. The dashed line here also refers to the action adopted under the situation of the violation.

The contract is generated in the state of "Fresh". In this state, the provider can customise the SLA parameters according to the negotiated results with the customer. Furthermore, the service detail can also be published onto the contract through "publishService". In our case, the detail is the public IP of the VM. All others are, therefore, notified. However, this SLA only proceeds when the number of the members in the witness committee is satisfied. Otherwise, the provider is unable to leverage the interface, "setupSLA", to transit into "Init" state. According to the witness model design in Sec-

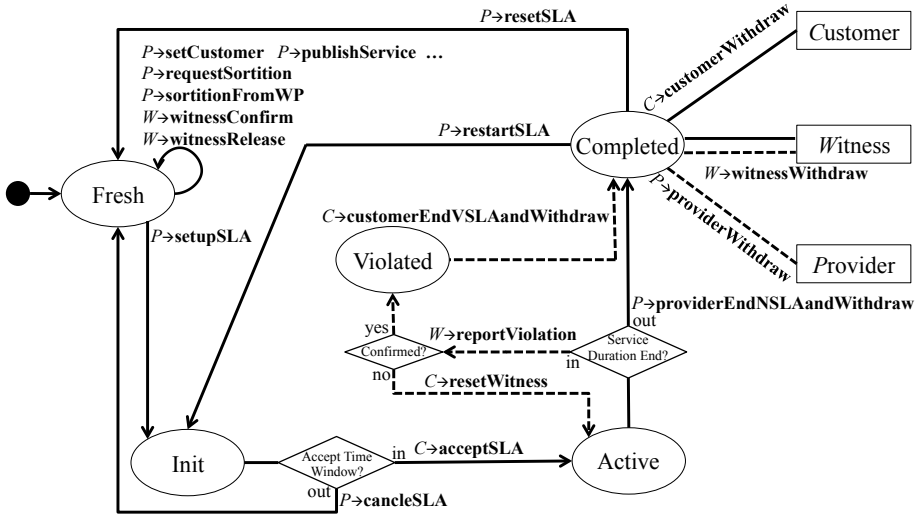


Figure 7.6: State transition diagram of SLA lifecycle for a specific SLA smart contract

tion 7.2.3, the provider needs to prepay some fee, $PF_{prepaid}$, to the smart contract for hiring witnesses. The actual amount of fee is calculated by the smart contract according to the scale of committee member and a basic hiring fee. Also, this amount of tokens is one of the requirements to invoke the interface. It ensures that only that amount of prepaid fee is transferred into the smart contract. The customer then decides whether to accept the SLA. If it accepts the SLA, it also needs to prepay the fee, $CF_{prepaid}$, including the service fee and its part of the hiring fee for witnesses. If not, the provider can "cancleSLA" and withdraw back its money. When the service is completed, all corresponding roles can retrieve their rewards through a set of withdrawing interfaces. After all the money is withdrawn from the contract, the provider can leverage "resetSLA" or "restartSLA" to rotate back to the previous state. These interfaces are designed for continuous service delivery instead of a long service duration to trap witnesses. The difference between these two interfaces is "resetSLA" dismisses the witness committee and the provider can change some other terms. On the other hand, "restartSLA" would keep the committee and quickly proceed another round of SLA iteration.

It is also worth to mention that the smart contract on blockchain cannot run itself. The state transition must be triggered by some interfaces and it takes some cost to execute. Therefore, we design the interface for the role, who is the greatest beneficiary in some cases, to modify the state. Because they have the motivation to perform the state transition. For example, when the service duration ends normally, the provider is the greatest beneficiary to gain the entire service fee. It must actively leverage the interface, "providerEndNSLAandWithdraw", to end the normal SLA and withdraw its own rewards. Meanwhile, it divides the prepaid money as the payoff function design in Section 7.3.2 to different witnesses. Afterwards, other roles are able to withdraw their part of rewards. Analogously, when there is a violation, the customer is the most motivated one to gain the compensation fee. It can leverage, "customerEndVSLAandWithdraw", to end the violated SLA and transit the state from "Violated" to "Completed". It is the same for "resetWitness", which is the customer to reset the witnesses' state and report time window, when the violation is not confirmed. Otherwise, witnesses cannot report the violation later on, if there are real ones.

7.4.4 Experimental Study

In order to test all the functionalities of our model and system design, we deploy the implemented smart contracts on the test net of Ethereum blockchain, "Rinkeby". It is a world-wide blockchain test net for developers to debug the smart contract. The 'Ether', which is the cryptocurrency of Ethereum, does not represent real value on the test net and can be applied for debugging. Hence, we generate several accounts on "Rinkeby" to simulate different roles, i.e., the provider, customer, and witnesses. We leverage the retrieved 'Ether' on each simulated account to execute the interfaces and prepay different types of fees according to the model. To conduct the experiment, we first deploy the basic witness-pool smart contract and make all the accounts registered to the witness pool. The provider then generates an SLA smart contract to start the SLA lifecycle with the customer. Afterwards, we test all possible scenarios to exploit and validate the functionality of different interfaces. The results demonstrate that our system implementation satisfies our model and payoff function design.

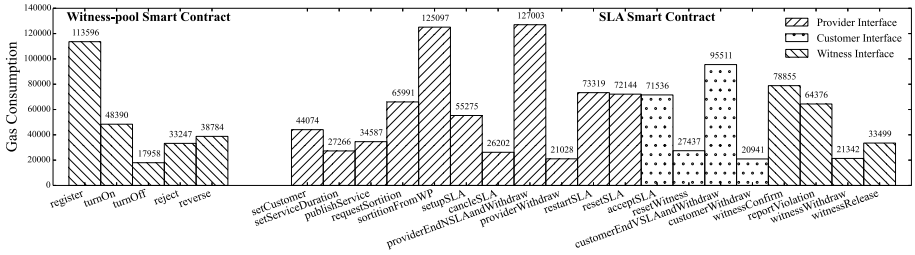


Figure 7.7: The gas consumption of each interface in the smart contracts

The trust part of the system is proved by game theory and ensured by the unbiased sortition algorithm, whose credibility is endorsed by the blockchain technique. Therefore, we mainly analyse some performance information from our experimental study. Here, the performance refers to the complexity of each interface in the smart contract. It determines the transaction fee needed to pay the miner in Ethereum since the miner needs to execute the program defined in the interface, which consumes the electricity power of the miner. The more complex of the interface is the more transaction fee required when it is invoked. The transaction fee is measured as 'Gas' defined in Ethereum, which is a unit that refers to how much work taken by the miner when executing the transaction. The final fee is the product of gas amount and the gas price for each unit. Hence, the gas consumption is similar no matter on the test net or main net. We, therefore, record all the gas consumption for each interface from the transaction history of the experiment.

Figure 7.7 illustrates the gas consumption of each interface. We show the major interfaces defined in the two types of smart contracts, which construct the system. Some simple interfaces, such as the one that sets the SLA parameters, are omitted. Their gas consumptions are similar to 'setServiceDuration' in the figure. Some other interfaces of the witness-pool smart contract are also omitted, as these are the ones that can only be invoked by the SLA smart contract. Its consumption is involved in some interface in the right part of the figure. Besides, the consumption of 'genSLAContract' in the witness-pool smart contract is also not shown in the figure, because it is more than 10 times higher than others, which is around 2,200,958. However, it is acceptable for the provider to invoke this to generate a new SLA smart contract, especially the SLA smart contract can be reusable.

From the experimental study, it can be derived that, compared with the customer and the witness, the provider tends to require more gas in the entire SLA lifecycle. The interfaces of customer and witness consume less. The consumption of different roles fits our model design and reality. Because in most cases, the provider earns the most rewards through offering service. It has the incentive to proceed with the lifecycle. The lightweight gas consumption for witness role is also able to convince blockchain users to take part in the system to work as a witness. Moreover, these gas consumption values are achieved through experiments based on the current implementation. There is still possible space to optimise the interface implementation further, in order to lower the gas consumption.

7.5 Conclusion

In this chapter, a witness model is proposed for Cloud SLA enforcement, and we specially design the payoff function for each witness. We leverage the game theory to analyse that the witness has to offer honest monitoring service in order to maximise its own rewards. Finally, a prototype system is fully implemented using smart contracts of Ethereum to realise the witness model. Not only the SLA enforcement lifecycle but also the witness management of the witness pool is implemented with the smart contract. The experimental study demonstrates the feasibility of our model and shows system performance. Via this way, the trust problem is transferred to economic issues. It is not the witnesses themselves would like to be honest, but the economic principles force them to tell the truth. Here, the blockchain plays as a public and immutable monetary management platform according to some predefined rules. We also believe the witness model can be applied in other scenarios with blockchain, where originally only two roles are involved in a contract.

For future work, there are mainly two directions: on-chain and off-chain. For the on-chain part of work, we are going to further optimise the interface implementation to reduce the gas consumption and enrich the functionalities of the smart contract. In addition, some more scenarios should be considered to apply our model. For the off-chain part of work, user-friendly tools are going to be developed for each role in the system to monitor the state on the chain and perform their corresponding interactions. On the other hand, it can also be combined with CloudsStorm framework to construct the witness ecosystem. The vision is to ensure the Cloud performance for applications through automated SLA enforcement.

8 Conclusions

In this thesis, we have presented the CloudsStorm framework for developers of quality-critical Cloud applications to seamlessly program and control virtual infrastructure in the DevOps lifecycle. The case studies and experimental results obtained from real Clouds demonstrate the feasibility and functionality. The blockchain based witness model further enhances the application Quality of Service (QoS) assurance through the trustworthy enforcement of Service Level Agreement (SLA).

To be specific: 1) in Chapter 3, we have designed three levels of infrastructure programmability to describe the infrastructure topology, including network connections. Innovatively, the infrastructure operations, such as failure recovery and scaling, are also defined for programming; 2) in Chapter 4, we have proposed two types of overlay network mechanisms to connect resources from separated data centres. The infrastructure is, therefore, able to be partitioned and accelerate the provisioning process, meanwhile, keep transparency to the applications; 3) in Chapter 5, we have proposed two types of control modes and implemented the "*Infrastructure Execution Engine*" to interpret the programmed infrastructure code. The engine empowers CloudsStorm framework with the ability to request corresponding resources from federated Clouds and control them at runtime; 4) in Chapter 6, we have assumed sufficient scenarios to leverage CloudsStorm to satisfy the application QoS, from the perspectives of task-based and service-based applications; 5) in Chapter 7, we have proposed the blockchain based witness model to credibly detect the service violation for enhancing the final stage of application QoS assurance.

In this chapter, we first summarise CloudsStorm with characteristics to highlight its key innovations, especially demonstrating its features and functional level in the Cloud application DevOps lifecycle. Then we conclude the thesis by answering the research questions proposed in Chapter 1. Finally, we discuss future directions.

8.1 Conclusions of Outcomes

This thesis presents CloudsStorm, a framework for seamlessly programming and operating virtual infrastructure function during the DevOps lifecycle of Cloud applications. We plot the CloudsStorm in the technical landscape of the DevOps tools in Figure 8.1. CloudsStorm works at the VM level. It provides design-level support for describing and customising the federated infrastructure, shown as the vertical pink line

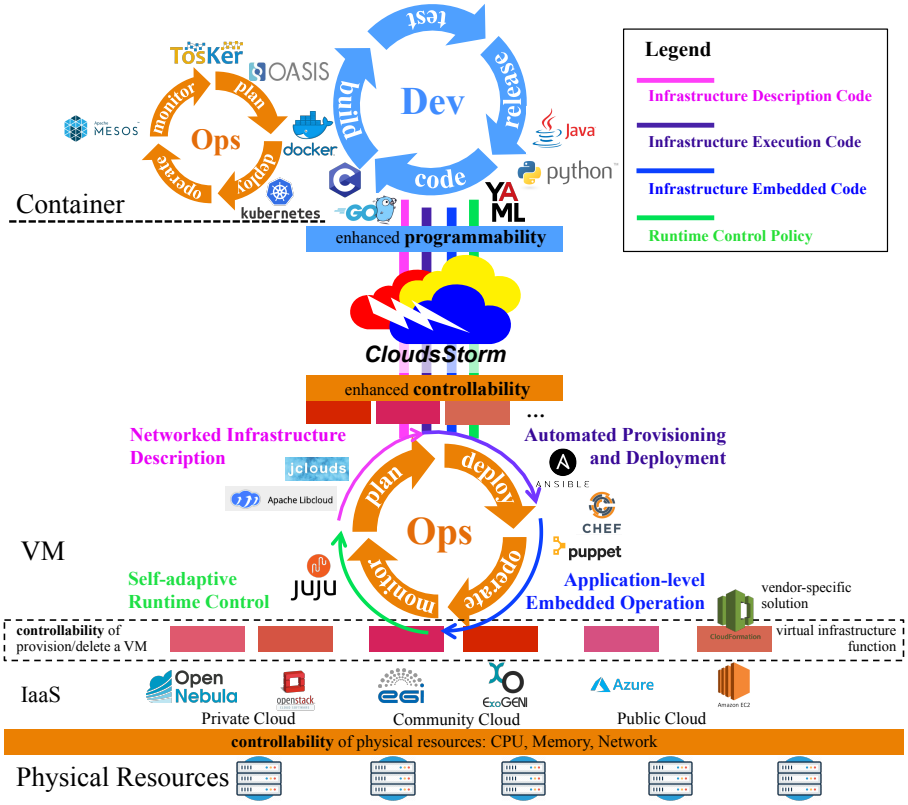


Figure 8.1: The infrastructure programmability and controllability provided by the CloudsStorm framework for DevOps and comparison with related tools

below CloudsStorm in Figure 8.1. Notably, the "*Infrastructure Description Code*" of CloudsStorm enables the ability to describe the networked infrastructure, which makes the infrastructure transparent to the application. Secondly, the "*Infrastructure Execution Code*" is proposed to program operations of the infrastructure-level resources. It corresponds to the vertical purple line below CloudsStorm in Figure 8.1. The infrastructure description code not only provides a static description but also can be executed. Thirdly, the "*Infrastructure Embedded Code*" is proposed to empower the programming on fine-grained infrastructure operations 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 8.1. In addition, we propose two types of control mode, namely passive and active mode, for the application to control its infrastructure. The "*Runtime Control Policy*" is proposed for this purpose, which refers to the vertical green line below CloudsStorm in Figure 8.1. In order to implement the framework, we put forward the network connection method [130] 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 one 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. A dedicated "*Control Agent*" is deployed for each application-defined infrastructure. It, therefore, works in a decentralised way to be more efficient, instead of acting as a centralised infrastructure management framework for many applications, even for many users. Besides, the GUI provided by the "*Control Agent*" affords an intuitive way to check and access Cloud resources. Because of the straightforward YAML format, the syntax of CloudsStorm is easy to learn and program with, especially for the definition of parallel operations.

We demonstrate the CloudsStorm framework using two case studies to migrate task-based applications and service-based applications onto real Clouds. The former one allows a user to program and control infrastructure-level resources using the "*Infrastructure Execution Code*". The latter one concentrates on the application-level programmability by leveraging the "*Infrastructure Embedded Code*" to make the underlying infrastructure being aware of the application-level events for adaption. Both case studies clearly demonstrate that through using CloudsStorm, Cloud applications can be efficient concerning Cloud resource usage when compared against traditional manual or fixed infrastructure configurations. The rich features for programming and controlling virtual infrastructure enable CloudsStorm to support on-demand and parallel infrastructure management. In conclusion, CloudsStorm can enable developers to effectively program a virtual infrastructure for a quality-critical application at the development phase and flexibly control the infrastructure for the application at runtime.

8.2 Conclusions on the Research Objectives

As mentioned in Chapter 1, the main research question to answer in this thesis was identified as:

RQ. How to seamlessly program and control the Cloud virtual infrastructure in the application DevOps lifecycle to satisfy the quality-critical constraints of the application?

Considering the DevOps lifecycle, we conclude that we proposed and implemented a framework, CloudsStorm, to enable seamless infrastructure programming and control in various phases. Therefore, the quality-critical constraints of the application can be better considered and satisfied because of: 1) a systematic infrastructure programmability design in the development phase; 2) fast provisioning and overlay network mechanisms in the provisioning phase; and 3) the seamless controllability implementation in the runtime phase. Moreover, during the entire lifecycle, we leveraged the blockchain based smart contract and proposed a witness model to enhance the trustworthiness of the SLA enforcement. Via this manner, we improved the quality assurance of the application.

To be specific, we conclude this thesis by answering the following detailed research questions.

RQ1. How can we customise and program the infrastructure according to different application quality requirements?

We answer this question by designing infrastructure programmability to address

applications' functional and non-functional requirements, respectively. Considering the functional requirements, we proposed and designed three levels of infrastructure programmability: 1) design-level, i.e., "*Infrastructure Description Code*", enabling the description of infrastructure resources to customise the Clouds and data centres, 2) infrastructure-level, i.e., "*Infrastructure Execution Code*", enabling the description of infrastructure operations to program the operations performed on the infrastructure, and 3) application-level, i.e., "*Infrastructure Embedded Code*", enabling the infrastructure operations to be embedded directly inside the application to achieve fine-grained programmability. Thus, the application can directly manage the infrastructure. Specifically, in order to make the infrastructure operation extensible for different Clouds, we proposed the infrastructure programming model based on basic Cloud VIFs (Virtual Infrastructure Functions). The idea is that the developer only needs to provide several basic Cloud VIFs of a new Cloud to plug into our framework. Then the developer can benefit the programmability of high-level infrastructure operations performed on the new Cloud from our framework. Through analysing the common service characteristics and infrastructure lifecycle of different Clouds, we modelled three basic Cloud VIFs as *VM Provisioning*, *VM Terminating*, and *VM Configuration*. Furthermore, we demonstrated the high-level infrastructure operations, such as scaling and failure recovery, can be constructed by these basic VIFs.

As to non-functional requirements, we designed the "*Runtime Control Policy*" allowing the developer to define the quality-critical constraints of the application through identifying specific monitoring metrics. In addition, the developer can define the operations to adjust the infrastructure if the condition of the monitoring metrics is met. The adjustment is essential to maintain the application's QoS. In particular, we allowed the developer to define the monitoring metrics of the infrastructure system and as well as the application.

RQ2. How can we effectively provision a networked infrastructure and enable topology partitioning across data centres or Cloud providers based on application QoS constraints?

We answer this question by presenting a fast and dynamic provisioning mechanism to accelerate the provisioning process for quality-critical Cloud applications. Before provisioning, we partitioned the original application-defined infrastructure topology into multiple sub-topologies and distributed them into different data centres and even Clouds. Then we proposed two overlay network configuration mechanisms, i.e., tunnel-based and NAT-based, to connect the partitioned sub-topologies transparently. Finally, the partitioned sub-topologies were provisioned concurrently. The advantages of this mechanism are as follows: 1) The automatic overlay network configuration makes the underlying infrastructure transparent to the application. Even the infrastructure is distributed in multiple data centres, the computing resources, i.e., the VMs, can still be connected with the application-defined private network. Then the dependency of the VM provisioning is decoupled, as the public IP address is not required to configure the application in this case. Thus, all the VMs can be provisioned in parallel; 2) The scale of the infrastructure is improved through distributing the resources into various data centres since Cloud providers often impose limitations on the number of VMs that one customer can apply from one data centre. Our mechanism makes it possible provision a large-scale infrastructure across multiple data centres; 3) The provisioning overhead is

reduced, as the time of concurrently provisioning multiple smaller sub-topologies in different data centres is less than that of provisioning the entire infrastructure in one data centre. Specifically, if a particular part of the infrastructure crashes, we only need to re-provision the crashed part, not the entire infrastructure to reduce the recovery overhead. Therefore, the application can benefit from the reduced infrastructure provisioning overhead to satisfy the quality-critical constraints, especially when the application needs to scale out or recover the infrastructure resources in specific time constraints.

Moreover, the provisioning mechanism and the overlay network design have been integrated into CloudsStorm framework. To test the efficiency of the mechanism, we conducted experiments on ExoGENI. The experimental results in practice and model analysis in theory both showed that our mechanism can reduce the provisioning overhead, especially for large-scale infrastructure.

RQ3. How can an application efficiently control the virtual infrastructure at runtime, preferably without vendor lock-in?

We answer this question by proposing a control model with two types of control modes, i.e., the passive mode and the active mode, for distributed virtual infrastructures, which are often across data centres and providers. In the passive mode, the infrastructure is passively controlled and adjusted based on the monitoring information. The threshold of the monitoring metrics to adjust the infrastructure is defined in "*Runtime Control Policy*". However, this passive mode is not sufficient for the application to achieve seamless control on the infrastructure. We, therefore, proposed and implemented another control mode, the active mode. In the active mode, we allows the developer to actively provision and terminate virtual infrastructure resources from scratch: the infrastructure topology is defined in "*Infrastructure Description Code*"; the operations are defined by "*Infrastructure Execution Code*". In addition, the active mode allows the application to actively adjust the infrastructure in advance before getting the monitoring information of a particular event, e.g., the bursty input workload. This type of controllability is achieved by using "*Infrastructure Embedded Code*". As the enhancement of the passive mode, the active mode control is beneficial to adapt the infrastructure seamlessly according to prior knowledge, before the influences have occurred due to the outside events.

Besides, these two control modes have been implemented in "*Infrastructure Execution Engine*" and "*Control Agent*" of CloudsStorm framework. To tackle the vendor lock-in issue, we leveraged specific software design patterns to ensure that the infrastructure programming model based on basic Cloud VIFs proposed in RQ1 can be realised. Thus, our controllability implementation achieves extensibility to support any public IaaS Cloud. Meanwhile, we improved the efficiency of performing infrastructure operations through multi-thread parallelisation. Compared with other tools, i.e., "jcloud" and "cloudinit.d", the scaling and provisioning performance evaluations demonstrate that CloudsStorm can achieve at least 10% efficiency improvement in our experiment settings.

RQ4. How can we effectively handle the SLA with the provider to make the service quality assurance trustworthy?

We answer this question by proposing a witness model using game theory and smart contract techniques. Based on the existing service model of Cloud SLA, we introduced

a new role called "*Witness*" to be responsible for the service violation detection and report. Witnesses gain rewards as an incentive for performing these duties, and the payoff function is carefully designed in a way that trustworthiness is guaranteed: in order to maximise the rewards, the witness has to tell the truth always. This fact that the witness has to be honest was analysed and proved through game theory using the Nash Equilibrium principle. In addition, we proposed an unbiased sortition algorithm to ensure the randomness of the independent witnesses selection from the decentralised witness pool, to avoid possible unfairness or collusion. Finally, to further reduce the risk of the model, we introduced an auditing mechanism to detect potential malicious witnesses. Explicitly, we defined three types of malicious behaviours and proposed quantitative indicators to audit and detect these behaviours.

Moreover, based on the witness model, we have developed a prototype[1] leveraging the smart contracts of Ethereum blockchain. The experiment was conducted on Rinkeby, which is a world-wide blockchain test net for developers to debug the developed smart contracts. Experimental studies demonstrated that the proposed model is feasible, and indicated that the performance, i.e., transaction fee, of each interface follows the design expectations.

8.3 Future Work Directions

The framework can be further extended in future along with the directions of the DevOps using CloudsStorm and the service management using blockchain.

8.3.1 Generalised programming and control models for heterogeneous infrastructures

With the exponential growth of the mobile and Internet of Things (IoT) devices, the emerging 5G technique would eventually boost to realise the situation that everything connects, which generates a mass of data. In this case, computing resources should be anywhere to satisfy the network and computing requirements of the large distributed mobile devices. The current Cloud computing technique is not sufficient due to its relatively centralised resource management in one big data centre. Fog computing [78] is, therefore, the trend to push the infrastructure services, traditionally bounded within big data centres, towards remote nodes or micro-clouds [33] closer to the end devices and data sources. The diversity and heterogeneity of the infrastructure then become a huge hurdle to utilise the resources for orchestrating applications. On the basis of Cloud virtual infrastructure programmability and controllability research in Chapter 3-6 of this thesis, we point out the future work in following two dimensions.

I. Horizontal dimension: distributed and diverse infrastructure management

Considerations on programming the distributed infrastructures, we have empowered the application developers with the ability to describe the Cloud virtual infrastructure

[1] https://github.com/zh9314/SmartContract4SLA

topology, program the operations, and customise the control policies in our CloudsStorm framework. It is specially worth mentioning we have taken care of the openness and extensibility of the framework. In the future work, a more comprehensive description of diverse infrastructure resources should be designed to support programming the Fog/Edge resources provided by some individuals or micro-clouds. Considering the common basic Cloud virtual infrastructure functions modelled in Chapter 3, the infrastructure operations performed on the Fog/Edge resources should also be programmed. On the other hand, with the increasing infrastructure provisioning options, including more small providers and resource geolocations, an algorithm to plan and map the infrastructure topology from the applications' high-level quality-critical requirements should be developed to relieve the developers' programming work.

II. Vertical dimension: heterogeneous infrastructure control

Considerations on controlling the resources in the computing stack, the work is not done after mainly focusing on the controllability of the VM level in this thesis. Within the Fog/Edge environment, the heterogeneity of the infrastructure requires to be considered, because the resources are provided by different individuals or micro-clouds. Meanwhile, traditional public Clouds are also providing heterogeneous infrastructure services, such as Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA). The controllability to handle these specific types of hardware is challenging. Another direction is to provide further programmability and controllability for the container level, which is a more lightweight virtualization than the VM. Meanwhile, Container-as-a-Service [65] is the trend to orchestrate applications and especially in the Fog/Edge environment.

8.3.2 Blockchain Enhanced Infrastructure Service Management

In Chapter 7, we have investigated the possibility of using the blockchain technique to achieve the trustworthy service violation detection between the Cloud provider and customer. But in the IoT environment within the scope of Fog/Edge computing, it becomes more challenging to build trust in managing diverse devices and computing resources of a large scale. The future work can be guided in following two directions.

I. On-chain: fine-grained trust management

The on-chain work refers to the smart contract design and implementation to build a trustworthy system. Based on the witness model proposed in Chapter 7, we are going to further optimise the interface implementation to reduce the gas consumption and enrich the functionalities of the smart contract. Besides, in a complex Fog/Edge computing environment, not only the permissionless blockchain, i.e., the public blockchain, but also the permissioned blockchain [5, 47] should be considered, due to the practical performance issue [118]. The application scenario cannot be limited in service violation detection but other scenarios in DevOps lifecycle, e.g., service publishing and discovery. Furthermore, it is a challenge to harmonise different types of blockchain and balance their advantages and disadvantages [62].

II. Off-chain: open Cloud ecosystem

The off-chain work refers to the user-friendly tool development for users easily interacting with the blockchain. This tool is required by all the blockchain participants for submitting transactions to smart contracts deployed no matter on the permissionless blockchain or the permissioned blockchain. With more comprehensive interacting support through mobile devices, Apps, and browsers, more roles, not only the witness role in Chapter 7, can be introduced to the complicated environment, containing edge devices, fog nodes or micro-cloud providers. Combining the CloudsStorm framework we developed, the blockchain technology needs to be really embedded and exploited in the infrastructure resource management among multiple stakeholders. It is, therefore, a big vision to build a blockchain enhanced open Cloud ecosystem allowing resources freely join and leave, which finally turns the computing into a utility.

Bibliography

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006. (Cited on page 11.)

[2] M. Al-Ayyoub, Y. Jararweh, M. Daraghmeh, and Q. Althebyan. Multi-agent based dynamic resource provisioning and monitoring for cloud computing systems infrastructure. *Cluster Computing*, 18(2): 919–932, 2015. (Cited on page 72.)

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008. (Cited on page 3.)

[4] A. Alamri, W. S. Ansari, M. M. Hassan, M. S. Hossain, A. Alelaiwi, and M. A. Hossain. A survey on sensor-cloud: architecture, applications, and approaches. *International Journal of Distributed Sensor Networks*, 2013, 2013. (Cited on page 97.)

[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018. (Cited on pages 16 and 137.)

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. (Cited on pages 88 and 108.)

[7] I. Baldin, J. Chase, Y. Xin, A. Mandal, P. Ruth, C. Castillo, V. Orlikowski, C. Heermann, and J. Mills. Exogeni: a multi-domain infrastructure-as-a-service testbed. In *The GENI Book*, pages 279–315. Springer, 2016. (Cited on page 54.)

[8] I. Baldine, Y. Xin, A. Mandal, P. Ruth, C. Heerman, and J. Chase. Exogeni: A multi-domain infrastructure-as-a-service testbed. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 97–113. Springer, 2012. (Cited on pages 26 and 54.)

[9] B. Balis, T. Bartynski, M. Bubak, G. Dyk, T. Gubala, and M. Kasztelnik. A development and execution environment for early warning systems for natural disasters. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 575–582. IEEE, 2013. (Cited on page 52.)

[10] N. Bansal, U. Feige, R. Krauthgamer, K. Makarychev, V. Nagarajan, J. Seffi, and R. Schwartz. Min-max graph partitioning and small set expansion. *SIAM Journal on Computing*, 43(2):872–904, 2014. (Cited on page 60.)

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003. (Cited on pages 1 and 11.)

[12] A. Barker and J. Van Hemert. Scientific workflow: a survey and research directions. In *International Conference on Parallel Processing and Applied Mathematics*, pages 746–753. Springer, 2007. (Cited on page 15.)

[13] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015. (Cited on page 17.)

[14] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain't markup language (yaml) version 1.1. *yaml. org, Tech. Rep*, page 23, 2005. (Cited on page 29.)

[15] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014. (Cited on page 54.)

[16] S. Bhardwaj, L. Jain, and S. Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010. (Cited on page 12.)

[17] K. Binmore. *Game theory: a very short introduction*, volume 173. Oxford University Press, 2007. (Cited on page 118.)

[18] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. Tosca: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014. (Cited on pages 4, 19, and 27.)

[19] A. Brogi, L. Rinaldi, and J. Soldani. Tosker: A synergy between tosca and docker for orchestrating multicomponent applications. *Software: Practice and Experience*, 2018. (Cited on page 20.)

[20] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016. (Cited on page 60.)

[21] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*,

2014. (Cited on pages 22, 109, and 110.)

[22] M. Caballer, C. de Alfonso, G. Moltó, E. Romero, I. Blanquer, and A. García. Codecloud: A platform to enable execution of programming models on the clouds. *J. Systems and Software*, 93:187–198, 2014. (Cited on page 18.)

[23] M. Caballer, I. Blanquer, G. Moltó, and C. de Alfonso. Dynamic management of virtual infrastructures. *J. Grid Comput.*, 13(1):53–70, 2015. (Cited on page 18.)

[24] E. Casalicchio and L. Silvestri. An inter-cloud outsourcing model to scale performance, availability and security. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 151–158. IEEE, 2012. (Cited on page 109.)

[25] M. Chen, S. Mao, and Y. Liu. Big data: a survey. *Mobile Networks and Applications*, 19(2):171–209, 2014. (Cited on page 97.)

[26] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016. (Cited on page 110.)

[27] A. V. Dastjerdi, S. K. Garg, O. F. Rana, and R. Buyya. Cloudpick: a framework for qos-aware and ontology-based service deployment across clouds. *Software: Practice and Experience*, 45:197–231, 2015. (Cited on pages 18 and 83.)

[28] M. Dayarathna, Y. Wen, and R. Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):732–794, 2015. (Cited on page 3.)

[29] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. (Cited on pages 16 and 29.)

[30] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 {USENIX} Annual Technical Conference ({USENIX} ATC 15)*, pages 499–510, 2015. (Cited on page 3.)

[31] J. Diaz-Montes, M. AbdelBaky, M. Zou, and M. Parashar. Cometcloud: Enabling software-defined federations for end-to-end application workflows. *IEEE Internet Computing*, 19(1):69–73, 2015. (Cited on pages 16, 18, and 28.)

[32] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 197–200. ACM, 2011. (Cited on page 86.)

[33] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivière. On using micro-clouds to deliver the fog. *IEEE Internet Computing*, 21(2):8–15, 2017. (Cited on page 136.)

[34] S. Ellis, A. Juels, and S. Nazarov. Chainlink: A decentralized oracle network. *white paper*, 2017. (Cited on page 110.)

[35] O. Elzinga, S. Koulouzis, A. Taal, J. Wang, Y. Hu, H. Zhou, P. Martin, C. de Laat, and Z. Zhao. Automatic collector for dynamic cloud performance information. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6. IEEE, 2017. (Cited on page 89.)

[36] F. Faniyi and R. Bahsoon. A systematic review of service level management in the cloud. *ACM Computing Surveys (CSUR)*, 48(3):43, 2016. (Cited on pages 108 and 109.)

[37] F. Farahnakian, T. Pahikkala, P. Liljeberg, J. Plosila, and H. Tenhunen. Utilization prediction aware vm consolidation approach for green cloud computing. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 381–388. IEEE, 2015. (Cited on page 3.)

[38] G. Faure and A. Miquel. *A categorical semantics for the parallel lambda-calculus*. PhD thesis, INRIA, 2009. (Cited on page 35.)

[39] M. Fazio and A. Puliafito. Cloud4sens: a cloud-based architecture for sensor controlling and monitoring. *IEEE Communications Magazine*, 53(3):41–47, 2015. (Cited on page 73.)

[40] J. Fink. Docker: a software as a service, operating system-level virtualization framework. *Code4Lib Journal*, 25:29, 2014. (Cited on page 12.)

[41] M. Fu, L. Zhu, D. Sun, A. Liu, L. Bass, and Q. Lu. Runtime recovery actions selection for sporadic operations on public cloud. *Software: Practice and Experience*, 47(2):223–248, 2017. (Cited on page 17.)

[42] C. Fuerst, S. Schmid, L. Suresh, and P. Costa. Kraken: Online and elastic resource reservations for multi-tenant datacenters. In *Computer Communications (INFOCOM), IEEE International Conference on*, 2016. (Cited on page 20.)

[43] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004. (Cited on page 20.)

[44] A. E. Gencer, S. Basu, I. Eyal, R. van Renesse, and E. G. Sirer. Decentralization in bitcoin and

ethereum networks. *arXiv preprint arXiv:1801.03998*, 2018. (Cited on page 117.)
[45] M. Ghijsen, J. Van Der Ham, P. Grosso, C. Dumitru, H. Zhu, Z. Zhao, and C. De Laat. A semantic-web approach for modeling computing infrastructures. *Computers & Electrical Engineering*, 39(8): 2553–2565, 2013. (Cited on page 54.)
[46] N. Ghosh and S. K. Ghosh. An approach to identify and monitor sla parameters for storage-as-a-service cloud delivery model. In *Globecom Workshops (GC Wkshps), 2012 IEEE*, pages 724–729. IEEE, 2012. (Cited on page 109.)
[47] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017. (Cited on page 137.)
[48] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009. (Cited on page 3.)
[49] N. Grozev and R. Buyya. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390, 2014. (Cited on page 53.)
[50] A. F. M. Hani, I. V. Paputungan, and M. F. Hassan. Renegotiation in service level agreement management for a cloud-based system. *ACM Computing Surveys (CSUR)*, 47(3):51, 2015. (Cited on page 109.)
[51] O. Haq and F. R. Dogar. Leveraging the power of cloud for reliable wide area communication. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 19. ACM, 2015. (Cited on page 99.)
[52] T. Heuer and S. Schlag. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. (Cited on page 60.)
[53] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. " O'Reilly Media, Inc.", 2017. (Cited on page 16.)
[54] M. Httermann. *DevOps for developers*. Apress, 2012. (Cited on page 2.)
[55] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao. Ecsched: Efficient container scheduling on heterogeneous clusters. In *European Conference on Parallel Processing*, pages 365–377. Springer, 2018. (Cited on page 20.)
[56] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on parallel and distributed systems*, 27(1):130–143, 2015. (Cited on page 3.)
[57] J. Hyun, J. Li, C. Im, J.-H. Yoo, and J. W.-K. Hong. A high performance volte traffic classification method using htcondor. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 518–524. IEEE, 2015. (Cited on page 20.)
[58] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. V. Papadopoulos, B. Ghit, D. Epema, and A. Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 75–86. ACM, 2017. (Cited on pages 3, 17, and 73.)
[59] C. Ji, Y. Li, W. Qiu, U. Awada, and K. Li. Big data processing in cloud computing environments. In *2012 12th International Symposium on Pervasive Systems, Algorithms and Networks*, pages 17–23. IEEE, 2012. (Cited on page 97.)
[60] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal cloud resource auto-scaling for web applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 58–65. IEEE, 2013. (Cited on page 109.)
[61] J. Jiang, R. Das, G. Ananthanarayanan, P. A. Chou, V. Padmanabhan, V. Sekar, E. Dominique, M. Goliszewski, D. Kukoleca, R. Vafin, et al. Via: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 286–299. ACM, 2016. (Cited on page 99.)
[62] L. Kan, Y. Wei, A. H. Muhammad, W. Siyuan, G. Linchao, and H. Kai. A multiple blockchains architecture on inter-blockchain communication. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 139–145. IEEE, 2018. (Cited on page 137.)
[63] J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia. Savi testbed: Control and management of converged virtual ict resources. In *Integrated Network Management, IFIP/IEEE International Symposium on*, pages 664–667, 2013. (Cited on pages 20 and 83.)
[64] J.-M. Kang, T. Lin, H. Bannazadeh, and A. Leon-Garcia. Software-defined infrastructure and the savi

testbed. In *International Conference on Testbeds and Research Infrastructures*, pages 3–13. Springer, 2014. (Cited on page 20.)

[65] K. Kaur, T. Dhand, N. Kumar, and S. Zeadally. Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE wireless communications*, 24 (3):48–56, 2017. (Cited on page 137.)

[66] K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In *eScience'08. IEEE Fourth International Conference on*, pages 301–308, 2008. (Cited on page 83.)

[67] K. T. Kearney, F. Torelli, and C. Kotsokalis. Sla: An abstract syntax for service level agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224. IEEE, 2010. (Cited on page 109.)

[68] I. K. Kim, J. Steele, Y. Qi, and M. 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. (Cited on page 89.)

[69] Y. Kouki, F. A. De Oliveira, S. Dupont, and T. Ledoux. A language support for cloud elasticity management. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 206–215. IEEE, 2014. (Cited on page 109.)

[70] S. Koulouzis, A. S. Belloum, M. T. Bubak, Z. Zhao, M. Živković, and C. T. de Laat. Sdn-aware federation of distributed data. *Future Generation Computer Systems*, 56:64–76, 2016. (Cited on page 18.)

[71] S. Koulouzis, P. Martin, T. Carval, B. Grenier, G. Judeau, J. Wang, H. Zhou, C. de Laat, and Z. Zhao. Seamless infrastructure customisation and performance optimisation for time-critical services in data infrastructures. In *Proceedings of the 8th International Workshop on Data-Intensive Computing in the Clouds, ACM SIGHPC, in IEEE Supercomputing*, 2017. (Cited on page 14.)

[72] S. Koulouzis, P. Martin, H. Zhou, Y. Hu, J. Wang, T. Carval, B. Grenier, J. Heikkinen, C. de Laat, and Z. Zhao. Time-critical data management in clouds: Challenges and a dynamic real-time infrastructure planner (drip) solution. *Concurrency and Computation: Practice and Experience*, page e5269, 2019. (Cited on page 17.)

[73] S.-k. Kwon and J.-h. Noh. Implementation of monitoring system for cloud computing environments. *International Journal of Modern Engineering Research (IJMER)*, 3(4):1916–1918, 2013. (Cited on page 28.)

[74] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009. (Cited on page 53.)

[75] S. Lane and I. Richardson. Process models for service-based applications: A systematic literature review. *Information and Software Technology*, 53(5):424–439, 2011. (Cited on page 16.)

[76] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 225–236. IEEE, 2013. (Cited on page 60.)

[77] H. Lin, X. Qi, S. Yang, and S. Midkiff. Workload-driven vm consolidation in cloud data centers. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 207–216. IEEE, 2015. (Cited on page 3.)

[78] R. Mahmud, R. Kotagiri, and R. Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018. (Cited on page 136.)

[79] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012. (Cited on pages 52, 55, 56, 60, and 89.)

[80] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008. (Cited on page 20.)

[81] S. Meng and L. Liu. Enhanced monitoring-as-a-service for effective cloud management. *IEEE Transactions on Computers*, 62(9):1705–1720, 2012. (Cited on page 73.)

[82] N. Mishra, C.-C. Lin, and H.-T. Chang. A cognitive adopted framework for iot big-data management and knowledge discovery prospective. *Jour. of Distributed Sensor Networks*, 2015. (Cited on page 96.)

[83] K. Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016. (Cited on page 19.)

[84] C. Muller, M. Oriol, X. Franch, J. Marco, M. Resinas, A. Ruiz-Cortes, and M. Rodriguez. Comprehensive explanation of sla violations at runtime. *IEEE Transactions on Services Computing*, 7(2):168–183,

2014. (Cited on page 109.)
[85] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. (Cited on pages 21, 109, and 110.)
[86] H. Nakashima and M. Aoyama. An automation method of sla contract of web apis and its platform based on blockchain concept. In *Cognitive Computing (ICCC), 2017 IEEE International Conference on*, pages 32–39. IEEE, 2017. (Cited on pages 109 and 110.)
[87] J. F. Nash et al. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950. (Cited on page 119.)
[88] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar. Provisioning software-defined iot cloud systems. In *2014 2nd International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 288–295. IEEE, 2014. (Cited on page 72.)
[89] V. Nelson and V. Uma. Semantic based resource provisioning and scheduling in inter-cloud environment. In *Recent Trends in Information Technology (ICRTIT), 2012 International Conference on*, pages 250–254. IEEE, 2012. (Cited on page 53.)
[90] S. J. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley-IEEE Press, 2011. (Cited on page 15.)
[91] D. Petcu, B. Di Martino, S. Venticinque, M. Rak, T. Máhr, G. E. Lopez, F. Brito, R. Cossu, M. Stopar, S. Šperka, et al. Experiences in building a mosaic of clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):12, 2013. (Cited on pages 18 and 28.)
[92] V. G. Pinto, L. Stanisic, A. Legrand, L. M. Schnorr, S. Thibault, and V. Danjean. Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach. In *2016 Third Workshop on Visual Performance Analysis (VPA)*, pages 17–24. IEEE, 2016. (Cited on page 15.)
[93] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 43–57, 2015. (Cited on page 3.)
[94] H. Qian and D. Andresen. Automate scientific workflow execution between local cluster and cloud. *the International Journal of Networked and Distributed Computing (IJNDC)*, 4(1):45–54, 2016. (Cited on page 89.)
[95] R. Ranjan. Streaming big data processing in datacenter clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014. (Cited on pages 96 and 97.)
[96] H. Ritzdorf, K. Wüst, A. Gervais, G. Felley, and S. Čapkun. Tls-n: Non-repudiation over tls enabling ubiquitous content signing for disintermediation. *IACR ePrint report*, 578, 2017. (Cited on page 110.)
[97] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu. Understanding the performance and potential of cloud computing for scientific applications. *IEEE Transactions on Cloud Computing*, 5(2):358–371, 2017. (Cited on page 89.)
[98] V. Scoca, R. B. Uriarte, and R. De Nicola. Smart contract negotiation in cloud computing. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 592–599. IEEE, 2017. (Cited on page 110.)
[99] W. Shi, C. Wu, and Z. Li. An online mechanism for dynamic virtual cluster provisioning in geo-distributed clouds. In *Computer Communications (INFOCOM), IEEE International Conference on*, 2016. (Cited on page 20.)
[100] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), IEEE 26th symposium on*, pages 1–10. IEEE, 2010. (Cited on page 96.)
[101] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007. (Cited on page 12.)
[102] S. G. Soriga and M. Barbulescu. A comparison of the performance and scalability of xen and kvm hypervisors. In *2013 RoEduNet International Conference 12th Edition: Networking in Education and Research*, pages 1–6. IEEE, 2013. (Cited on page 11.)
[103] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software*, 136:19–38, 2018. (Cited on pages 28, 37, and 72.)
[104] S. Talluri, A. Łuszczak, C. L. Abad, and A. Iosup. Characterization of a big data storage workload in the cloud. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 33–44. ACM, 2019. (Cited on page 89.)
[105] C. Tang. Fvd: A high-performance virtual machine image format for cloud. In *USENIX Annual Technical Conference*, 2011. (Cited on page 53.)

[106] D. Thomas, A. Hunt, C. Fowler, et al. *Programming Ruby: the pragmatic programmers' guide*. Raleigh, NC: Pragmatic Bookshelf,, 2005. (Cited on page 19.)

[107] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), IEEE International Conference on*, pages 996–1005. IEEE, 2010. (Cited on page 96.)

[108] D. Trihinas, G. Pallis, and M. D. Dikaiakos. Jcatascopia: Monitoring elastically adaptive applications in the cloud. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 226–235. IEEE, 2014. (Cited on pages 28 and 73.)

[109] R. B. Uriarte, F. Tiezzi, and R. D. Nicola. Slac: A formal service-level-agreement language for cloud computing. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 419–426. IEEE Computer Society, 2014. (Cited on page 109.)

[110] T. Van Damme, C. De Persis, and P. Tesi. Optimized thermal-aware job scheduling and control of data centers. *IEEE Transactions on Control Systems Technology*, 27(99):760–771, 2019. (Cited on page 3.)

[111] S. Venkateswaran and S. Sarkar. Architectural partitioning and deployment modeling on hybrid clouds. *Software: Practice and Experience*, 48(2):345–365, 2018. (Cited on pages 3 and 17.)

[112] T. Wang, B. Jaumard, and C. Develder. A scalable model for multi-period virtual network mapping for resilient multi-site data centers. In *Transparent Optical Networks (ICTON), 2015 17th International Conference on*. IEEE, 2015. (Cited on page 53.)

[113] T. Wang, F. Liu, and H. Xu. An efficient online algorithm for dynamic sdn controller assignment in data center networks. *IEEE/ACM Transactions on Networking*, 25(5):2788–2801, 2017. (Cited on page 3.)

[114] X. Wang, C. S. Yeo, R. Buyya, and J. Su. Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm. *Future Generation Computer Systems*, 27(8):1124–1134, 2011. (Cited on page 17.)

[115] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath. Image distribution mechanisms in large scale cloud providers. In *Cloud Computing Technology and Science, 2010 IEEE 2nd International Conference on*, pages 112–117. IEEE, 2010. (Cited on page 53.)

[116] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann. Streamlining devops automation for cloud applications using tosca as standardized metamodel. *FGCS*, 56:317–332, 2016. (Cited on pages 17 and 28.)

[117] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012. (Cited on page 16.)

[118] K. Wüst and A. Gervais. Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54. IEEE, 2018. (Cited on page 137.)

[119] Y. Xin, I. Baldine, A. Mandal, C. Heermann, J. Chase, and A. Yumerefendi. Embedding virtual topologies in networked clouds. In *Proceedings of the 6th International Conference on Future Internet Technologies*, pages 26–29. ACM, 2011. (Cited on page 53.)

[120] A. R. Zamani, M. Zou, J. Diaz-Montes, I. Petri, O. Rana, A. Anjum, and M. Parashar. Deadline constrained video analysis via in-transit computational environments. *IEEE Trans on Services Computing*, 2017. (Cited on pages 16 and 18.)

[121] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 aCM sIGSAC conference on computer and communications security*, pages 270–282. ACM, 2016. (Cited on page 110.)

[122] Y. Zhang and N. Ansari. Hero: Hierarchical energy optimization for data center networks. *IEEE Systems Journal*, 9(2):406–415, 2013. (Cited on page 3.)

[123] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu. Vmthunder: fast provisioning of large-scale virtual machine clusters. *Parallel and Distributed Systems, IEEE Transactions on*, 25(12):3328–3338, 2014. (Cited on page 53.)

[124] Z. Zhang, D. Li, and K. Wu. Large-scale virtual machines provisioning in clouds: challenges and approaches. *Frontiers of Computer Science*, 10(1):2–18, 2016. (Cited on pages 53 and 98.)

[125] Z. Zhao, A. Belloum, and M. Bubak. Special section on workflow systems and applications in e-science. *Future Generation Computer Systems*, 25(5):525–527, 2009. (Cited on page 88.)

[126] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses, et al. Developing and operating time critical applications in clouds: The state of the art and the switch approach. *Procedia Computer Science*, 68:17–28, 2015. (Cited on pages 14, 52, 54, 97, and 108.)

[127] X. Zheng, P. Martin, W. Powley, and K. Brohman. Applying bargaining game theory to web services negotiation. In *2010 IEEE International Conference on Services Computing*, pages 218–225. IEEE, 2010. (Cited on page 118.)

[128] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang. Blockchain challenges and opportunities: A

survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018. (Cited on page 21.)

[129] A. C. Zhou, Y. Gong, B. He, and J. Zhai. Efficient process mapping in geo-distributed cloud data centers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 16. ACM, 2017. (Cited on page 3.)

[130] H. Zhou, J. Wang, Y. Hu, J. Su, P. Martin, C. De Laat, and Z. Zhao. Fast resource co-provisioning for time critical applications based on networked infrastructures. In *Cloud Computing (CLOUD), IEEE International Conference on*, pages 802–805, 2016. (Cited on page 132.)

[131] H. Zhou, C. de Laat, and Z. Zhao. Trustworthy cloud service level agreement enforcement with blockchain based smart contract. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 255–260. IEEE, 2018. (Cited on page 115.)

[132] H. Zhou, S. Koulouzis, Y. Hu, J. Wang, C. de Laat, A. Ulisses, and Z. Zhao. Migrating live streaming applications onto clouds: Challenges and a cloudsstorm solution. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 321–326. IEEE, 2018. (Cited on page 52.)

[133] H. Zhou, A. Taal, S. Koulouzis, J. Wang, Y. Hu, G. Suciu, V. Poenaru, C. de Laat, and Z. Zhao. Dynamic real-time infrastructure planning and deployment for disaster early warning systems. In *International Conference on Computational Science*, pages 644–654. Springer, 2018. (Cited on pages 17 and 52.)

[134] H. Zhu, X. Liao, C. de Laat, and P. Grosso. Joint flow routing-scheduling for energy efficient software defined data center networks: A prototype of energy-aware network management platform. *Journal of Network and Computer Applications*, 63:110–124, 2016. (Cited on page 3.)

[135] J. Zhu, Z. Jiang, and Z. Xiao. Twinkle: A fast resource provisioning mechanism for internet services. In *INFOCOM, 2011 Proceedings IEEE*, pages 802–810. IEEE, 2011. (Cited on page 53.)

[136] H. Ziafat and S. M. Babamir. Optimal selection of vms for resource task scheduling in geographically distributed clouds using fuzzy c-mean and molp. *Software: Practice and Experience*, 2018. (Cited on page 17.)

Acronyms

API Application Programming Interface.

AWS Amazon Web Service.

CJDL Cloud Job Description Language.

DAS-5 The Distributed ASCI Supercomputer 5.

DC Data Centre.

DevOps software Development and Operations.

DRIP Dynamic Real-Time Infrastructure Planner.

EC2 Elastic Compute Cloud.

EEA European Economic Area.

EGI European Grid Infrastructure.

EU European Union.

EVM Ethereum Virtual Machine.

FPGA Field Programmable Gate Array.

GCE Google Compute Engine.

GDPR General Data Protection Regulation.

GENI Global Environment for Network Innovations.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HDFS Hadoop Distributed File System.

IaaS Infrastructure-as-a-Service.

ICMP Internet Control Message Protocol.

IM Infrastructure Manager.

INDL Infrastructure and Network Description Language.

IoT Internet of Things.

IRPS Inter-cloud Resource Provisioning System.

JAR Java ARchive.

JVM Java Virtual Machine.

KVM Kernel-based Virtual Machine.

NAT Network Address Translation.

NIaaS Networked Infrastructure-as-a-Service.

OS Operating System.

P2P Peer to Peer.

PaaS Platform-as-a-Service.

PBFT Practical Byzantine Fault Tolerance.

PoS Proof of Stake.

PoW Proof of Work.

QoE Quality of Experience.

QoS Quality of Service.

REST REpresentational State Transfer.

RQ Research Question.

SaaS Software-as-a-Service.

SAVI System Architecture Virtual Integration.

SDK Software Development Kit.

SDN Software-defined Network.

SLA Service Level Agreement.

SSH Secure Shell.

TOSCA Topology and Orchestration Specification for Cloud Applications.

UvA University of Amsterdam.

vCPU virtual CPU.

VIF Virtual Infrastructure Function.

VLAN Virtual Local Area Network.

VM Virtual Machine.

YAML YAML Ain't Markup Language.

Summary

By providing elastic infrastructure capacity and flexible pay-as-you-go business model, Cloud environments can significantly reduce the operational cost for resource-intensive applications like big data, deep learning, and the Internet of Things (IoT). In the application lifecycle, Clouds can not only automate the provisioning of application infrastructure, the deployment of the software components, but also provide advanced features, e.g., dynamic migration, scaling, and programmable virtual networking, for adapting the complex application to be continuously operational. However, these advanced Cloud features are so far only used at runtime phase of the application and have not yet been effectively included in the application programming model, which makes the Cloud application optimisation difficult across the entire software development and operation (DevOps) lifecycle. For applications with high-quality constraints, e.g., when processing IoT data queries within a given time window, or transmitting the data from the source to the computing resource within required latency, only runtime adaption will not be sufficient if infrastructures are not correctly designed.

We thus identify our key research question as: *how to seamlessly program and control the virtual infrastructure in the Cloud application DevOps lifecycle*? To tackle the problem, we investigated how to leverage "Infrastructure as Code" to represent infrastructure specifications and to model infrastructure operations, e.g., scaling and failure recovery. By decoupling the infrastructure abstraction from the application development and operation, we studied effective Cloud programming models and control mechanisms. Overlay network mechanisms were further explored to provision a networked infrastructure. Finally, we investigated blockchain to improve Cloud Service Level Agreement (SLA) for enhancing the infrastructure service quality assurance from the provider at runtime.

In this thesis, we tackle the main research question by dividing into the following four detailed research questions, according to different aspects of Cloud DevOps, including developing, provisioning, operating, and SLA assurance:

- How can we program and customise the infrastructure according to different application quality requirements?

  We analysed the common service characteristics and infrastructure lifecycle of different Clouds, modelled the basic Cloud virtual infrastructure functions, i.e., provisioning and terminating one VM. Then we empower Cloud applications with the infrastructure programmability at levels of resource customisation, operation description, and application logic to enable fine-grained topology and operation description.

- How can we effectively provision a networked infrastructure and enable topology partitioning across data centres or Cloud providers based on application QoS constraints?

  We developed overlay network mechanisms for transparently configuring network among virtual infrastructures, in particular when they are provisioned across data centres. Thus, we can partition the infrastructure to be distributed. Combining the Cloud performance study, we can improve the efficiency of provisioning and adapting distributed virtual infrastructure through independently controlling partitioned infrastructures.

- How can an application efficiently control the virtual infrastructure at runtime, preferably without vendor lock-in?

We proposed a control model for distributed virtual infrastructures (often across data centres and providers) with two control modes as i) a passive mode based on monitoring and predefined conditions, and ii) an active mode by providing programming interfaces to be included in the application logic. We implemented those control in an open framework, called CloudsStorm, which provides distributed control engines to execute the infrastructure operations.

To validate our solutions to all the above questions, we demonstrated the usage of CloudsStorm in managing dynamic task-based applications and in supporting service-based applications on Clouds. The experimental studies demonstrated that CloudsStorm can improve the application QoS and reduce the monetary cost of the Cloud resource usage at the same time.

- How can we effectively handle the SLA with provider to make the service quality assurance trustworthy?

We developed a blockchain based witness model. In the model, a new role, "Witness", was introduced for detecting service violation. Witnesses can gain rewards when taking the responsibility. An incentive model was carefully designed to guarantee their trustworthiness: witnesses have to always tell the truth, in order to maximise their rewards. This conclusion was analysed and proved by game theory. Furthermore, we have implemented the prototype system leveraging the smart contracts of Ethereum blockchain and performed experiments.

Samenvatting

Door het aanbieden van elastische infrastructuurcapaciteit en flexibel pay-as-you-go bedrijfsmodel, kunnen cloudomgevingen de operationele kosten voor resource-intensieve applicaties zoals big data, deep learning en het Internet of Things (IoT) aanzienlijk verlagen. In de toepassingslevenscyclus kan Cloud or kunnen Clouds niet alleen de levering van toepassingsinfrastructuur en de implementatie van de software-componenten automatiseren, maar ook geavanceerde functies bieden, zoals dynamische migratie, schaling en programmeerbare virtuele netwerken, voor het aanpassen van de complexe toepassing zodat deze continu operationeel is. Deze geavanceerde Cloud-functies worden tot nu toe echter alleen gebruikt in de runtime-fase van de applicatie en zijn nog niet effectief opgenomen in het programmeermodel van de applicatie, waardoor de optimalisatie van de Cloud-applicatie moeilijk is gedurende de gehele levenscyclus van softwareontwikkeling en -bediening (DevOps) moeilijk is. Voor toepassingen met van hoge kwaliteitseisen, bijvoorbeeld bij het verwerken van IoT-gegevensquery's binnen een bepaald tijdvenster of het verzenden van de gegevens van de bron naar de computerresource binnen de beperkte tijd, is alleen een runtime-aanpassing niet voldoende als de infrastructuur niet correct is ontworpen.

Onze belangrijkste onderzoeksvraag is dus: *hoe kunnen we de virtuele infrastructuur in de DevOps-levenscyclus van de Cloud-applicatie naadloos programmeren en besturen*? Om het probleem aan te pakken, hebben we onderzocht hoe we " Infrastructure as Code " kunnen gebruiken om infrastructuurspecificaties weer te geven en infrastructuuractiviteiten te modelleren, bijvoorbeeld voor schaalvergroting en herstel van storingen. Door de abstractie van de infrastructuur los te koppelen van de ontwikkeling en werking van de applicatie, hebben we effectieve cloud-programmeermodellen en besturingsmechanismen bestudeerd. Ook hebben wij overlay-netwerkmechanismen verder onderzocht om een netwerkinfrastructuur te bieden. Tot slot hebben we blockchain onderzocht om de Cloud Service Level Agreement (SLA) te verbeteren om de kwaliteit van de infrastructuur tijdens runtime beter te kunnen waarborgen.

In dit proefschrift behandelen we de belangrijkste onderzoeksvraag door het op te delen in de volgende vier gedetailleerde onderzoeksvragen, aan de hand van de verschillende aspecten van Cloud DevOps, waaronder ontwikkeling, levering, uitvoering en SLA assurance:

- Hoe kunnen we de infrastructuur programmeren en aanpassen aan de hand van verschillende vereisten voor de kwaliteit van applicaties?

We hebben de gemeenschappelijke servicekenmerken en infrastructuurlevenscyclus van verschillende Clouds geanalyseerd geanalyseerd en de. de basisfuncties van de virtuele cloudinfrastructuur gemodelleerd, d.w.z. het inrichten en beindigen van n VM. Vervolgens ondersteunen we cloud-applicaties met de programmeerbaarheid van de infrastructuur op het niveau van resource-aanpassing, operatiebeschrijving en applicatielogica om verfijnde topologie en operatiebeschrijving mogelijk te maken.

- Hoe kunnen we een netwerkinfrastructuur effectief inrichten en topologiepartitionering in datacenters of cloudproviders mogelijk maken op basis van QoS-beperkingen voor toepassingen?

We hebben overlay-netwerkmechanismen ontwikkeld voor het transparant configureren van het netwerk tussen virtuele infrastructuren, met name wanneer ze worden aangeboden in datacenters. Zo kunnen we de te distribueren infrastructuur partitioneren. Door het Cloud-prestatieonderzoek te combineren, kunnen we de efficintie van het bevoorraden en aanpassen van gedistribueerde virtuele infrastructuur verbeteren door door de gepartitioneerde infrastructuren onafhankelijk te beheren.

- Hoe kan een applicatie de virtuele infrastructuur tijdens runtime efficint beheren, bij voorkeur zonder lock-in van leveranciers?

We stelden een besturingsmodel voor gedistribueerde virtuele infrastructuren (vaak over datacenters en providers) voor met twee besturingsmodi, zoals i) een passieve modus op basis van monitoring en vooraf gedefinieerde omstandigheden, en ii) een actieve modus door programmeerinterfaces te bieden die in de applicatielogica moeten worden opgenomen. We hebben die besturing gemplementeerd in een open framework, CloudsStorm genaamd, dat gedistribueerde besturingsengines biedt om de infrastructuuractiviteiten uit te voeren.

Om onze oplossingen voor alle bovenstaande vragen te valideren, hebben wij CloudsStorm gebruikt voor het beheren van dynamische taakgebaseerde toepassingen en bij het ondersteunen van op service gebaseerde toepassingen op Clouds. Uit de experimentele onderzoeken is gebleken dat CloudsStorm de Servicekwaliteit van de applicatie kan verbeteren en tegelijkertijd de monetaire kosten van het gebruik van cloudresources kan verlagen.

- Hoe kunnen we effectief omgaan met de SLA met de provider om de kwaliteitsborging betrouwbaar te maken?

We hebben een op blockchain gebaseerd getuigenmodel ontwikkeld. In het model werd een nieuwe rol gentroduceerd, “Getuige”, die overtredingen van de SLA kan detecteren. Getuigen kunnen beloningen ontvangen wanneer ze de verantwoordelijkheid nemen. Een stimuleringsmodel werd zorgvuldig ontworpen om hun betrouwbaarheid te garanderen: getuigen moeten altijd de waarheid vertellen om hun beloningen te maximaliseren. Deze conclusie werd geanalyseerd en bewezen door speltheorie. Verder hebben we het prototypesysteem gemplementeerd met behulp van de slimme contracten van Ethereum blockchain en experimenten uitgevoerd.

Journal Publications

- **Zhou, H.**, Ouyang, X., Su, J., de Laat, C., Zhao, Z., "Enforcing Trustworthy Cloud SLA with Witnesses: A Game Theory based Model using Smart Contracts", Journal of *Concurrency and Computation: Practice and Experience*, e5511. Wiley, 2019.
- **Zhou, H.**, Hu, Y., Ouyang, X., Su, J., Koulouzis, S., de Laat, C., Zhao, Z., "CloudsStorm: A Framework for Seamlessly Programming and Controlling Virtual Infrastructure Functions during the DevOps Lifecycle of Cloud Applications", Journal of *Software: Practice and Experience*. Wiley, 2019.
- Koulouzis, S., Martin, P., **Zhou, H.**, Hu, Y., Wang, J., Carval, T., Grenier, B., Heikkinen, J., de Laat, C., Zhao, Z., "Time-critical data management in clouds: Challenges and a Dynamic Real-Time Infrastructure Planner (DRIP) solution", Journal of *Concurrency and Computation: Practice and Experience*, e5269. Wiley, 2019. (as co-first author)
- Hu, Y., **Zhou, H.**, de Laat, C., Zhao, Z., "Concurrent Container Scheduling on Heterogeneous Clusters with Multi-Resource Constraints", Journal of *Future Generation Computer Systems*. Elsevier, 2019.
- Uriarte, R., **Zhou, H.**, Kritikos, K., Shi, Z., De Nicola, R., Zhao, Z., "Distributed SLA Management with Smart Contracts and Blockchain", Journal of *Concurrency and Computation: Practice and Experience*. Wiley, 2019. (as co-first author, under review)
- Wang, J., Taal, A., Martin, P., Hu, Y., **Zhou, H.**, Pang, J., de Laat, C., Zhao, Z., "Planning virtual infrastructures for time critical applications with multiple deadline constraints", Journal of *Future Generation Computer Systems*, 75, pp. 365-375. Elsevier, 2017.

Conference Publications

- **Zhou, H.**, Ouyang, X., Ren, Z., Su, J., de Laat, C., Zhao, Z., "A Blockchain based Witness Model for Trustworthy Cloud Service Level Agreement Enforcement", In *IEEE International Conference on Computer Communications* (INFOCOM), pp. 1567-1575. IEEE, 2019. (Best In-session Presentation Award)
- Shi, Z., **Zhou, H.**, Hu, Y., Surbiryala, J., de Laat, C., Zhao, Z., "Operating Permissioned Blockchain in Clouds: A Performance Study of Hyperledger Sawtooth", In *18th IEEE International Symposium On Parallel And Distributed Computing* (ISPDC). IEEE, 2019.
- **Zhou, H.**, Shi, Z., Donkers, P., Afanasyev, A., Koulouzis, S., Taal, A., Ulisses, A., Zhao, Z., "Large distributed virtual infrastructure partitioning and provisioning across providers", In *IEEE International Conference on Smart Internet of Things* (SmartIoT). IEEE, 2019. (Best Student Paper Award)

- **Zhou, H.**, de Laat, C., Zhao, Z., "Trustworthy Cloud Service Level Agreement Enforcement with Blockchain Based Smart Contract", In *IEEE International Conference on Cloud Computing Technology and Science* (CloudCom), *workshop on resource brokering with blockchain* (RBChain), pp. 255-260. IEEE, 2019.

- **Zhou, H.**, Koulouzis, S., Hu, Y., Wang, J., de Laat, C., Ulisses, A., Zhao, Z., "Migrating Live Streaming Applications onto Clouds: Challenges and a CloudsStorm Solution", In *11th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (UCC Companion), *workshop on Cloud-Native Applications Design and Experience* (CNAX), pp. 321-326. IEEE, 2018.

- **Zhou, H.**, Taal, A., Koulouzis, S., Wang, J., Hu, Y., Suciu, G., Poenaru, V., de Laat, C., Zhao, Z., "Dynamic Real-Time Infrastructure Planning and Deployment for Disaster Early Warning Systems", In *International Conference on Computational Science, workshop on Data, Modeling, and Computation in IoT and Smart Systems*, pp. 644-654. Springer, Cham, 2018.

- **Zhou, H.**, Hu, Y., Su, J., Chi, M., de Laat, C., Zhao, Z., "Empowering Dynamic Task-Based Applications with Agile Virtual Infrastructure Programmability", In *IEEE 11th International Conference on Cloud Computing* (CLOUD), pp. 484-491. IEEE, 2018.

- **Zhou, H.**, Hu, Y., Su, J., de Laat, C., Zhao, Z., "CloudsStorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures", In *International Conference on Cloud Computing*, pp. 265-280. Springer, Cham, 2018.

- Hu, Y., **Zhou, H.**, de Laat, C., Zhao, Z., "Ecsched: Efficient container scheduling on heterogeneous clusters", In *European Conference on Parallel Processing* (EuroPar), pp. 365-377. Springer, Cham, 2018.

- Ouyang, X., **Zhou, H.**, Clement, S., Townend, P., Xu, J., "Mitigate data skew caused stragglers through ImKP partition in MapReduce", In *IEEE 36th International Performance Computing and Communications Conference* (IPCCC), pp. 1-8. IEEE, 2017.

- Wang, J, **Zhou, H.**, Hu, Y., de Laat, C., Zhao, Z., "Deadline-aware coflow scheduling in a dag", In *IEEE International Conference on Cloud Computing Technology and Science* (CloudCom), workshop on NetCloud, pp. 341-346. IEEE, 2017.

- Hu, Y., Wang, J., **Zhou, H.**, Martin, P., Taal, A., de Laat, C., Zhao, Z., "Deadline-aware deployment for time critical applications in clouds", In *European Conference on Parallel Processing* (EuroPar), pp. 345-357. Springer, Cham, 2017.

- Elzinga, O., Koulouzis, S., Taal, A., Wang, J., Hu, Y., **Zhou, H.**, Martin, P., de Laat, C., Zhao, Z., "Automatic collector for dynamic cloud performance information", In *International Conference on Networking, Architecture, and Storage* (NAS), pp. 1-6. IEEE, 2017.

- **Zhou, H.**, Martin, P., Su, J., de Laat, C., Zhao, Z., "A Flexible Inter-locale Virtual Cloud For Nearly Real-time Big Data Application", In *IEEE Real Time System Symposium* (RTSS), *International workshop on Interoperable infrastructures for interdisciplinary big data sciences* (IT4RIs), 2016.

- **Zhou, H.**, Wang, J., Hu, Y., Su, J., Martin, P., de Laat, C., Zhao, Z., "Fast resource co-provisioning for time critical applications based on networked infrastructures", In *IEEE 9th International Conference on Cloud Computing* (CLOUD), pp. 802-805. IEEE, 2016.

- **Zhou, H.**, Hu, Y., Wang, J., Martin, P., de Laat, C., Zhao, Z., "Fast and dynamic resource provisioning for quality critical cloud applications", In *IEEE 19th International Symposium on Real-Time Distributed Computing* (ISORC), pp. 92-99. IEEE, 2016.

Other Publications

- **Zhou, H.**, Martin, P., de Laat, C., Zhao, Z., "A network transparent solution for flexibly provisioning connected virtual infrastructure across multiple data centers", poster and oral presentation in *ICT.OPEN2017*, 2017. `https://www.delaat.net/posters/pdf/2017-03-22-switch-ictopen.pdf`

- **Zhou, H.**, de Laat, C., Zhao, Z., "CloudsStorm: An Application-driven DevOps Framework for Managing Networked Infrastructures on Federated Clouds", poster and oral presentation in *ICT.OPEN2018*, 2018. `https://www.delaat.net/posters/pdf/2018-03-20-ICTOPEN_Huan.pdf`

Supervision

- Andrey Afanasyev. "Virtual infrastructure partitioning and provisioning under nearly real-time constraints", Master's thesis, University of Amsterdam, 2018. `https://work.delaat.net/rp/2017-2018/p50/report.pdf`

Software Development

- CloudsStorm – The core engine to empower the application with the ability to seamlessly program and control Cloud virtual infrastructures.
  Code: `https://github.com/zh9314/CloudsStorm`
  Documentation: `https://cloudsstorm.github.io/`
  Integration: integrated as the provisioning agent of DRIP component in the software release of European project SWITCH and ARTICONF.

- CloudsStormCA – The control agent implementation for CloudsStorm framework, including the graphical user interface and REST APIs to be invoked.
  Code: `https://github.com/zh9314/CloudsStormCA`

- CloudsStormREST – The Java implantation of "*Infrastructure Embedded Code*" for invoking CloudsStorm REST APIs using Java.
  Code: `https://github.com/zh9314/CloudsStormREST`

- A prototype implementation using Ethereum blockchain based smart contracts for SLA enforcement.
  Code: `https://github.com/zh9314/SmartContract4SLA`

Acknowledgements

Pursuing the PhD degree during the past four years is not an easy journey for me. When I look back, all the things are so clear and just like what happened yesterday. I can still remember that it was an early and cold morning at the end of September in 2015 that I arrived in Amsterdam. I was so excited since it was the first time for me to be abroad. In the following four years, I have experienced all the moments of disappointment, struggling, and happiness, to go through this journey. First of all, I would like to thank the guy who worked so hard in these four years. Without him, this journey can never be finished and reach this point. He is on his own, but he is not alone.

I would like to thank my PhD promotor, Prof. Cees de Laat. The monthly meeting with him adjusted my research directions and inspired me a lot. I would also like to thank my supervisor, Dr. Zhiming Zhao. He provided me with plenty of insights through the intensive weekly meeting. Both of them are beacons for me to light and guide my research road.

I am also grateful towards Prof. Jinshu Su and Dr. Xiaofeng Wang, who are my supervisors during my bachelor and master degree. They taught me to write my first academic paper and led me to step into this amazing research world. Without the seed of interest planted by them, I can never imagine to make all these happen and achieve this stage of finishing the thesis.

For this thesis, I am honored to have Prof. Henri Bal, Prof. Radu Prodan, Prof. Jinshu Su, Prof. Rob V. van Nieuwpoort, Prof. Pieter Adriaans, and Dr. Adam Belloum as my committee members. Meanwhile, I appreciate their work on evaluating this thesis.

I also appreciate the majority of the funding support from the China Scholarship Council and the additional subsidies from the University of Amsterdam. They provided me with sufficient financial assistance allowing me to focus on the research.

It was also a fantastic experience to work with all the colleagues from SNE lab. I learned a lot from them through the group meeting, the reading club, and discussions. Moreover, I enjoyed the time that we spent together for skiing, drinking, and barbecue parties. I make a long list here to thank and memorise all of you: Ameneh, Ana, Andy, Arie, Benjamin, Clemens, Dolly, Fahimeh, Giovanni, Giulio, Hao, Hongyun, Henk, Jamila, Joseph, Julius, Junchao, Lu, Lu-Chi, Lukasz, Marijke, Mary, Misha, Mostafa, Paola, Paul, Pieter, Ralph, Reggi, Ruyue, Sara, Simon, Spiros, Uraz, Xiaofeng, Xin, Yuri, Zeshun.

Besides, it was fortunate for me to meet many interesting friends here. They come from different departments of University of Amsterdam and even other universities. I was delighted to chat with them on various topics, like biology, psychology, and philosophy. All of you mean a lot to me, though I cannot list all the names here: Biwen Wang, Hui Xiong, Jiahuan Pei, Jian Lin, Jinglan Wang, Lingling Zhang, Nan Jiang, Peng Wang, Qi Wang, Renjie Lv, Shuai Liao, Shuaishuai Wang, Si Wen, Songyu Yang, Wanyu Chen, Wei Zhang, Weijian Wu, Wenyang Wu, Xianya Mi, Xinyi Li, Yang Liu, Yifan Chen, Yiwei Sun, Yumei Wang, Zhijie Ren, Zenglin Shi, Ziming Li. The names are too many to list. If your name has been left out, please still accept my sincere thanks. Especially, I would like to give special thanks to Jun, Xiaolong, Yang, and Zijian. They have accompanied me for these four years. I benefited a lot from discussions with them and enjoyed the time playing games and laughing together.

I would definitely like to thank my parents, who give me unconditional love. I would also like to thank my other family members, my grandparents, parents-in-law, and my cousins. They all stand on my side and support me. Last but not the least, I would like to express my gratitude towards my beloved wife, Xue. We have known each other for ten years, and she has already been an indispensable part of my life. In those dark days, it was her to cheer me up to get through.

I tried so hard and got so far. In the end, it doesn't have to matter. But I enjoyed the process of the journey and would be grateful to all the people I have met.