實踐 DevOps 所需環境的自動化雲端沙箱部署
Automated Cloud Sandbox Deployment for Implementing DevOps

陳啟文, 王凡
CHI-WEN CHEN, FARN WANG
國立臺灣大學電機工程學研究所
Graduate Institute of Electrical Engineering, National Taiwan University

ABSTRACT

This paper introduces a software framework, called the" DEVOPSER", which enables automated cloud services deployment used to implement the *DevOps* [1][3] process. With the DEVOPSER, we can deploy sets of clouds where every cloud within has its position in the DevOps workflow. Our work aims at the design of the configuration documents that DEVOPSER needs, as well as the way to analyze the configuration documents, and translate them into the *Heat Orchestration Template(HOT)* (see 2.2), which enables automated cloud service deployment. We perform an experiment on the "DEVOPSER" to verify its capacity and functionality and show that DEVOPSER can parse our configuration document and output a valid HOT.

KEY WORLDS: *Cloud Computing, DevOps, Open-source Project, OpenStack, Software Engineering.*

1 INTRODUCTION

DevOps is a clipped compound of "software DEVelopment" and "information technology OPerationS." It is a practice focusing on establishing an environment where building, testing, and releasing software can happen frequently, rapidly, and more reliably. In traditional software delivery, developers and the operators work separately; as a result, there would be a dilemma when the operators had a problem while they did not know how to fix it because they were not involved in development. In order to handle this dilemma, the concept of "DevOps" was first mentioned in 2008[4]. It is a principle or a practice that enhances the efficiency of the software delivery by promoting the collaboration of the software developers and IT (Information Technology) operator. Every big software company nowadays adopts its own DevOps principles. But in general, the DevOps principles include the following ingredients:

(1) Making sure the application behaves and performs well in developing and testing.
(2) Creating a delivery pipeline for continuous automated deployment and testing the application.
(3) Amplifying the feedback loops between developers and operators.

There are many open-source tools aiming at the particular stage in the DevOps process; for example, Jenkins [5] is a CI (Continuous Integration) [6] tool for the build stage. In this paper, we present a framework which enables to automatically deploy all the requirements under the whole life cycle of the DevOps process, not just a tool used for a particular stage. We call it "DEVOPSER", which means that it is a tool for a whole DevOps process. Every stage in DevOps process for DEVOPSER requires a *SBAT* (Sandbox Arrangement Template) in YAML (Yet Another Markup Language) for its deployment. YAML is a human-readable data serialization language that is commonly used for configuration file [7]. We take the format in YAML to specify the SBAT for the convenience of parsing as there is a Python package parsing YAML format. DEVOPSER has the capability of parsing these SBAT into HOT (see 2.2) and the HOT enable us to automatically launch cloud services on OpenStack (see 2.1). We call the cloud services deployed by SBAT as the *Sandbox*. In our work, each DevOps stage is implemented by a Sandbox; for example, the Build stage for DevOps process is implemented by a *Build Sandbox*. The Sandbox can be composed of one or more than one machine and other cloud services.

HOT already has the capability of specifying the cloud deployment (see 2.2); however; in our work, we define another configuration document SBAT to specify the cloud deployment. The reason is that the syntax and structure of the HOT is much more complicated. In the HOT, we have to specify everything while deploying a machine, such as the network configuration to the machine, which is not the main point when we want to deploy for the DevOps process; nevertheless, the network configuration is essential if we want to deploy a valid machine to use. As a result, the reason that we define SBAT to implement the automated cloud sandbox deployment is to simplify the specification and enhance the automated deployment in the DevOps process.

In order to present our framework as an automated deployment tool for DevOps process, there are some issues that have to be considered as the following:

(1) How many SBAT the user has to provide to implement a DevOps pipeline?
Ans: The DevOps process include three main stages: (1) Development, (2) Build, (3) Deployment. As a result, with the SBAT specified each stage, we can deploy the resources under a DevOps pipeline.

(2) How to make sure that the user can deploy the environment for Continuous Integration(CI) (see 2.3) and how CI is implemented as it is an important step for the DevOps process?
Ans: In the SBAT which specifies the Build Sandbox, we request the user to give a script location URL to set up the booted environment of the Build Sandbox. And we perform the CI process with the help of the Jenkins, which is an open-source tool for Continuous Integration.

(3) How to ensure that the Sandbox deployment result meets all the requirements under the DevOps process?
Ans: The answer of (1) indicates that the DevOps

process includes three main stages. Each stage will deploy a Sandbox in charge of the assignment at that stage. If each sandbox is correctly built, we can convince the validity of the HOT. We know that each sandbox is specified by SBAT. If there is no error with the content of the SBAT and no parsing error, the deployment result can be convinced.

2 PRELIMINARY

2.1 Automated Cloud Service Deployment

We can implement the automated cloud services deployment by means of the OpenStack [8] [9]. OpenStack is a free and open-source software and is used for cloud computing and deployed as an Infrastructure as a service (IaaS) [10]. The OpenStack platform consists of many projects [11], such as *Nova*, which is a cloud-computing controller, *Neutron* is a system managing network services, and the *Horizon*, which provides the user with a graphical interface to access other project resources. After configuring all of the necessary projects, we can easily launch cloud applications on the OpenStack.

2.2 Heat Orchestration Template

Heat Orchestration Template (HOT) is a template format [12] that allows users to describe the deployments of the cloud applications in text file. HOT requires a specific syntax and structure supported by *Heat* in the OpenStack. Heat is a project in the OpenStack which implements an orchestration engine to automatically launch multiple composite cloud applications based on the HOT. As a result, we can launch cloud applications automatically on the OpenStack by uploading a valid HOT into the Heat.

```
heat_template_version: 2015-04-30
resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      image: ubuntu-trusty-x86_64
      flavor: m1.small
```

Fig.1 A simplest HOT sample used to deploy a machine

2.3 Continuous Integration(CI)

Continuous Integration is an important step at the build stage for the DevOps process [13].CI merges all the developer working copies to mainline, and performs tests or build in each version of the copies. It implements the automated build and tests hence prevents the integration problems. As a result, CI is a key step while implement the DevOps. We deploy the cloud resources to implement a Build Sandbox for CI on the OpenStack. The configurations of the Build Sandbox can be Specified by a SAT for it.

3 ARCHITECTURE OF THE DEVOPSER

Fig. 2 shows the architecture and the user story of the DEVOPSER. The DevOps workflow in our work includes three main stages: *Development*, *Build*, and *Deployment*. Each stage requires a SBAT to specify for its deployment. The DEVOPSER loads these SBAT as the input to produce HOT. Then, we can upload the output HOT to OpenStack to launch the Sandbox for the stages. The following in this section describes the design on SBAT, and the specification for SBAT to deploy each Sandbox respectively.

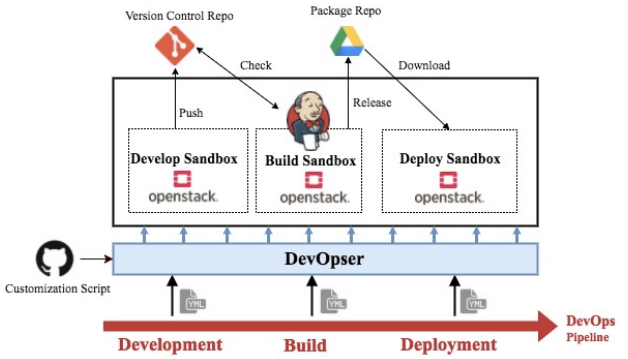

Fig. 2 The DEVOPSER architecture and its user story

3.1 Sandbox Arrangement Template

Sandbox Arrangement Template (SBAT) is a template input for DEVOPSER. SBAT is used to describe and configure the deployment of a Sandbox. In our work, we want to implement DEVOPSER as a platform as a service (PaaS) [10] cloud application. As we have mentioned, OpenStack is a IaaS cloud application that services cloud as the infrastructure, which means that OpenStack provides the basic elements or components under requests. However, the deployment for a DevOps story is usually deploying the machines in charge of assignments. As a consequence, we simply the process to launch a valid machine by only requesting the user to provide environment information of the machine such as its hardware requirements. Moreover, the configuration on the machine is another key while deploying the machine for DevOps since every machine under a DevOps workflow has its assignment. In other words, we must emphasize the flexibility on the machine because the user may want to make machine "versatile." In order that, we request the user to give what they want to set up on the machine in shell script, which enables automatically performing commands in Linux system. More flexibly, we define the time phases for the scripts for the users to enables that the scripts can be performed depend on its assignments and timing.

3.1 SBAT for the Develop Sandbox

The Develop Sandbox is used for what the developers will do under a DevOps process. Namely, before the developers start to develop an application, they might need to set up the development environment such as installing tools for their development. After they set up the environment, they start to program and perform some unit test on the developing machine to ensure each production is fit for use. Finally, while developers finish a version, the source code can be upload to a "Version Control Repository" (e.g. git). The Develop Sandbox takes charge of this assignment, and the following Fig.3 illustrates this process.

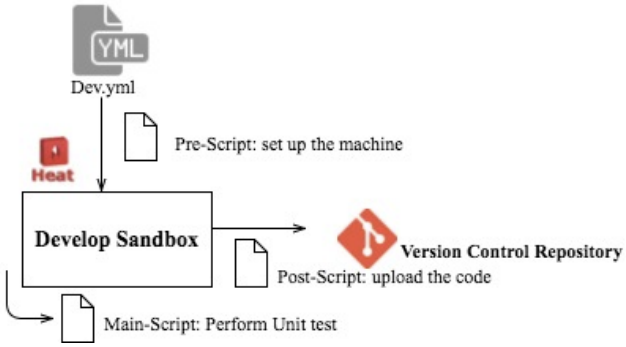

Fig.3 User Story of the Development Stage

In order to deploy the requirements for the Develop Sandbox, the user has to specify the SBAT as shown in Fig. 4. We specify the Develop Sandbox as a virtual machine that provides a developer environment.

Hostname: "Dev.1"
Application: "Python"
OS: "Ubuntu14"
Flavor:
cpu: "2" *#processor core count*
mem: "4" *#RAM in GB*
disk: "20" *#storage in GB*
SCM(*):
URL(*): "https://github.com/developer/DEV.git"
Pre-Stage(*): "DEV/init_env.sh"
Main-Stage(*): "DEV/unit_test.sh"
Post-Stage(*): "DEV/upload_code.sh"
(*) : Optional

Fig. 4 SBAT for a Develop Sandbox

As Fig. 4 shows, "Hostname", "Application", "OS" and "Flavor" are the four necessary keys in the SBAT for a Develop Sandbox. The "Flavor" specifies the hardware requirements of the machine. The "SCM" is short for "Source Code Management", which specifies a link and shell script directory for setting up the machine or for other assignment. A developer can provide the source code management URL (e.g. GitHub) to set up the environment of this develop machine or perform other assignment.

The "Pre-Stage", the "Main-Stage", and the "Post-Stage" are specified as the script directory after they are cloned to the develop machine. Each script can contain the commands for shell in Linux to automatically set up the environment for the machine or perform other assignments. For instance, we can provide the "URL" linking to a GitHub repository which has the prepared script that can be cloned to the built machine and executed in different time. The "Pre-Stage" script is used in the time before the developer starts to develop the applications. Usually, it means the time when the develop machine is just initiated. The "Main-Stage" script will be performed as the applications development goes on. Similarly, the "Post-Stage" script is used to set up the final state of the develop machine when the developer finishes developing the application. The "Post-Stage" can be used to upload the source code to "Version Control Repository". The "SCM" section in this SBAT is optional, meaning that it is not necessary to provide if the developer has no wish to customize the develop machine or perform some automated script.

3.2 Build Sandbox

The main purpose of the Build Sandbox is to launch a CI server for Jenkins, which will integrate the codes from "Version Control Repository" and automatically compile the source code to a "Package Repository" in the end. As we have mentioned, we launch the machines by HOT in OpenStack; however, the requirements under a CI server depend on what application is building. If the user just provides a basic specification without any customized script to set up the build machine, OpenStack may launch a raw machine that does not satisfy the requirements of Jenkins. In other words, it will be a problem to implementing the Build Sandbox if Jenkins doesn't work due to the lack of required resources or invalid system environment. As a result, the "Pre-Stage" in the "SCM" in this SBAT is necessary. The user must prepare a script to set up the environment for the Build Sandbox. After the Build Sandbox is complete successfully, Jenkins will do its work and return a build or test report for the users. Finally, we can compile and release the application by the "Main-Stage" and "Post-stage" script or just command in the Build Sandbox.

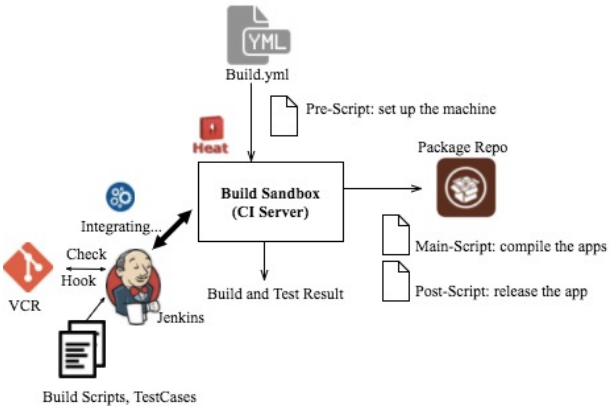

Fig. 5 User Story of the Build Sandbox.

The Fig.5 illustrates the process of the Build Sandbox. The specification of the "Pre-Stage" makes sure that we can enable the Build machine to automatically build and test the application by Jenkins. Developer can compile the applications with the "Main-Stage" script and release it to a Package Repository (e.g. Google Drive) with the "Post-Stage" Script in the build machine. Fig.6 specifies the SBAT for Build Sandbox. The specification of this sandbox is almost the same as SBAT for Develop Sandbox, except a "Pre-Stage" is necessary in this specification.

Hostname: "Build"
Application: "Python"
OS: "Ubuntu14"
Flavor:

cpu: "2" *#processor core count*
mem: "8" *#RAM in GB*
disk: "200" *#storage in GB*
SCM:
URL: "https://github.com/developer/BUILD.git"
Pre-Stage: " BUILD /init_env.sh"
Main-Stage: "BUILD /compile.sh"
Post-Stage: "BUILD /release.sh"

Fig. 6 SBAT for a Build Sandbox

3.3 Deploy Sandbox

Deploy Sandbox is used to deploying the application after it was released. At the Deployment stage of the DevOps processes, the user might request more than one machine in the deployment of the Deploy Sandbox for the purpose to establish a database, or a front website server ... etc. As a result, the specification of the SBAT for the Deploy Sandbox might be much more complicated than others.

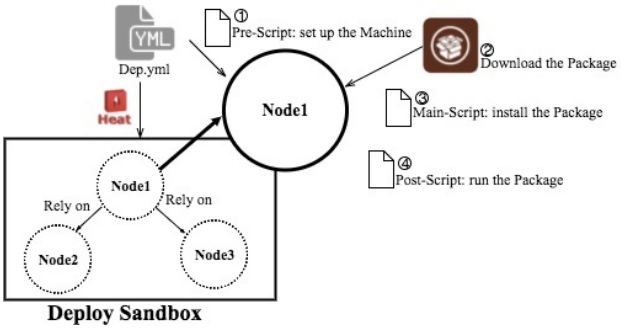

Fig. 7 User Story of the Development Stage.

Fig.7 illustrates a sample user story and the architecture of Deploy Sandbox. In this sandbox, more than one machine is deployed within. Every machine is work independently but may have a dependency with each other. Taking Eig.6 for instance, we can see that Node1 start to launch if and only if Node3 is completed. Each machine within the Deploy Sandbox is deployed by one SBAT for the Deploy Sandbox, and has its "Pre-Stage", "Main-Stage", and "Post-Stage" to specify its customization. Therefore, the way to deploy a machine is the same as previously, user has to specify the deployment in one SBAT. And if the user wants to launch more than one machine with the same configuration, he can just specify it in the "Hostname" splitting with a ",". The difference in this stage is the user has to specify the "Rely on" if there is a dependency and specify the "Package" with a URL which links to the package repository (e.g. a Google Drive link). Following Fig.8 shows the specification of the deployment SBAT for which the user story is shown in the Fig.7.

 # Node2 and Node3 is the same machine.
Hostname: "Node2","Node3"
Application: "Python"
OS: "Ubuntu14"
Flavor:
cpu: "2" #processor core count
mem: "8" #RAM in GB
disk: "200" #storage in GB
SCM:
URL: " https://github.com/developer/Deploy.git"
Pre-Stage: " Deploy /init_env.sh"
Main-Stage(*): " Deploy /install.sh"
Stage-Stage(*): " Deploy /run.sh"

Hostname: "Node1"
Rely on: "Node2","Node3"
Application: "Python"
OS: "Ubuntu14"
Flavor:
cpu: "2" #processor core count
mem: "8" #RAM in GB
disk: "200" #storage in GB
Package: " https://drive.google.com/drive/sample.zip"
SCM:
URL: " https://github.com/developer/Node1.git"
Pre-Stage: " Node1 /init_env.sh"
Main-Stage: "Node1 / install.sh"
Post-Stage: "Node1 / run.sh"

Fig.8 SBAT for a Deploy Sandbox

4 IMPLEMENTATION

For the purpose to implement our DEVOPSER, first, we have to implement a parser which has the capacity of analyzing these SBAT and generates the specification to HOT format. Then, uploading these generated HOT to the OpenStack to approach automated deployment. Following sections will describe the algorithms and the functionalities of the parsers within the DEVOPSER:

4.1 DEV. PARSER

Algorithm 1. Dev. Parser

Input:" Devinpt.yaml"
1. data= yaml.load(Input)
2. template = yaml.load(raw_template)
3. **if CheckInputFormat**(data)==True:
4. **ParseInstance** (data, template)
5. **if** ('SCM' in data.keys()):
6. **if CheckStage**(data) ==True:
7. **ParseSCM**(data.template)
8. **Output** = yaml.dump(template)
9. **end**

Algorithm1. illustrates the way to parsing the input (SBAT for Develop Sandbox) to the HOT format. **ParseInstance()** parses the machine deployment of the input and translate into the description in HOT format. **ParseSCM()** parses "SCM" in the input and translate into the description which enables us to perform the script for indicated machine in HOT format.

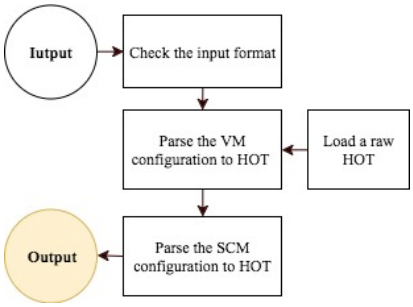

Fig.9 Flow chart of the Dev. PARSER

4.2 BUILD PARSER

Algorithm 2. Build Parser

Input:" Buinpt.yaml"

```
1.  data = yaml.load(Input)
2.  template = yaml.load(Build template)
3.  if CheckInputFormat(data)==True and
    CheckPreStage(data) == True:
4.      ParseInstance (data, template)
5.      if CheckStage(data) ==True:
6.          ParseSCM(data.template)
7.          Output = yaml.dump(template)
8.  end
```

Algorithm 2. shows the way to parse the SBAT to a HOT for the Build Sandbox. The process in this algorithm is almost the same as Algorithm 1. except there is a condition added for detecting whether the user specifies the "Pre-Stage", as we request a specification in the "Pre-Stage" to ensure that we can successfully build a CI server that enables Jenkins.

4.3 DEP. PARSER

Algorithm 3. Dep. Parser

Input:" depinpt.yaml"

```
1.  data = yaml.load(Input)
2.  Rely Sort(data)      # Scheduling launching order of
    the VMs and checking whether there exist a deadlock
3.  If Rely Sort(data) return a result:
4.    For all VM in data do:
5.      ParseInstance (data, template)
6.      if CheckStage(data) ==True:
7.          ParseSCM(data.template)
8.          GiveOrder(template)    #To give the VM
    rely information and set up the wait condition
9.          Output = yaml.dump(template)
10. #Following specify the RelySort:
11. def RelySort(data):
12.   if there is no deadlock:
13.     Scheduling the data
14.     and return the launching order
15. end
```

Algorithm 3. shows the pseudo code to parse the SBAT for the Develop Sandbox. The way to parse a single machine specified in the input to HOT is the same as the above algorithm that we have mentioned. However, here, we must count the number of the machines, specifying each machine with one HOT, handling the launching order of machines, and detecting whether there exists a deadlock. In our algorithm; first, after loading the input, **RelySort ()** is performed to count the machines and detect the deadlock, if there is no deadlock, **RelySort()** will schedule them and return the result. Next, within the for loop, **ParseInstance ()** parses the specification of every machine into a single HOT. Finally, **GiveOrder()** specify the order in each machine with the scheduling result of **RelySort()**. We implement the process that machine can be launched by its order by specifying a wait condition syntax descried in HOT format on each machine.

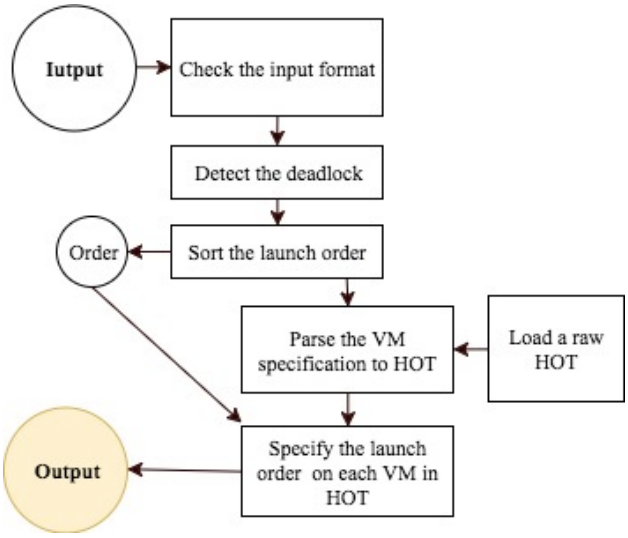

Fig.10 Flow chart of the DEPLOYMENT PARSER

5 EXPERIMENT

We perform an experiment on the parser in two step to ensure that the DEVOPSER can produce a valid HOT which satisfies with what we specified in the SBAT. Step1 is the functionality of the parser, the Step 2 is the validity of the HOT generated by the parser.

5.1 FUNCTIONALITY OF THE PARSER

As the Dev. Parser and Build Parser are basically the same, we experiment on each parser for its functionality with the test cases shown and explained as following:

- **DEV. PARSER and BUILD PARSER**

Fig11. shows a common mistake while specifying the SBAT with the use of invalid indentation. This kind of indentation conforms to the YAML format but dose not conform to our format.

```
Hostname: Dev1
Application: Python
OS: ubuntu14
Flavor:
cpu: 2
mem: 2
disk: 10
SCM:
URL:  https://github.com/aciculachen/simplescript.git
Pre-Stage: simplescript/test.sh
Main-Stage: simplescript/add.sh
```

Fig.11 Invalid Indentation on the document

- **DEP. PARSER**

In the verification for the Dep. Parser, we focus on two functionalities: *Deadlock Detection* and the *Scheduling,* as the methodology related to input format analysis and the

functionality that translate the specifications in the input into HOT format is the same for all parser in our work. If the Dep. Parser can handle the error and have no wrong in translation, so does the other parser.

```
Hostname: Node1
Rely on: Node2

Hostname: Node2
Rely on: Node1
```

Fig.12 A test case with a deadlock

We perform the functionality experiment with the test cases shown in **Fig.11**, **Fig.12**, and **Fig.13** on the Dep. Parser.

```
Hostname: java
Rely on: Node3
Application: Java
OS: ubuntu14
Flavor:
 cpu: 2
 mem: 2
 disk: 10
SCM:
 URL: https://github.com/aciculachen/simplescript.git
 Pre-Stage: simplescript/java.sh
 Main-Stage: simplescript/test.sh
 Post-Stage: simplescript/test.sh

Hostname: Node3
Application: Django
OS: ubuntu14
Flavor:
 cpu: 2
 mem: 2
 disk: 10
SCM:
 URL: https://github.com/aciculachen/simplescript.git
 Pre-Stage: simplescript/add.sh
 Main-Stage: simplescript/test.sh
 Post-Stage: simplescript/test.sh
```

Fig.13 A valid specification test case

5.2 VALIDITY OF THE HOT

A valid HOT must pass the following criteria:

(1) Passing the syntax parser on the Heat, it means that there is no syntax error on the HOT.

(2) Every VM (Virtual Machine) launched based on uploaded HOT is in accordance with its specification and is built successfully.

(3) Scripts given in "SCM" can be cloned to the machine, and "Pre-Script" script can perform automatically when the machine is just initiated. The script is used for customizing the VM, which is the key to approaching our DevOps story.

(4) If the user specifies that VM1 relies on the VM2, the VM1 start to launch if and only if the VM2 has been complete.

We perform the experiment to test the validity with the HOT generated by the Dep. Parser with the input sown in **Fig.13.** since this test case complies with the format we defined.

5.3 EXPERIEMENT RESULT

The result of the experiment on 5.1 shows that our parser can detect the semantic error in the input as shown in following:

(1) Invalid Indentation on specifying the "Flavor" and the "SCM":

```
aciciuladeMacBook-Air:generator aciciula$ pyth
[ERROR Format]Invalid Flavor setting
[ERROR Format]Please check your Stage config
```

Fig.14 The result of Fig.11 as the input

(2) A test case with a deadlock

```
aciciuladeMacBook-Air:generator aciciula$  python3 deploy_
find a deadlock
[ERROR Format]Hostname, Application, OS or Flavor is need.
```

Fig.15 The result of Fig12 as the input.

The experiment on the validity of the HOT is performed together with the functionality of Scheduling with the Fig.13 as the input. If the Scheduling function is work correctly, Dep. Parser generates a valid HOT theoretically. The Fig.13 shows a SBAT specifying a Deploy Sandbox. The "java.sh" on the specification for the "Pre-Stage" on the machine "java" is used to set up a JAVA Developer Environment on the "java", the "Main-Stage" script "add.sh" is used to add a user on the machine, and the "test.sh" is the script used to output a string on the screen. The content of the script in detail can be checked out with the URL specified in the "SCM". The scripts are all used to test whether the launched machine will automatically execute them or not. As a result, the content of them is not the key in our experiment.

After we take Fig.13 as an input for the Dep. Parser and produce a HOT. We upload this HOT to the OpenStack and check out the deployment result. First, we check out the launch order by see the "Stack Topology" page on the Horizon. The "Stack Topology" dynamically illustrated the process while the Heat launch a HOT. Then, we check out the build result on the "Instance" page on the Horizon. As Fig.15 shows, this Deploy Sandbox is launched by the HOT generated by our Dep. Parser with the SBAT seen in Fig.13 as the input. Finally, we compile a java file in the machine to verify whether the script used to set up a JAVA Developer Environment did its job or not as Fig14 shows.

	INSTANCE NAME	IMAGE NAME	IP ADDRESS
☐	java	Ubuntu14.04-2015.3.0	192.168.111.79 Floating IPs: 10.113.4.18
☐	Node3	Ubuntu14.04-2015.3.0	192.168.111.77 Floating IPs: 10.113.3.238

Fig.15 The build result shown on the Horizon.

```
ubuntu@java:~/tmp/example/java$ pwd
/home/ubuntu/tmp/example/java
ubuntu@java:~/tmp/example/java$ ls
hi.java
ubuntu@java:~/tmp/example/java$ javac hi.java
ubuntu@java:~/tmp/example/java$ ls
HelloWorldApp.class  hi.java
ubuntu@java:~/tmp/example/java$ java HelloWorldApp
Hello World!
```

Fig.16 Compile a HelloWorldApp in JAVA on the machine "java".

6 CONCLUSION

The experiment results seen in section 5 shows that the DEVOPESER enables automated deployment for the Sandboxes used to implementing a DevOps story. Each Sandbox can be deployed with the configurations documents that we defined and each Sandbox can be launch by the OpenStack with the HOT generated by our DEVOPSER.

In the future, we will find a developing project to perform the complete DevOps process with those Sandboxes that is deployed by the SBAT specified the sandbox of that project. On the other hand, we will finish the complete task of the build stage by implement the Jenkins on the build machine after we find out a application project.

REFERENCES

[1] M. Hüttermann, DevOps for Developers., 2012

[2] D. Chapman, Introduction to DevOps on AWS, 2014

[3] "DevOps is Agile for the Rest of the Company", DevOps.com.

[4] P Debois ,Agile Infrastructure and Operations: How Infra-gile are You?, Agile 2008, 2008

[5] Jenkins User Handbook, https://jenkins.io/user-handbook.pdf

[6] P. M. Duvall, S. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk,2007

[7] B. Ingerson, C. C. Evans, O. Ben-Kiki Yet Another Markup Language (YAML) 1.0 ,Working Draft",10 Dec 2001.

[8] OpenStack: Toward an Open-source Solution for Cloud Computing, Sefraoui, Omar; Aissaoui, Mohammed; Eleuldj, Mohsine. International Journal of Computer Applications; New York55.3 ,2012.

[9] Rakesh Kumar , Neha Gupta , Shilpi Charu , Kanishk Jain , Sunil Kumar Jangir, Open Source Solution for Cloud Computing Platform Using OpenStack, International Journal of Computer Science and Mobile Computing, 2014

[10] Qusay F. Hassan ,Demystifying Cloud Computing, The Journal of Defense Software Engineering.,2011.

[11] Development resources for OpenStack clouds, developer.openstack.org

[12] Heat Orchestration Template Guide,openstack.org

[13] J. Holck and N. Jørgensen, CONTINUOUS INTEGRATION AND QUALITY ASSURANCE: A CASE STUDY OF TWO OPEN SOURCE PROJECTS, Australasian Journal of Information Systems,2003

[14] S Sharma, DevOps For Dummies, IBM,2014