# CloudTUI-FTS: a user-friendly and powerful tool to manage Cloud Computing Platforms

Massimo Canonico and Davide Monfrecola
Department of Science and Innovation Technology (DISIT)
University of Piemonte Orientale
Italy
massimo.canonico@uniupo.it, davide.monfrecola@gmail.com

## ABSTRACT

The NIST defines Cloud Computing as a model for enabling ubiquitous network access to a shared pool of configurable computing resources. Thanks to the popularity of Cloud Computing and its various area of applicability, in the last years various projects have been realized for building Cloud systems (e.g., Amazon Web Services, Nimbus, OpenStack, Eucalyptus, and Microsoft Azure, just to name a few). Unfortunately, most of these projects (especially the open source ones) have some drawbacks: the user interfaces are not user-friendly, the basic tasks are complex to setup and to configure even for users with computer science skills. In this paper we present CloudTUI-FTS: a powerful and user-friendly tool able to easily interact with different Cloud platforms. In particular, with CloudTUI-FTS the user can perform both basic tasks (e.g., start-up/shut-down a service) and advanced tasks (e.g., create policies and mechanisms to prevent faults and to provide service scalability). We evaluate the effectiveness of our tool on the CloudLab infrastructure.

## Keywords

Cloud Computing systems, user interface, fault tolerance, scalability

## CCS Concepts

•**Networks** → **Cloud computing;**

## 1. INTRODUCTION

In the last years only few research areas related to computer science have had a rapid evolution such as *Cloud Computing* [8]. Indeed, most of the services that we use everyday (e.g., webmail, social network, web search, etc.) run over a Cloud Computing platform. These services are widely available and most of them are very easy to use. Unfortunately, this simplicity has nothing to do with the user interface that the Cloud administrators has to use in order to provide these services. Indeed, even for a simple task such as starting-up a service (which corresponds to start up a new *Virtual Machine* (VM)), the user has to configure/manage various aspects: (i) the VM image type, (ii) the VM system memory occupancy, (iii) the VM disk occupancy, (iv) all network settings (e.g., IP allocation, netmask, gateway), (v) all security settings (e.g., firewall configuration, certificate credentials) just to name a few.

(a)
```
euca-run-instances -k mykey -n 1 emi-0B951139
-t c1.medium
```

(b)
```
./bin/cloud-client.sh --run --name hello-cloud
--hours 2
```

(c)
```
nova boot --flavor m1.small --image
"images/ubuntu-12.04" --key_name $USER-key
$USER-001
```

**Figure 1: Launch a new service/VM with (a) Eucalyptus API, (b) Nimbus API and (c) OpenStack API.**

The scenario is even more complex if the user has to manage various Cloud Computing platforms. Indeed, despite of the effort put by the *Open Cloud Computing Interface* (OCCI) [12] in trying to create a common and flexible API for each Cloud computing system, any Cloud platform has its own procedure, syntax and terminology (e.g., a running VM is called *instance* or *workspace* depending on which Cloud platform is being used). For example, in Fig. 1, we reported the commands to start-up a new VM (one of the most common task in any Cloud platform) in three of the most important open source Cloud platforms: *Eucalyptus* [10], *Nimbus* [5] and *OpenStack* [4]. Due to space constrain, we cannot provide more examples showing how the terminology, usability, syntax and also the semantic of the same task can be significantly different between the various Cloud solutions proposed in the last years.

With *CloudTUI-FTS* it is possible to interact with various Cloud Computing platforms by using a unique user-friendly and powerful textual interface. Thanks to CloudTUI-FTS, the user can easily complete basic tasks (e.g., service starting-up/shutting-down) and also complex tasks (e.g., create policies and mechanisms to prevent faults and to provide ser-

VALUETOOLS 2015, December 14-16, Berlin, Germany

DOI 10.4108/eai.14-12-2015.2262718

vice scalability). These characteristics are included in the acronyms we used in the name of our tool: *Text User Interface* (TUI), *Fault Tolerant*(FT) and *Scalable*(S).

CloudTUI-FTS is an evolution of our previous project called CloudTUI [6] with many new features such as: (i) a new rule engine able to make decisions based on knowledge base facts; (ii) a new monitoring module able to manage a wide range of parameters concerning the VM health, (iii) an improved textual interface including new options for the users and, finally, (iv) a complete rewriting of the code in order to ease the multi-platform management.

In the rest of this paper, we describe in detail the main features of the CloudTUI-FTS. In particular, in Section 2 we discuss the related works, while in Section 3 we present the CloudTUI-FTS tool by describing its architecture and its main components: the *Monitor* (Section 3.1, the *Rule Engine* (Section 3.2), the *Agent* (Section 3.3) and the *Manager* (Section 3.4). In Section 3.5 we describe how the textual interface works, while in Section 4, we describe the experimental evaluation of our tool. Finally, in Section 5, we discuss the conclusions and the future works concerning our project.

## 2. RELATED WORKS

As mentioned before, Cloud Computing is having an enormous success in the last years. This is due to the fact that this model has introduced new opportunities in the way the services can be provided, but this comes with many complex aspects to manage. Indeed, once the platform is up and running, the Cloud administrator has to figure out how to optimize the service provisioning. In order to do that, the Cloud administrator has two different user interfaces: a textual and/or a web interface depending on the Cloud platform.

The problem with textual interfaces regard mainly the syntax. As showed in Fig. 1, even a simple task such as starting a VM requires a lot of parameters that are different from a Cloud platform to another. In particular, these parameters are often a sequence of number and letters (i.e., IDs) which the user has to remember and to use many times, in order to run a service. The textual interfaces do not help out the user since these IDs are not stored anywhere. This is just one example regarding the limitations of the textual interfaces that affects all major open source Cloud platforms such as the *euca tools* provided by Eucalyptus [10], the *cloud-client* [5] provided by Nimbus and the *nova tool* provided by OpenStack [4], just to name a few.

With the web interfaces, the user does not have to remember the IDs as for the textual interfaces thanks to lists and drop-down menu that make easier the user selections. The drawbacks of the web interfaces regard their limitations. For example, *Horizon* [1] is the web interface implemented in OpenStack and it is one of the most complete web interface provided by an open source Cloud Computing platform. Unfortunately, Horizon does not provide any reliable mechanism to prevent service faults or to manage service scalability. For example, the only fault tolerant mechanism available in Horizon consists in sending ping command to a VM and if the VM does not reply, OpenStack considers the VM down. Of course, this mechanism may generate many false positives.

Finally, to the best of our knowledge, none of the interfaces both textual and web have been developed to be multi-platform. A project who claimed to be multi-platform was *Phantom* [2]. Unfortunately, the project has never reached an acceptable level of robustness and has been dismissed since one year ago and also the web site of the project is no longer available.

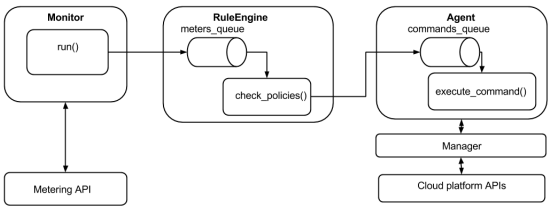


**Figure 2: The CloudTUI-FTS architecture.**

## 3. THE CLOUDTUI-FTS TOOL

In Fig. 2, we show the architecture of CloudTUI-FTS with its four main components: the Monitor, the Rule Engine, the Agent and, finally, the Manager. In this section we describe how they work.

### 3.1 The Monitor component

The Monitor component must collect information concerning the *health* of each VM, where with health we mean how the VM is using the resources such as CPU, memory, network, etc. For example, if the percentage of CPU used by a VM is closed to 100%, it probably means that this VM is receiving too much work. In this case, it could be necessary to start up a clone of the VM (that is, a VM able to provide the same service) in order to split up the work between both VMs.

The Monitor component periodically retrieves information from the VMs running in the Cloud platform: the retrieval frequency, the number of parameters to retrieve and many other aspects of the Monitor component can be customized. Also, the API used to retrieve information are customizable and can be specified in the *Metering API* module. Thanks to this module, the Monitor component can be independent with respect to the monitor tool used. For example, OpenStack uses *Ceilometer* [3] as preferred monitoring tool, while Eucalyptus uses a custom version of the *Amazon CloudWatch* [11]. Our Monitor component is compatible with both solutions. Moreover, the Monitor component is able to detect outliers values so they are not stored into the *meters_queue*. The samples stored in the *meters_queue* are used by the Rule Engine component as described in the following section.

### 3.2 The Rule Engine component

All data collected by the Monitor component are sent to the Rule Engine component in its *meters_queue*. The Rule Engine reads the data from this queue and it has to decide if some action must be taken. More specifically, the Rule Engine has a set of rules where each of them are composed by two parts: the *condition* and the *action*. The Rule Engine reads the data from the *meters_queues* and checks if the condition is true, and, if so, it performs the action described in the rule. For example, a rule can be defined as the following one: *"if the CPU utilization is higher than 80% then clone the VM"*. By cloning a VM (that is a service), the workload can be split up between more VMs in order to

```
1  import logging
2
3  from intellect.classes.Resource import Resource
4
5  threshold = 0.1
6
7  rule "test load":
8          agenda-group cpu
9          when:
10              $resource := Resource( get_sample("cpu_util")>threshold)
11         then:
12              attribute cpu_util = $resource.get_sample("cpu_util")
13              $resource.clone()
```

**Figure 3: A Intellect rule**

```
1  def run(self, meters_queue):
2          self.init_resources()
3          self.read_policies()
4          logging.info("Rule engine initialization completed")
5
6          while self.signal:
7              try:
8                  element = meters_queue.get()
9
10                 self.resources[element["resource_id"]].add_sample(meter=element["meter"],
11                                                                   value=element["value"],
12                                                                   timestamp=element["timestamp"])
13                 self.check_policies()
14
15 def clone(self):
16         self._command_queue.put({'command': 'clone', 'resource_id': self._resource_id})
```

**Figure 4: The main functions of the Rule Engine component**

decrease the service time. Of course, the rule can be more complex by including different conditions related to various sample parameters.

This component has been realized by exploiting *Intellect* [13]: a *Domain-specific language* (DSL) and a rule manager for Python. The rules manager provides an intellect, a form of artificial intelligence, a faculty of reasoning and understanding objectively over a working memory. Fig. 3 illustrates a rule by using the Intellect syntax. In particular, we can note the *when* block (lines 9-10) where the conditions to check are specified and the *then* block (lines 11-13) where the action to take (if the conditions are satisfied) is specified. In particular, the example code proposed in Fig. 3 sets a threshold (line 5) and then it checks if the CPU utilization is higher than the threshold set (line 10). If so, the resource has to be cloned (line 13) by executing the *$resource.clone()* method.

The Rule Engine component checks if a policy has to be implemented by using a specific method called *check_policies()*. As shown in Fig. 4, the Rule Engine *run* function wakes up when some element is present into the *meters_queue* (line 6), then it pulls the first element out from the queue (line 8) and, finally it extracts the information from the element such as the meter type (e.g., CPU utilization, bandwidth utilization, memory occupancy, etc.), the value and the timestamp when the sample was stored (lines 10-12). Once all information has been retrieved, the *check_policies()* method can check if there is any rule satisfied. The actions to take are put into the *commands_queue* (line 16). The elements in this queue will be used by the Agent component as described in the following section.

## 3.3 The Agent component

The Agent component has to translate the action decided by the Rule Engine component into the specific API depending on the Cloud platform in use. As mentioned in Section 2, each Cloud platform has its own API and a user has to remember the correct syntax and semantic in order to implement what the Rule Engine component has decided.

```
1  def run(self, cmd_queue):
2          while self.signal:
3              try:
4                  command = cmd_queue.get()
5                  self.execute_command(command)
6
7              except Exception, e:
8                  logging.error("Error %s:" % e.args[0])
9
10
11 def execute_command(self, command):
12         action_method = self.get_action_method(command['command'])
13         action_method(command['resource_id'])
```

**Figure 5: The main functions of the Agent component**

```
1  def clone_instance(self, instance_id):
2
3              if self.networks is None:
4                  self.networks = self.nova.networks.list()
5
6              nics = []
7              for net_id in instance._info['addresses'].keys():
8                  for network in self.networks:
9                      if network.label == net_id:
10                         nics.append({"net-id": network.id})
11
12             security_groups = []
13             for security_group in instance._info['security_groups']:
14                 security_groups.append(security_group['name'])
15
16             instance = self.nova.servers.create(name=instance.name + "-clone",
17                                                 image=instance._info['image']['id'],
18                                                 flavor=instance._info['flavor']['id'],
19                                                 key_name=instance._info['key_name'],
20                                                 security_groups=security_groups,
21                                                 nics=nics)
```

**Figure 6: The Manager component: the function to clone a VM**

Fortunately, the CloudTUI-FTS user has nothing to remember since the Agent component is able to compose the right sequence of APIs to execute. After that, the Agent sends the sequence to the Manager component which is in charge to actually run it. In Fig. 5 an extract of the Agent component source code is showed. As other components, the Agent has a *run()* method (line 1) which contains the main function. First of all, the Agent has to extract the next command in the the *cmd_queue* (line 4) and then it has to execute it (line 5) by invoking the generic *execute_command* method (line 11). This method is able to retrieve the type of command to execute (line 12) and then it starts the procedure by passing the input parameters required (line 13). As discussed in the next section, the Manager is in charge of actually running the command with the input parameters on the Cloud platform selected.

## 3.4 The Manager component

The Manager component is specific for a Cloud platform since it is in charge of running the API specialized commands received by the Agent component. The Manager runs each command, once a time, and it notifies the user if the command succeeded or not. In Fig. 6, we can see the source code related to the VM clone procedure specific for OpenStack Cloud platform. In order to clone a VM, it is necessary to retrieve all information concerning the original VM. Indeed, the *clone_instance()* method retrieves information concerning the network configuration (lines 3-10), the security group (lines 12-14) and, finally, all VM attributes such as the image ID (line 17), the image size (line 18) and the key pair ID (line 19). Once all is set, the *self.nova.servers.create()* method can be invoked in order to create a clone of the VM.

## 3.5 CloudTUI-FTS textual interface

For our tool, we decided to implement a textual interface instead of a graphic or a web one since it is efficient, it does

not require any particular library or hardware installed to run and, last but not least, it can be used on a remote terminal (this is very important since in some Cloud platform the user has to login into a remote *controller machine* to manage the system).

After that, the user has to decide which task to perform by selecting one of the following options: the user can create a new instance (Option 1), can require the list of the running VMs (Option 2), can reboot a VM (Option 3) and so on. By selecting the Option 2, the user has to decide many aspects of the instance to create: (i) the image operating system (e.g., Ubuntu, Fedora, Windows, and so on depending on which operating system images have been stored in the Cloud platform), (ii) the image size (which defines sizes for RAM, disk, number of cores, and so on, (iii) the secure group (i.e., which firewall rules have to be activated), (iv) the network setup (i.e., how the instance will request an IP address), and (v) the public key to use (that is necessary to log in to the instance). After that, a new instance will be created and the user will be notified when it will be up and running. Due to the space constrains, we cannot describe in details all the tasks provided by CloudTUI-FTS. The interested reader can find more details in the project web site [7].

# 4. CLOUDTUI-FTS EXPERIMENTAL EVALUATION

In order to evaluate the reliability and the performance of our tool we have conducted several experiments with various users. Since our tool needs a Cloud platform up and running, we decide to use the resources provided by *CloudLab* [9]. CloudLab is a scientific infrastructure for research regarding Cloud Computing where researchers can build their own Clouds. CloudLab comprises approximately 5,000 cores and from 300 to 500 Terabytes of storage in the latest virtualization-capable hardware.

CloudTUI-FTS has being used by CloudLab users and computer science students during the distributed system course in our university. Note that our students have not any specific background in Cloud Computing and they were able to play with different Cloud platforms achieving the basic tasks (e.g., start a service) and also the advance tasks (e.g., create a "clone" policy). In the last six months we have fixed the bugs pointed out by the users and we can claim that CloudTUI-FTS is reliable and user-friendly as we expected to be. CloudTUI-FTS software is an open source project and it is freely available for download [7].

# 5. CONCLUSIONS AND FUTURE WORKS

In this paper we have presented CloudTUI-FTS: a textual interface able to interact with various Cloud platforms allowing the user to perform basic and advanced tasks without any preliminary skills on Cloud Computing. Our tool has four main components in order to make all procedures modular and easy to expand/improve/customize. An intense experimental evaluation performed by a large user community proofed that CloudTUI-FTS is reliable and user-friendly. Our tool is open source and freely available for everyone who wants to start studying Cloud Computing or who just wants a better interface with respect to what Cloud platforms usually provide.

CloudTUI-FTS is an on-going project and we are implementing new features. Currently, we are testing a new component called *Logger* which is in charge of creating various log files in order to track the behavior of any components in our tool. The experiments done reveal that the Logger is almost ready to be released with the next version of CloudTUI-FTS. The preliminary code of this component can be found in a specific branch of the repository of the project [7]. Since to the best of our knowledge there is not a web platform able to interact with many Cloud platforms concurrently, we are planning to implement a web version of our tool. Finally, as mentioned before, CloudTUI-FTS is able to interact with various monitoring tools such as Ceilometer and CloudWatch. Even if these tools work well, they are quite complex to install, to configure and to use. For this reason, we are planning to make the Monitor component more general. The user just needs to implement a method called *getMeter()* and the Monitor component will invoke this method to retrieve the new meter value.

# 6. ACKNOWLEDGMENTS

We thank Dr. S. Garione for its contribution on user input error handling and on public IP allocation procedure.

# 7. REFERENCES

[1] Horizon: The OpenStack Dashboard Project. goo.gl/21cQX0. [Online; accessed 2-Nov-2015].

[2] Nimbus Phantom. http://goo.gl/2SMxGS. [Online; accessed 2-Nov-2015].

[3] OpenStack Telemetry (Ceilometer). https://goo.gl/cnDZt5, 2015. [Online; accessed 2-Nov-2015].

[4] K. Jackson. *OpenStack Cloud Computing Cookbook*. Packt Publishing, 2012.

[5] K. Keahey. Nimbus: Open source infrastructure-as-a-service cloud computing software. In *Workshop on adapting applications and computing services to multi-core and virtualization*, 2009.

[6] M. Canonico et al. Cloudtui: a multi cloud platform text user interface. In *VALUETOOLS*, pages 294–297. ICST/ACM, 2013.

[7] D. Monfrecola and M. Canonico. The CloudTUI-FTS project. goo.gl/UNi47r. [Online; accessed 2-Nov-2015].

[8] P. Mell et al. The NIST Definition of Cloud Computing. goo.gl/ghvWBH, 2015. [Online; accessed 2-Nov-2015].

[9] R. Ricci et al. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *login*, 39(6):36–38, 2014.

[10] R. Wolski et al. The eucalyptus open-source cloud-computing system. In *Proc. of the 9th IEEE/ACM Int. Sym. on Cluster Computing and the Grid*, CCGRID '09, pages 124–131. IEEE Computer Society, 2009.

[11] M. Ryan. *AWS System Administration: Best Practices for Sysadmins in the Amazon Cloud*. O'Reilly Media, Inc., 1st edition, 2015.

[12] The Occi project. An Open Community Leading Cloud Standards. www.occi-wg.org, 2015. [Online; accessed 2-Nov-2015].

[13] M. J. Walsh. Intellect: A Domain-specific language and Rules Engine for Python. goo.gl/vzK5bH, 2015. [Online; accessed 2-Nov-2015].