Towards Autonomic Cloud Configuration and Deployment Environments

Bill Karakostas
VLTN GCV
Terninckstraat13/208
2000 Antwerp
Belgium
bill.karakostas@vltn.be

Abstract— Cloud configuration deployment and management is still largely a manual task carried out by system administrators. Introducing autonomy in Cloud management would entail, amongst other things, the ability for the Cloud manager to automatically scale up or down the number and type of deployed images/virtual machines, to meet SLAs for performance etc. In this paper we present *autoJuJu*, a prototype autonomic cloud manager built on top of Juju, a Cloud service orchestration and deployment manager for the Ubuntu Linux OS. *AutoJuJu* makes autonomous decisions about when to scale Cloud services horizontally (by adding or removing instances) and vertically (by deploying different architectural components) to improve performance. We show how this autonomic Cloud manager can self configure and optimise a Cloud deployment.

Keywords— Ubuntu; Juju; Linux Virtual Containers; autonomic computing; cloud deployment and management

I. INTRODUCTION

The effort required to deploy large software systems has long been underestimated, compared to the development effort, and only now has started to receive due attention. In recent years, software configuration and deployment has become more automated. Many software tools and environments that automate administrative tasks such as installing packages and modules, defining authorisations and updating configurations, have been developed. New software tools, like the Chef [2] infrastructure configuration and management framework, for instance, facilitate the work of administrators, providing higher-level languages. Still, as systems become larger and more decentralised (e.g. Cloud based) there will be more need to automate more configuration and deployment activities. Self configuring systems are one of the main goals of autonomic computing. Autonomic systems are expected to absorb the complexity of usually manual administrative tasks and provide intuitive, high-level interfaces for human administrators. [7]

Deployment of software on a Cloud environment is currently the responsibility of system operators (SysOps), and although tools are provided by the Cloud vendors, significant manual effort is required. Management of Cloud deployments is therefore a crucial feature, which needs to be automated and integrated with intelligent strategies for dynamic provisioning of resources in an autonomic manner. [1]

New flexible deployment systems like Chef allow the desired configuration requirements to be specified in a configuration file in a declarative manner. However, even such systems require SysOps to manually reconfigure deployed systems. To increase productivity in Cloud, deployed systems should be able to self-manage their configurations and dynamically reconfigure as the environment changes, by for example, scaling up or down, or deploying machines with different hardware/software profiles.

This paper presents such as an autonomic Cloud manager that is implemented on top of an existing Cloud service orchestration manager, Juju. This autonomic manager uses high level policies and rules to determine when to scale horizontally (by increasing or reducing the number of deployed machines) a deployment, and also when to scale vertically (by introducing a different type of architecture) to meet performance and other SLA requirements.

The paper is structured as follows. Section II surveys existing work in autonomic computing and Cloud deployment environments. Section III describes the architecture of the autonomic Cloud manager prototype. Section IV presents experiments showing its behaviour in a Cloud deployment. Finally, section V discusses further research needed to make autonomic Cloud management systems capable of managing real life Cloud deployments..

II. STATE OF THE ART

A. Cloud System Management

The principles of autonomic computing [5] have been applied to the design of web service architectures and subsequently, to service deployment and management on the Cloud [12]. Several types of autonomic capabilities for web service based systems have been proposed, for example, in [13], [14], for web service composition in [10] and for an autonomic service bus in [9].

B. Autonomic Service Management

Architectures for autonomic service systems have also been proposed. For example, [4] proposes such an architecture for the management of Web services, where each autonomic element maintains a self-representation which embodies the component's current goal settings and its current performance statistics. Updates made to the component's self-representation trigger changes to the actual system, and also, if necessary, changes are made to the component's self-representation in order to reconfigure it.

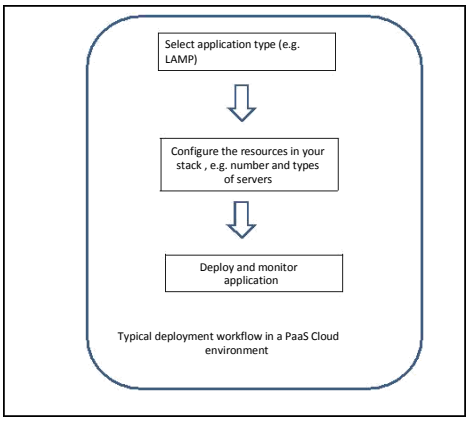

Fig. 1. Typical non-autonomic Cloud deployment workflow.

C. Policy-Based Cloud Management

[3] proposes policy-based management of autonomic web service systems. Policies represent the high-level service objectives and operation control logics that can determine the behaviour of managed systems. The promise of policy-based Cloud management is that the operation of computing/networking resources can be guided to follow certain rules, and dynamically configured so that they can achieve certain goals, derived from SLAs and SLOs.

III. *AUTOJUJU*:AN AUTONOMIC CLOUD CONFIGURATION AND DEPLOYMENT MANAGER

A. Introduction to JuJu

Service deployment is a key system management functionality [8]. In general, deploying a distributed service oriented application means configuring and running a set of services in one or more machines to compose an integrated system. However, this requires not only configuring the processes for particular needs, but also appropriately orchestrating the services that compose the system. In a Cloud deployment, both the processes and the machines that they will run on need to be determined, at some stage.

Juju is a Cloud configuration, deployment and monitoring environment that can be used to deploy services across multiple cloud or physical servers and orchestrate those services.

The activities within a service deployed by Juju are orchestrated by a juju *charm*, which is a deployable service or application component [6].

JuJu already has some autonomic features: for example, it internally tracks the state of units and their relations. It keeps track of their lifecycle by using state transition models. If the deployment of a unit fails, then Juju notes this failure, and transitions either the unit or the unit relations to other units to a failure state. However, JuJu cannot automatically recover from such failure states. It is expected that a SysOp will have a look at the system to determine or correct the issue, before removing the error block after correction of the original issue. Recovering from failures ('healing') therefore is an autonomic feature that would be useful to JuJu as well as to other Cloud deployment environments. However, in the following section we show how Juju could become even more autonomic by addressing other autonomic aspects such as self-configuration and self-optimisation.

B. Architecture of AutoJu

autoJuJu is a prototype autonomic Cloud deployment manager that we have developed on top of JuJu. It has been inspired by configuration managers such as Chef [2] and, like Chef , it is implemented in the Ruby language. As shown in Figure 2, *autoJuJu* communicates with JuJu through JuJu's command line interface (CLI). *AutoJuJu* monitors the status of the Cloud deployment through Juju's status reporting features and directly from the operating system using standard Linux commands like *uptime* and *top*.

AutoJuJu runs in a *sense-plan-act* control loop. It senses changes in the operating conditions of the Cloud deployment, decides whether a scaling up or down to the deployment is required, according to SLA derived rules, and finally acts by sending appropriate commands to Juju. *autoJuJu* uses therefore Condition-Action reactive rules to react to changes in the Cloud deployment such as system overloading, which can be dealt with by horizontal scaling (scaling up) or with scaling out actions, where the architecture needs to change.

IV. MANAGING CLOUD DEPLOYMENTS WITH *AUTOJUJU*

For the experiments, we deployed a Cloud on a single machine by utilising the Linux LXC virtual containers (LXC). LXC is a lightweight virtualisation technology that allows the deployment of independent Linux installations on the same host machine. As JuJu supports other Cloud architectures like OpenStack, our approach is portable. Due to resource limitation on the host computer, we configured Juju to deploy up to a maximum of 10 LXC containers, ('machines'), using standard Ubuntu Linux O/S images. As all machines were identical, we did not use rules or constraints to define policies for machine configuration such as minimum or maximum number of CPUs, RAM etc.

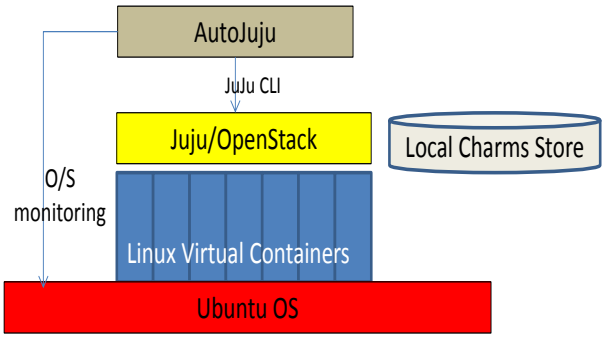

Fig 2: Architecture of the Autonomic Cloud manager

To test the behaviour of *autoJuJu*, we deployed a single Wordpress application. Wordpress is an open source blogging application and one of the locally available *charms* in the *charms* store of Juju. *AutoJuJu* maintains an internal database of the services (‘charms’) available in the local charms store, together with their properties and dependencies. It populates the database by reading the metadata files of the available charms. For example, *autoJuJu* describes the Wordpress deployment as follows

```
{"wordpress" => {"charmtype" => "blogging tool",
                 "mandatoryrelations" => ["db"],
"scalerelations" => ["reverse proxy", "monitoring service"]}}
```

Effectively, as shown in the above example, *autoJuJu* maintains information about the type of the Wordpress service (a blogging tool), the types of mandatory and optional components/services that the deployment requires (a database) and the types of services it requires for scaling the deployment (a reverse proxy and a monitoring service).

To initiate a new deployment *autoJuJu* creates a new JuJu environment and instantiates a new deployment object as per Figure 3.

```
irb(main):002:0> mydeployment=JuJuDeployment.new
=> #<JuJuDeployment:0x000000020fb2f0>
irb(main):003:0>
```

Fig. 3. Starting a new cloud deployment

As shown in Figure 4, a new service (the Wordpress application) is then added to the new deployment.

```
irb(main):022:0*
irb(main):023:0* mydeployment.deploy('wordpress')
```

Fig. 4: Deploying the wordpress service

As *autoJuJu* has knowledge of service dependencies, when a service like Wordpress is added to the deployment, all its default dependencies and their relationships to Wordpress are also loaded automatically. This is shown in Figure 5.

```
irb(main):003:0> mydeployment.startautojuju('wordpress')
resolving dependencies...
deploying wordpress
deploying supporting services...
deploying if not already deployed: mysql
exposing wordpress
```

Fig. 5: Automatic deployment based on service dependencies.

Automatic loading of all mandatory dependent services is a feature currently missing from standard Juju, which requires that all dependent services and relationships must start manually by the SysOp. Figure 6 shows a part of the current status of the deployment report which is obtained directly from JuJu through the CLI.

```
services:
  juju-gui:
    charm: cs:precise/juju-gui-81
    exposed: true
    units:
      juju-gui/0:
        agent-state: started
        agent-version: 1.16.5.1
        machine: "2"
        open-ports:
        - 80/tcp
        - 443/tcp
        public-address: 10.0.3.213
```

Fig. 6. Showing the status of the deployment

```
irb(main):003:0> mydeployment = JuJuDeployment.new
=> #<JuJuDeployment:0x00000001b4be90>
irb(main):004:0> mydeployment.startautojuju('wordpress')
resolving dependencies...
deploying wordpress
deploying supporting services...
deploying if not already deployed: mysql
exposing wordpress
adding more units to wordpress
```

Fig. 7 scaling a deployment horizontally

```
def monitordeployment(charm)
 u=Usagewatch.uw_load #obtain system load from the O/S if u >
 0.9
      puts 'adding more units to ' + charm
      wait(CoolDown)
      system('juju add-unit '+ charm)
     unless JuJuDeployment.how_many_instances(charm) > 10

end
end
```

Fig. 8 rule for scaling up deployment

As shown in the code snippet of Figure 8, the current prototype of *autoJuJu* adds more units (service instances) to the deployment if it senses the system load to exceed a certain value (0.9). *autoJuJu* adds a new machine for each new service instance (up to a user defined limit of machines), which is not always the most economical approach. At the expense of more complicated housekeeping, this can be altered by configuring *autoJuJu* to hosting multiple instances of a service on the same machine As it does not know the exact number of units it has to add, *autoJuJu* keeps adding instances, one at a time until the 15 minute average system load index falls below 0.9. There is a built-in delay (a CoolDown constant similar to the one used in Amazon's AWS) that prevents the resource allocator from making any changes to the system for that amount of time. The motivation behind this is to avoid frequent creation/termination of instances when the workload exhibits high variability, as this would have a negative impact on the cost of the service [15]

Fig. 9: Vertical scaling of a deployment

Vertical scaling policies are not independent from horizontal scaling ones. As the number of deployed instances increases, workloads need to be distributed equally amongst all deployed instances. *autoJuJu* is capable of scaling vertically by maintaining a record of the Juju charms that can be used to scale vertically, for example the following structure

"scalerelations" => ["reverse proxy", "monitoring service"]

defines that there are two service types that you need to deploy in order to scale vertically: a reverse proxy and a monitoring service. Using this knowledge, *autoJuJu* can search in the charms store for existing services that match the above service types.

AutoJuJu, therefore, utilises policies for vertical scalability that depend on the state of horizontal scalability, and then finds and deploys charms of suitable type. In the Wordpress example, *autoJuJu* will find *HaProxy*, a load balancer/reverse proxy in the charms store and deploy it when the number of deployed Wordpress instances exceeds 1 (Figure 9). Finally, scaling down the deployment is a functionality supported by JuJu, that, however, has not been implemented in the current prototype of *autoJuJu*.

V CONCLUSIONS AND RECOMMENDATIONS

In the future, as Cloud deployments become larger and more complex we expect more autonomic features to be introduced as bolt-on existing Cloud managers, and also as standalone ones. The current version of *autoJuJu* is a proof of concept work that autonomic features can be bolted on to existing configuration and deployment environments. *AutoJuJu* however employs simple detect and respond policies and rules that can be improved in several ways. For example, *autoJuJu*'s response strategy could become proactive rather than reactive: Instead, for example, of reacting when the system load exceeds a constant, response should gradually start when the load exceeds a lower threshold, in order to reduce lag. Also, *autoJuJu's* architecture could become more distributed with many running instances of *autoJuJu* deployed on different nodes, as, at the moment, it is centralised and represents a performance bottleneck and a single point of failure.

Finally, more sophisticated policies could be implemented and tested in *autoJuJu* that consider other architectural and performance *characteristics* of the deployed application. More OS probes could be utilised to measure different system characteristics such as file and network I/O.

We believe that all autonomic capabilities are important but in particular *healing* should be a core autonomic capability of the Cloud deployment, but not necessarily visible to, or controlled by, the end user (although logs of the healing actions should be made available to SysOps). Self configuration and optimisation functions should however be visible to and controlled by end users/customers via policies. Similar to Service Level Objectives (SLOs), end users should be able to specify such policies using high-level business terminology. Possibly, a Domain Specific Language (DSL) could be defined for this purpose [11]. Next, such high level SLOs would need to be translated into lower-level policies for the autonomic Cloud manager.

REFERENCES

[1] Buyya, Rajkumar , Rodrigo N. Calheiros, and Xiaorong Li "Autonomic Cloud Computing: Open Challenges and Architectural Elements". EAIT 2012.

[2] Chef http://www.getchef.com/chef/

[3] Cheng, Yu , Ramy Farha, Myung Sup Kim, Alberto Leon-Garcia, "A generic architecture for autonomic service and network management". Computer Communications 29(18): 3691-3709 (2006)

[4] Farrell, J. A. , and H. Kreger, "Web Services Management Approaches". IBM Systems Journal, 41(2), (2002)

[5] Huebscher Markus C. , Julie A. McCann "A survey of autonomic computing—degrees, models, and applications". ACM Computing Surveys (CSUR) Surveys Homepage archive Volume 40 Issue 3, August 2008

[6] Juju "Juju documentation" juju.ubuntu.com 2014

[7] Lalanda Philippe , Julie A McCann and Ada Diaconescu. "Autonomic Computing Principles Design and Implementation" Springer Verlag London 2013.

[8] Li, Wubin , Petter Sv¨ard, Johan Tordsson and Erik Elmroth "A General Approach to Service Deployment in Cloud Environments". Proceeding CGC '12 Proceedings of the 2012 Second International Conference on Cloud and Green Computing Pages 17-24

[9] Morand, Denis, Isaac Garcia, Philippe Lalanda. "Towards Autonomic Enterprise Service Bus". MAASC'11 - Workshop on Middleware and Architectures for Autonomic and Sustainable Computing, Paris : France (2011)

[10] Pautasso, Cesare , Thomas Heinis Gustavo Alonso "Autonomic Execution of Web Service Compositions". In Proc. ICWS 2005. Proceedings. 2005 IEEE International Conference on Web Services, 2005.

[11] Sledziewski, Krzysztof , Behzad Bordbar and Rachid Anane. "A DSL-based Approach to Software Development and Deployment on Cloud". 2010 24th IEEE International Conference on Advanced Information Networking and Applications.

[12] Sterritta,, Roy , Manish Parasharb,, Huaglory Tianfieldc,, Rainer Unland "A concise introduction to autonomic computing". Advanced Engineering Informatics 19 (2005) 181–187

[13] Tian, Wenhu Farhana H. Zulkernine, Jared Zebedee, Wendy Powley, Patrick Martin: "Architecture for an Autonomic Web Services Environment". WSMDEIS 2005: 32-44

[14] Tosi, Davide , Giovanni Denaro, Mauro Pezzè: "Towards autonomic service-oriented applications". Int. J. Auton. Comp. 1(1): 58-80 (2009)

[15] Jorge M. Londoño-Peláez, Carlos A. Florez-Samu "An Autonomic Auto-scaling Controller for CloudBased Applications" (IJACSA) International Journal of Advanced Computer Science and Applications,Vol. 4, No. 9, 2013