Managing Appliance Launches in Infrastructure Clouds

John Bresnahan
Mathematics and Computer Science Division
Argonne National Laboratory
bresnahan@mcs.anl.gov

Tim Freeman
Computation Institute
University of Chicago
freeman@mcs.anl.gov

David LaBissoniere
Computation Institute
University of Chicago
labisso@uchicago.edu

Kate Keahey
Mathematics and Computer Science Division
Argonne National Laboratory
Computation Institute
University of Chicago
keahey@mcs.anl.gov

ABSTRACT

Infrastructure cloud computing introduces a significant paradigm shift that has the potential to revolutionize how scientific computing is done. However, while it is actively adopted by a number of scientific communities, it is still lacking a well-developed and mature ecosystem that will allow the scientific community to better leverage the capabilities it offers. This paper introduces a specific addition to the infrastructure cloud ecosystem: the cloudinit.d program, a tool for launching, configuring, monitoring, and repairing a set of interdependent virtual machines in an infrastructure-as-a-service (IaaS) cloud or over a set of IaaS clouds. The cloudinit.d program was developed in the context of the Ocean Observatory Initiative (OOI) project to help it launch and maintain complex virtual platforms provisioned on demand on top of infrastructure clouds. Like the UNIX init.d program, cloudinit.d can launch specified groups of services and the VMs in which they run, at different run levels representing dependencies of the launched VMs. Once launched, cloudinit.d monitors the health of each running service to ensure that the overall application is operating properly. If a problem is detected in a service, cloudinit.d will restart only that service and any other service that failed that depended on it.

General Terms

Management, Design, Experimentation.

Keywords

Cloud computing, Infrastructure-as-a-service, Platform-as-a-service, Nimbus.

1. INTRODUCTION

Infrastructure-as-a-service (IaaS) cloud computing [1] (sometimes also called "infrastructure cloud computing") has recently emerged as a promising outsourcing paradigm that has been widely embraced commercially and is also beginning to make inroads in scientific communities. Infrastructure clouds allow users to exercise control over remote resources by introducing a virtualization layer that ensures isolation from the provider's infrastructure and thus a separation between provider's hardware and the user's environment. This feature proves particularly attractive to scientific communities where control over the environment is critical [2]. Furthermore, by providing on-demand access, cloud computing becomes an attractive solution to applications that are deadline-driven (e.g., experimental applications) or require urgent computing [3] capabilities.

Although many scientific projects are actively taking advantage of cloud computing, the development of its ecosystem is still in its infancy. The requirements for tools enabling platform-independent computing [4, 5], contextualization [6], elastic computing [7], or offering other functionality providing easy access to cloud facilities to the end user are still being developed. As cloud computing became more popular, the applications used in the cloud exceeded the basic need to deploy a few images and run a simple application. Sophisticated launches often contain additional service nodes including storage, databases, identity servers, brokers, and other support services. Further, while infrastructures serving local communities are often built or modified in a matter of months or years by a consistent team, a complex cloud launch may be repeated many times a day by several different people.

These considerations create a need for a tool enabling a controlled and repeatable launch and management of a set of virtual machines (VMs) working together to achieve a single goal. This task is often challenging because little can be assumed about the network locations of these VMs (their IP addresses are dynamically provisioned), they are frequently interdependent on each other, and their deployment can be spread across many different clouds providers, potentially supporting different interfaces. Specifically, the following questions arise: How can we orchestrate large-scale, multicloud, and multi-VM application launches? How can we organize, manage, and coordinate the bootstrap process of these complex cloud applications in a repeatable way? Once these applications are running, how can we ensure that they continue to work, and can we recover from failures without having to waste valuable time and potential data by completely restarting them?

In this paper, we introduce cloudinit.d, a tool for launching, configuring, monitoring, and repairing a set of interdependent VMs in an IaaS cloud or over a set of IaaS clouds. A single launch can consist of many VMs and can span multiple IaaS providers, including offerings from commercial and academic space such as the many clouds offered by the FutureGrid project [14]. Like the UNIX init.d program, cloudinit.d can launch specified groups of VMs at different run levels representing dependencies of the launched VMs. The launch is accomplished based on a well-defined launch plan. The launch plan is defined in a set of easy-to-understand text-based *ini* formatted files that can be kept under version control. Because the launch process is repeatable, cloud applications can be developed in a structured

TG'11, July 18 - July 21 2011, Salt Lake City, UT, USA

Copyright 2011 ACM 978-1-4503-0888-5/11/07…$10.00.

and iterative way. Once launched, cloudinit.d monitors the health of each running service and the VMs in which they run. If a problem is detected in a service, cloudinit.d will restart only that service (and potentially its hosting VM) and the dependencies of that service which also failed. Cloudinit.d was developed in the context of the Ocean Observatory Initiative project [8] to coordinate and repair launches of VMs and services hosted by both VMs and bare metal machines.

This paper is structured as follows. In Section 2 we define the requirements and design principles guiding the development of cloudinit.d. In Section 3 we describe its architecture and implementation. In Section 4 we present an application example and review the design features based on this example. In Section 5 we discuss related efforts. We summarize our conclusions in Section 6.

2. REQUIREMENTS AND GOALS

Based on our experience involving launches of scientific applications in the clouds [14] as well as the experiences of complex launches within the Ocean Observatory Initiative (OOI) project [8], we have developed a set of requirements and goals for an infrastructure cloud-friendly launch tool. One important insight was that the tool should be able to not only launch complex sets of interdependent VMs but also diagnose failure as appropriate and keep the VMs running if possible. These guidelines are summarized below:

- *Repeatable, one-click deployment of sets of VMs.* Outsourcing to infrastructure clouds often includes support services—such as storage, databases or identity servers—forming a complex network. Further, those networks of services can be deployed and redeployed in the cloud frequently and by different actors. In order to achieve consistent behavior of such systems, it is important to execute VM launches based on a launch plan that can be *created once and executed many times, by different actors, in exactly the same way*. The execution of this requirement is limited by the degree of repeatability provided by IaaS providers: in many cases it is impossible to repeat individual deployment actions; for example, a deployment of an instance on the Amazon Web Services (AWS) provider [9] may result in many different instantiations [10].

- *Coordination of interdependent launches.* The services within one launch can be interdependent in that information required for the deployment of one can be provided as a result of the deployment of another. For example, a service may need to know the hostname of a database server to complete its launch sequence—in this case the database server needs to be deployed first, and the information about the hostname needs to be conveyed. On the other hand, services can also be independent and in this case can be deployed concurrently to save time. Dividing the service launch into *run levels* composed of independent services accommodates both scenarios; each run level can define and resolve attributes to values that can be used by services launched in downstream run levels.

- *Federated cloud deployment scenario.* Many deployments move between different cloud providers, use several different providers for one launch [24], or use both cloud and noncloud resources [25]. Thus, any launch management tool for infrastructure clouds should be platform-agnostic so that it *can be deployed on any IaaS cloud as well as integrate noncloud resources*. In order to achieve portability and flexibility it is important not only to work with multiple providers but also to launch different VMs on different clouds with a single launch plan. This can be achieved by using adapters, such as libcloud [5] or deltacloud [4] that provide a bridge to many IaaS cloud providers and services, or by leveraging the increasing availability of standards such as the specification recently released by Open Grid Forum's Open Cloud Computing Interface (OCCI) [11]. Launching services on noncloud resources can be accomplished by accessing the resource and configuring the service; however, launching them on a VM has the advantage that the environment is known and controlled by the user and thus carries the risk of fewer potential failures and inconsistencies.

- *Testing a launch.* To deal with complex launches in a structured way and be able to reason about a complex system, a user must be able to make and verify assertions about vital properties of the system. Those assertions can be both generic (e.g., "Is the VM responding to pings"?), and user-defined (e.g., testing an application-specific property of a system). For this reason, it is important that the user can define arbitrary soundness tests for the system. Management tools for infrastructure clouds should thus provide mechanisms that allow users to select or configure such tests, associate them with services, and execute them to validate the correctness of a launch both at deployment time and running time. In order to ensure meeting a wide range of useful tests, they should be executed inside the VMs (e.g., based on ssh into the VM), rather than rely on external information only.

- *Ongoing monitoring of a launch.* In order to closely monitor the health of the system, it is essential that the vital assertions about the system can be reevaluated at any time. Therefore, if such assertions are embedded in the launch tests, those tests need to be able to be rerun not just at launch but at any time by an action triggered automatically or manually by the user (i.e., launch operator). It should be possible to store the results of monitoring tests in a database for launch analysis and recreation.

- *Policy-driven repair and maintenance of a launch.* If any of the assertions about the system (as embodied by the tests) fail, it should be possible to repair the launch components by applying a repair action defined by a policy. For example, a failure can lead to a number of repeats of a launch action or abandonment of a launch component or even the whole launch if a component is deemed to be irreparable.

- *Lightweight and easy to use.* In a cloud scenario, where multiple users are building, sharing, and improving complex launch plans on a daily basis, a launch tool that is complex, has a learning curve, and requires configuration is only exchanging one type of complexity for another. In such an environment a launch plan has to operate based on an easily copied launch plan and a "one-click" action. Therefore, while it is tempting to provide a more general system that can be customized to this particular task, the launch tool must be purpose-specific and spare in terms of functionality. Further, for ease of use, a minimum of software should be required to be preinstalled and running on a host system. Often configuration management tools require that an agent be custom installed and run. This requirement makes it impossible to use the multitude of freely and readily available Linux VM image distributions without having to rebundle the VM images. Rebundling VM images can be a difficult task for all but the most advanced users.

3. ARCHITECTURE AND IMPLEMENTTATION

We discuss in this section the architecture of cloudinit.d and its implementation.

3.1 Launch Plans

Cloudinit.d arranges an application into three basic constructs: atomic service, run level, and launch plan.

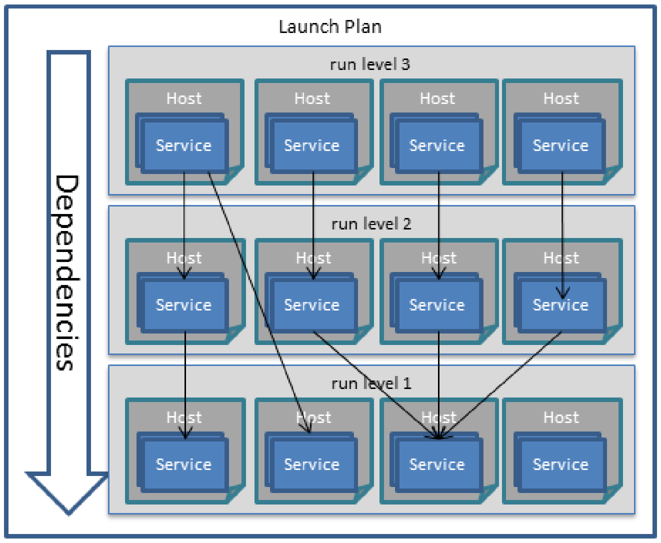

Figure 1: Launch plan example shows relationships between components: the first run-level contains all the services without dependencies as well as services that run-level 2 depends on; run-level 3 depends on run-level 2.

- A *service* is an entity confined to a single host machine and responsible for a well-defined task. A service can be hosted by a VM that is automatically launched by cloudinit.d or by an existing machine that is accessible via ssh. Many services can be configured to run in a single host, but often a service is associated with a VM dedicated to its needs. One may think of a service as a newly launched and configured VM with a single, dedicated purpose. Examples of s*ervices ar*e an HTTP server, a node in a Cassandra [12] pool, or a node in a RabbitMQ [13] message queue.
- A *run level* is a collection of services with no dependencies on each other. All services in a run level can be launched at the same time. A run-level launch is considered complete when all of the services in it have successfully started. Services in a run level can be run on one single cloud or across many different clouds, since cloudinit.d makes no assumptions about locality. Any service in a run level can depend on any service from a previous run level. For example, run-level 1 forms a database. A web application in run-level 2 can depend on that database, meaning, it can acquire all of the information needed to connect to it, like security tokens and contact port, dynamically at boot time.
- A *launch* is an ordered set of run levels. To make a launch plan, first one defines all the services. Then those services are arranged into run levels in a specific order: the services with no dependencies are put in run-level 1 since they all can be started simultaneously without any additional information, the next run level is composed of services with dependencies on level 1 only, and so forth. The completed launch forms a complete cloud (or intercloud) application.

Figure 1 shows an example of services collected into run levels; the arrows show the dependencies of one service on another. When a service needs information from another, it depends on it and thus must be in a higher run level. It can request dynamic information about another service at boot time or repair time. This powerful feature allows the location of any given service to be entirely dynamic. We note that a service may depend on already running systems not controlled by the launch operator. For example, if a service uses Amazon's S3 [28] as a data store, all the information needed to connect with the operators S3 account can be passed into a service as a first-class part of the bootstrap process.

3.2 Configuring Services

Two factors determine how a service behaves and how cloudinit.d interacts with it: the software preinstalled on the service's host and a collection of scripts defining its startup and termination properties as well as its operating assertions. The service's host can be a VM or a bare-metal running machine. The various scripts and their functions are described below:

- The *startup script (bootpgm)* is a program that is run once at launch time to configure the service. The purpose of this program is to set up the host server with all needed software and start that software using any tools convenient to the user. The program is copied by cloudinit.d to a distinct location inside the service's host as soon as ssh access to that host has been verified. This strategy minimizes the need for preinstalled software to simply sshd and thus maximizes the selection of possible resources to use as host systems for a service. The startup agent will often download and install software and then configure that software for use. Tools such as apt-get[15], yum[16], chef-solo[17], or puppet [29] can be used by this script to perform these functions: the choice of this implementation technology is left to the user, the only constraint being that it must be executable by the service's host.

- The test script (*readypgm)* is a program whose purpose is to check the status and health of the service. It can be, and typically is, run many times during the lifetime of an application. When the user of cloudinit.d requests the current status of a previously launched cloud application the dependency graph of run levels is again walked. This time the test script is copied into the service's host and run via ssh. As an example, if the service's goal is to serve HTTP, the readypgm might connect to localhost:80, download a known web page, and check its content. If all is well, the readypgm returns 0, and the service is reported as working. If not, the service is marked as *down,* and the cloud application is in need of repair. The output from the readypgm is logged in a file local with respect to cloudinit.d so that the user can inspect the results of a failed service for more details.

- *The termination script (terminatepgm)* is a program run when a service is shut down. It is there to nicely clean up resources associated with the service. For example, a service that has data kept in memory buffers can use this hook to make sure all buffers are flushed to a permanent store.

Both the software installed on the various services host systems and the scripts described above provide a baseline for the capabilities of the service and are defined by the author of the launch plan (and thus the author of the service).

3.3 Boot Process

Here we describe how a single service is brought into existence by cloudinit.d. First, a launch plan is created that describes a cloud application with functionality arranged into the components described above; this launch plan is given to the cloudinit.d command line program. The first action cloudinit.d takes is to validate that the launch plan has no errors. Launch plans tend to require the acquisition of VM resources, and this acquisition is often costly in terms of time and money; thus we want to avoid the case where the first nine run levels pay the price, only to discover a bug in the launch plan in level 10.

Once the plan is validated, all IaaS requests for new VMs are made. We prestage this request as an optimization: starting a new VM can take a significant amount of time, and many tasks can be done in parallel.

Then cloudinit.d starts monitoring the services at run-level 1, first waiting for the IaaS system on which any VM on that level was launched to report an associated hostname. Once a hostname is present, cloudinit.d repeatedly, but not aggressively, attempts to contact the service's host. Contact is made by attempting to ssh into the system. When a successful ssh connection is made, the bootpgm is copied to a distinct location via scp. In higher run levels where there is a possible service dependency, an additional document that contains dependency information about all previously created services (currently expressed in json[18]) is copied to the service's host. The bootpgm script is then run inside that host. It can use the information in the dependency document for discovering values about its dependencies from services at lower run levels. As described above, this script is responsible for setting up the service for use in the cloud application. If it is successful, it must have an exit code of 0.

Along with the exit code, the bootpgm can return a dependency document containing a set of key/value pairs that describe attributes of the newly configured service for consumption by higher-level services. For example, an HTTPS service may want to return the public key that it generated at boot time so that higher-level services can safely access it. This document is used by cloudiit.d to furnish services at higher run levels with dynamic information about this newly created service.

Once all the services at a run level have completed successfully, the process is then performed on the next run level until all run levels have completed.

3.4 Security Considerations

The operator of cloudinit.d must have ssh private keys that allow access to all service hosts in the cloud application. When dealing with IaaS clouds (typical with cloudinit.d) this is not a problem. Ssh key management is a first-class feature with all IaaS clouds, and there is a well-understood process for managing keys. When the service's host is a static machine, the operator must make sure that the machine can be accessed via ssh in order to use it with cloudinit.d. Various configuration options in the launch plans handle the nuances of this situation.

The operator will also need to have access tokens for any and all of the IaaS clouds from which resources will be requested. The details of those keys are associated with the launch plan and are described in later sections.

We note that the operator of a cloudinit.d (i.e., the account that launched the boot plan) should be considered the root-level user of the application. This operator will have root access to all IaaS-launched VMs and will often need root-level access to the static system. Further, the cloudinit.d operator has the power to terminate the application. As part of the launch or once the cloud application has been launched, lower-level access can be granted in an application-specific way, but the owner and operator of the launch plan will be the root-level user.

3.5 Repair

In any distributed system, failures are inevitable. In complex cloud applications, diagnosing and repairing single failures can be an arduous task. System failures are detected by cloudinit.d by running the test script, readypgm, inside the service's host machine. The user can manually decide when to check the status of an application with the cloudinit.d command-line program or the cloudinit.d python API. If the readypgm returns a failure on a given service, that service can be automatically repaired.

The repair process works by first running the terminatepgm (if it exists) and then shutting down the VM hosting the service if necessary. The boot process described above is then run on that service alone. In this reboot many of the dynamically determined attributes of this service (e.g., hostname) will likely change. Hence, once the reboot is complete, cloudinit.d will test all the services at a higher level. If any of those services fail, they too will be repaired in this way, propagating the repair up the levels.

4. APPLICATION EXAMPLE

Figure 2 shows an example cloud application run on four of the FutureGrid clouds using cloudinit.d. Here we have a highly available web application that uses a Cassandra distributed database for highly available storage, Apache HTTP servers for its web application, and a load balancer to distribute the work. To avoid a single point of failure and to ensure locality, each node of both the Cassandra database and the web farm is distributed geographically across all four clouds. The load balancer is run on a single static host with a known location. As part of our launch plan in this example, we will tell cloudinit.d how to contact each of the four clouds and that we want two VMs in each cloud, one to handle the workload of Cassandra and a second to handle the web requests. In total the application will have eight VMs (two in each cloud) and one static machine.

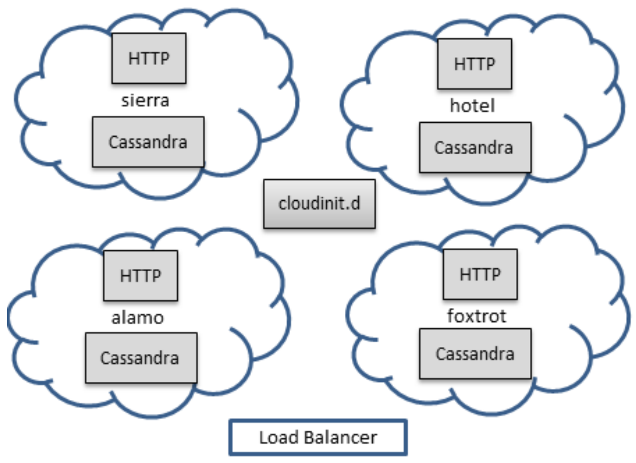

Figure 2: Example application using cloudinit.d deployed on FutureGrid Nimbus clouds.

4.1 Booting and Configuring a Single Instance

The launch plan for this application has three run levels. The first has the four Cassandra nodes as services configured to run in each of the four FutureGrid clouds. The second is a set of replicated HTTP servers, configured similarly. The final run level is the load balancer, which is run on a bare-metal host that is assumed to be running prior to the boot of this application. The plan is configured in such a way as to route the important connection information from the Cassandra cluster to each HTTP server. Similarly, the list of HTTP servers is sent to the load balancer once run-level 2 completes.

As input, cloudinit.d takes a set of configuration files. To provide a practical understanding of how to create a working launch plan, we describe its details here. The launch plan has two main file types: a top-level configuration file and a run-level configuration file. We chose the ini style file format here because the description needs are simple and limited to key/value pairs. A feature-rich, yet harder to read and understand description language like XML or JSON would make the launch plan author's job unnecessarily complicated. The top-level configuration file simply enumerates the run levels and associates each run level with an additional configuration file. Our example top-level configuration file follows.

```
[runlevels]
level1: cassandra.conf
level2: http.conf
level3: loadbalance.conf
```

This tells cloudinit.d that there are three run levels and where the description of those run levels can be found. Inside each of those files is a description of each service.

Here we will just introduce the *service* section of the boot level configuration file that describes the Cassandra service that will be run on the FutureGrid sierra cloud.

```
[svc-sierraCassandra]
iaas_key: XXXXXX
iaas_secret: XXXX
iaas_hostname: sierra.futuregrid.org
iaas_port: 8443
iaas: Nimbus
image: ubunut10.10
ssh_username: ubuntu
localsshkeypath: ~/.ssh/fg.pem
readypgm: cass-test.py
bootpgm: cass-boot.sh
```

The first five entries describe the cloud on which the service will be created. User security tokens and the contact point of the cloud are placed here. Cloudinit.d is also told what type of cloud has been described. In this case it is *Nimbus,* but it could be other common cloud types such as EC2 [19] or Eucalyptus [20].

The line *image: ubuntu10.10* is a directive saying to request that the IaaS cloud launch the image with that name. In our example the image is a base Ubuntu 10.10 image. Both *ssh_username* and *localsshkeypath* give cloudinit.d the needed information for establishing a communication link with the VM instance launched from the *ubuntu10.10* image.

Readypgm and bootpgm point to scripts that perform the tasks associated with the respective directives (described in detail above). In our case cass-boot.sh will be copied to the VM instance with scp. The ssh session will be formed using the key at *localsshkeypath* and the username *ubuntu*. Then cass-boot.sh will be run via ssh as the *ubuntu* user. It will download all the software needed for Cassandra to run and configure this node to be a member of the four-node cluster. When it is complete, it will return to cloudinit.d the contact information of the newly created Cassandra node. A similar entry is made in cassandra.conf for the other clouds.

Once all four Cassandra services have been successfully booted, cloudinit.d will open the configuration file http.conf. The contents of this file will look similar to that of cassandra.conf, the main difference being the bootpgm used to configure the system. Again four new VMs will be started on each of the four clouds. Cloudinit.d will again ssh into these machines and stage in the configuration scripts. Instead of setting up Cassandra, however, the bootgm will download, install, and configure a web server. Furthermore, it will connect to the Cassandra data store created in the previous boot level.

The final step in our example application is setting up the load balancer. In this case the host machine will not be a VM. Instead it will be a static machine at a given hostname. The process for configuring it works in almost an identical way, only without the initial request to an IaaS framework to start a VM. That first step is skipped, and the process continues by accessing the given hostname with scp and ssh. Because there are no further run levels, once this run level successfully completes, details about each started service are reported to a log file, and a summary is reported to the console for immediate observation by the operator.

5. RELATED WORK

CloudFormation [21] is a product created by Amazon Web Services. Much like cloudinit.d, it is used to create and operate distributed applications in the cloud in a predictable and repeatable way. Unlike cloudinit.d, however, CloudFormation cannot be used across many clouds; it is a tool entirely dedicated for use with AWS only and cannot be used with the many resources available in science clouds or other commercial clouds. A further and important difference between the two systems is that cloudinit.d is designed to boot and contextualize an ordered hierarchy of VMs. CloudFormation is designed specifically to make all the AWS services (like SQS [22] and Elastic Beanstalk [23]), work in concert with each other; it does not do VM contextualization, and it does not have an explicit notion of boot order.

Configuration management systems such as puppet and chef are designed to manage the configuration of systems declaratively. Much like cloudinit.d, they can set up and maintain services on a set of systems typically in a cluster. However, unlike cloudinit.d, the systems on which they are run require that existing software agents be installed and running on all systems in their cluster. In contrast, the only initial dependency that cloudinit.d has is the ubiquitous sshd. This allows cloudinit.d to work with base images readily available on virtually every cloud and does not require the user to bundle a special image. For example, Ubuntu images are regularly released on EC2 for public use (as are many other Linux distributions). Furthermore, far from being a competitor of configuration management systems, cloudinit.d is designed to work with them. In best practices cloudinit should be used to provision a host with all of the software, security information, and configuration files needed to properly run a configuration management system such as chef solo (as demonstrated in our example).

The Context Broker [30] from the Nimbus project is a service that allows clients to coordinate large virtual cluster launches and created the concept of the *one-click virtual cluster*. While cloudinit.d handles one-sided configuration dependencies between services, the Context Broker is capable of handling more complex interdependencies with additional complexity cost. Furthermore, while the Context Broker focuses exclusively on configuration management of VMs provisioned by another agent, cloudinit.d bundles configuration management with provisioning resources across multiple clouds. Wrangler [31] is a recent research project providing functionality similar to the Context Broker, with similar trade-offs.

Claudia is an open source service manager originally affiliated with the One Nebula project but now absorbed by Stratus Cloud. Claudia is more tightly coupled to IaaS services and the management of VMs than is cloudinit.d is. Further, its architecture has several running components and thus presents a much more complex and heavyweight solution than does cloudinit.d.

6. SUMMARY

This paper introduces cloudinit.d, a tool for launching, configuring, monitoring, and repairing a set of interdependent VMs in an infrastructure-as-a-service (IaaS) cloud or over a set of IaaS clouds. In addition, similar to its namesake the UNIX init.d program, cloudinit.d can launch specified groups of VMs at different run levels representing dependencies of the launched VMs. This feature facilitates dealing with interdependencies with VM while optimizing the launch by allowing independent VMs to launch at the same time.

Cloudinit.d provides a new addition to the cloud computing ecosystem, making it easier for scientists to repeatedly launch, manage, and reason about sets of VMs deployed in he cloud. The capability to deploy launches repeatably is particularly important in the construction of stable system; and the ability to evaluate, at any time, application-specific assertions significantly simplifies VM launches in cloud environment.

ACKNOWLEDGMENTS

This material is based on work supported in part by the National Science Foundation under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." Partners in the FutureGrid project include U. Chicago, U. Florida, San Diego Supercomputer Center - UC San Diego, U. Southern California, U. Texas at Austin, U. Tennessee at Knoxville, U. of Virginia, Purdue I., and T-U. Dresden. This work also was supported in part by the Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. The OOI Cyberinfrastructure program is funded through the JOI Subaward, JSA 7-11, which is in turn funded by the NSF contract OCE-0418967 with the Consortium for Ocean Leadership, Inc.

REFERENCES

1. Armbrust, M., et al., Above the Clouds: A Berkeley View of Cloud Computing. 2009, University of California at Berkeley.

2. Keahey, K., T. Freeman, J. Lauret, and D. Olson. Virtual Workspaces for Scientific Applications. in SciDAC Conference. 2007. Boston, MA.

3. Beckman, P., S. Nadella, N. Trebon, and I. Beschastnikh, SPRUCE: A System for Supporting Urgent High-Performance Computing. IFIP International Federation for Information Process, Grid-Based Problems Solving Environments, 2007(239): pp. 295-311.

4. deltacloud: http://incubator.apache.org/deltacloud/.

5. libcloud: a unified interface to the cloud: http://incubator.apache.org/libcloud/.

6. Keahey, K., and T. Freeman. Contextualization: Providing One-click Virtual Clusters. in eScience. 2008. Indianapolis, IN.

7. Marshall, P., K. Keahey, and T. Freeman, Elastic Site: Using Clouds to Elastically Extend Site Resources. CCGrid 2010, 2010.

8. Meisinger, M., C. Farcas, E. Farcas, C. Alexander, M. Arrott, J. de La Beaujardiere, P. Hubbard, R. Mendelssohn, and R. Signell. Serving Ocean Model Data on the Cloud. Oceans 09, 2009.

9. Amazon Web Services (AWS): http://aws.amazon.com/.

10. Jackson, K., L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud Amazon Web Services Cloud. CloudCom. 2010.

11. Open Cloud Computing Interface (OCCI): http://occi-wg.org/.

12. Cassandra: http://cassandra.apache.org/.

13. RabbitMQ: http://www.rabbitmq.com/.

14. FutureGrid: https://portal.futuregrid.org/

15. Debian HowTo: http://www.debian.org/doc/manuals/apt-howto/

16. Fedora Project: http://fedoraproject.org/wiki/Tools/yum

17. OpsCode: http://wiki.opscode.com/display/chef/Chef+Solo

18. JSON: http://www.json.org/

19. EC2: http://aws.amazon.com/ec2/

20. Eucalyptus: http://www.eucalyptus.com/

21. CloudFormation: http://aws.amazon.com/cloudformation/

22. SQS: http://aws.amazon.com/sqs/

23. Elastic Beanstalk: http://aws.amazon.com/elasticbeanstalk/

24. Riteau, Pierre, Mauricio Tsugawa, Andrea Matsunaga, José Fortes, Tim Freeman, David LaBissoniere, and Kate Keahey. Sky Computing on FutureGrid and Grid'5000. TeraGrid 2010, Pittsburgh, PA. August 2010.

25. http://www.isgtw.org/feature/case-missing-proton-spin

26. Wilde, Michael, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A Language for Distributed Parallel Scripting. Parallel Computing 2011.

27. Sonntag, Mirko, Dimka Karastoyanova, and Ewa Deelman. Bridging the Gap between Business and Scientific Workflows. e-Science 2010, Brisbane, Australia, 2010.

28. Amazon Simple Storage Service (Amazon S3): http://aws.amazon.com/s3/.

29. Puppet: http://projects.puppetlabs.com/projects/puppet

30. Keahey, K., T. Freeman. Contextualization: Providing One-Click Virtual Clusters, eScience 2008, Indianapolis, IN. December 2008.

31. Juve, Gideon, and Ewa Deelman. Wrangler: Virtual Cluster Provisioning for the Cloud. Short paper in Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC'11), 2011.