

Automating Application Deployment in Infrastructure Clouds

Gideon Juve and Ewa Deelman
USC Information Sciences Institute
Marina del Rey, California, USA
{gideon,deelman}@isi.edu

Abstract—Cloud computing systems are becoming an important platform for distributed applications in science and engineering. Infrastructure as a Service (IaaS) clouds provide the capability to provision virtual machines (VMs) on demand with a specific configuration of hardware resources, but they do not provide functionality for managing resources once they are provisioned. In order for such clouds to be used effectively, tools need to be developed that can help users to deploy their applications in the cloud. In this paper we describe a system we have developed to provision, configure, and manage virtual machine deployments in the cloud. We also describe our experiences using the system to provision resources for scientific workflow applications, and identify areas for further research.

Keywords—cloud computing; provisioning; application deployment

I. INTRODUCTION

Infrastructure as a Service (IaaS) clouds are becoming an important platform for distributed applications. These clouds allow users to provision computational, storage and networking resources from commercial and academic resource providers. Unlike other distributed resource sharing solutions, such as grids, users of infrastructure clouds are given full control of the entire software environment in which their applications run. The benefits of this approach include support for legacy applications and the ability to customize the environment to suit the application. The drawbacks include increased complexity and additional effort required to setup and deploy the application.

Current infrastructure clouds provide interfaces for allocating individual virtual machines (VMs) with a desired configuration of CPU, memory, disk space, etc. However, these interfaces typically do not provide any features to help users deploy and configure their application once resources have been provisioned. In order to make use of infrastructure clouds, developers need software tools that can be used to configure dynamic execution environments in the cloud.

The execution environments required by distributed scientific applications, such as workflows and parallel programs, typically require a distributed storage system for sharing data between application tasks running on different nodes, and a resource manager for scheduling tasks onto nodes [12]. Fortunately, many such services have been developed for use in traditional HPC environments, such as clusters and grids. The challenge is how to deploy these services in the cloud given the dynamic nature of cloud environments. Unlike

clouds, clusters and grids are static environments. A system administrator can setup the required services on a cluster and, with some maintenance, the cluster will be ready to run applications at any time. Clouds, on the other hand, are highly dynamic. Virtual machines provisioned from the cloud may be used to run applications for only a few hours at a time. In order to make efficient use of such an environment, tools are needed to automatically install, configure, and run distributed services in a repeatable way.

Deploying such applications is not a trivial task. It is usually not sufficient to simply develop a virtual machine (VM) image that runs the appropriate services when the virtual machine starts up, and then just deploy the image on several VMs in the cloud. Often the configuration of distributed services requires information about the nodes in the deployment that is not available until after nodes are provisioned (such as IP addresses, host names, etc.) as well as parameters specified by the user. In addition, nodes often form a complex hierarchy of interdependent services that must be configured in the correct order. Although users can manually configure such complex deployments, doing so is time consuming and error prone, especially for deployments with a large number of nodes. Instead, we advocate an approach where the user is able to specify the layout of their application declaratively, and use a service to automatically provision, configure, and monitor the application deployment. The service should allow for the dynamic configuration of the deployment, so that a variety of services can be deployed based on the needs of the user. It should also be resilient to failures that occur during the provisioning process and allow for the dynamic addition and removal of nodes.

In this paper we describe and evaluate a system called *Wrangler* [10] that implements this functionality. *Wrangler* allows users to send a simple XML description of the desired deployment to a web service that manages the provisioning of virtual machines and the installation and configuration of software and services. It is capable of interfacing with many different resource providers in order to deploy applications across clouds, supports plugins that enable users to define custom behaviors for their application, and allows dependencies to be specified between nodes. Complex deployments can be created by composing several plugins that set up services, install and configure application software, download data, and monitor services, on several interdependent nodes.

The remainder of this paper is organized as follows. In the next section we describe the requirements for a cloud deployment service. In Section III we explain the design and operation of Wrangler. In Section IV we present an evaluation of the time required to deploy basic applications on several different cloud systems. Section V presents two real applications that were deployed in the cloud using Wrangler. Sections VI and VII describe related work and conclude the paper.

II. SYSTEM REQUIREMENTS

Based on our experience running science applications in the cloud [11,12,33], and our experience using the Context Broker from the Nimbus cloud management system [15] we have developed the following requirements for a deployment service:

- *Automatic deployment of distributed applications.* Distributed applications used in science and engineering research often require resources for short periods in order to complete a complex simulation, to analyze a large dataset, or complete an experiment. This makes them ideal candidates for infrastructure clouds, which support on-demand provisioning of resources. Unfortunately, distributed applications often require complex environments in which to run. Setting up these environments involves many steps that must be repeated each time the application is deployed. In order to minimize errors and save time, it is important that these steps are automated. A deployment service should enable a user to describe the nodes and services they require, and then automatically provision, and configure the application on-demand. This process should be simple and repeatable.
- *Complex dependencies.* Distributed systems often consist of many services deployed across a collection of hosts. These services include batch schedulers, file systems, databases, web servers, caches, and others. Often, the services in a distributed application depend on one another for configuration values, such as IP addresses, host names, and port numbers. In order to deploy such an application, the nodes and services must be configured in the correct order according to their dependencies, which can be expressed as a directed acyclic graph. Some previous systems for constructing virtual clusters have assumed a fixed architecture consisting of a head node and a collection of worker nodes [17,20,23,31]. This severely limits the type of applications that can be deployed. A virtual cluster provisioning system should support complex dependencies, and enable nodes to advertise values that can be queried to configure dependent nodes.
- *Dynamic provisioning.* The resource requirements of distributed applications often change over time. For example, a science application may require many worker nodes during the initial stages of a

computation, but only a few nodes during the later stages. Similarly, an e-commerce application may require more web servers during daylight hours, but fewer web servers at night. A deployment service should support dynamic provisioning by enabling the user to add and remove nodes from a deployment at runtime. This should be possible as long as the deployment's dependencies remain valid when the node is added or removed. This capability could be used along with elastic provisioning algorithms (e.g. [19]) to easily adapt deployments to the needs of an application at runtime.

- *Multiple cloud providers.* In the event that a single cloud provider is not able to supply sufficient resources for an application, or reliability concerns demand that an application is deployed across independent data centers, it may become necessary to provision resources from several cloud providers at the same time. This capability is known as federated cloud computing or sky computing [16]. A deployment service should support multiple resource providers with different provisioning interfaces, and should allow a single application to be deployed across multiple clouds.
- *Monitoring.* Long-running services may encounter problems that require user intervention. In order to detect these issues, it is important to continuously monitor the state of a deployment in order to check for problems. A deployment service should make it easy for users to specify tests that can be used to verify that a node is functioning properly. It should also automatically run these tests and notify the user when errors occur.

In addition to these functional requirements, the system should exhibit other characteristics important to distributed systems, such as scalability, reliability, and usability.

III. ARCHITECTURE AND IMPLEMENTATION

We have developed a system called Wrangler to support the requirements outlined above. The components of the system are shown in Figure 1. They include: *clients*, a *coordinator*, and *agents*.

- *Clients* run on each user's machine and send requests to the coordinator to launch, query, and terminate, deployments. Clients have the option of using a command-line tool, a Python API, or XML-RPC to interact with the coordinator.
- The *coordinator* is a web service that manages application deployments. It accepts requests from clients, provisions nodes from cloud providers, collects information about the state of a deployment, and acts as an information broker to aid application configuration. The coordinator stores information about its deployments in an SQLite database.

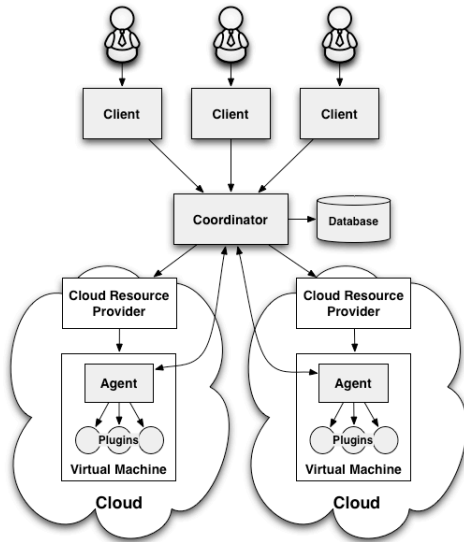


Figure 1: System architecture

- *Agents* run on each of the provisioned nodes to manage their configuration and monitor their health. The agent is responsible for collecting information about the node (such as its IP addresses and hostnames), reporting the state of the node to the coordinator, configuring the node with the software and services specified by the user, and monitoring the node for failures.
- *Plugins* are user-defined scripts that implement the behavior of a node. They are invoked by the agent to configure and monitor a node. Each node in a deployment can be configured with multiple plugins.

A. Specifying Deployments

Users specify their deployment using a simple XML format. Each XML request document describes a deployment consisting of several *nodes*, which correspond to virtual machines. Each node has a *provider* that specifies the cloud resource provider to use for the node, and defines the characteristics of the virtual machine to be provisioned—including the VM image to use and the hardware resource type—as well as authentication credentials required by the provider. Each node has one or more *plugins*, which define the behaviors, services and functionality that should be implemented by the node. Plugins can have multiple *parameters*, which enable the user to configure the plugin, and are passed to the script when it is executed on the node. Nodes may be members of a named *group*, and each node may depend on zero or more other nodes or groups.

An example deployment is shown in Figure 2. The example describes a cluster of 4 nodes: 1 NFS server node, and 3 NFS client nodes. The clients, which are identical, are specified as a single node with a “count” of three. All nodes are to be provisioned from Amazon EC2, and different images and instance types are specified for the server and the clients. The server is configured with an “nfs_server.sh” plugin, which starts the required NFS services and exports the /mnt

```

<deployment>
  <node name="server">
    <provider name="amazon">
      <image>ami-912837</image>
      <instance-type>c1.xlarge</instance-type>
      ...
    </provider>
    <plugin script="nfs_server.sh">
      <param name="EXPORT">/mnt</param>
    </plugin>
  </node>
  <node name="client" count="3" group="clients">
    <provider name="amazon">
      <image>ami-901873</image>
      <instance-type>m1.small</instance-type>
      ...
    </provider>
    <plugin script="nfs_client.sh">
      <param name="SERVER">
        <ref node="server" attribute="local-ipv4">
      </param>
      <param name="PATH">/mnt</param>
      <param name="MOUNT">/nfs/data</param>
    </plugin>
    <depends node="server" />
  </node>
</deployment>

```

Figure 2: Example request for 4 node virtual cluster with a shared NFS file system

directory. The clients are configured with an “nfs_client.sh” plugin, which starts NFS services and mounts the server’s /mnt directory as /nfs/data. The “SERVER” parameter of the “nfs_client.sh” plugin contains a <ref> tag. This parameter is replaced with the IP address of the server node at runtime and used by the clients to mount the NFS file system. The clients are part of a “clients” group, and depend on the server node, which ensures that the NFS file system exported by the server will be available for the clients to mount when they are configured.

B. Deployment Process

Here we describe the process that Wrangler goes through to deploy an application, from the initial request, to termination.

Request. The client sends a request to the coordinator that includes the XML descriptions of all the nodes to be launched, as well as any plugins used. The request can create a new deployment, or add nodes to an existing deployment.

Provisioning. Upon receiving a request from a client, the coordinator first validates the request to ensure that there are no errors. It checks that the request is valid, that all dependencies can be resolved, and that no dependency cycles exist. Then it contacts the resource providers specified in the request and provisions the appropriate type and quantity of virtual machines. In the event that network timeouts and other transient errors occur during provisioning, the coordinator automatically retries the request.

The coordinator is designed to support many different cloud providers. It currently supports Amazon EC2 [1], Eucalyptus [24], and OpenNebula [25]. Adding additional providers is designed to be relatively simple. The only functionalities that a cloud interface must provide are the

ability to launch and terminate VMs, and the ability to pass custom contextualization data to a VM.

The system does not assume anything about the network connectivity between nodes so that an application can be deployed across many clouds. The only requirement is that the coordinator can communicate with agents and vice versa.

Startup and Registration. When the VM boots up, it starts the agent process. This requires the agent software to be pre-installed in the VM image. The advantage of this approach is that it offloads the majority of the configuration and monitoring tasks from the coordinator to the agent, which enables the coordinator to manage a larger set of nodes. The disadvantage is that it requires users to re-bundle images to include the agent software, which is not a simple task for many users and makes it more difficult to use off-the-shelf images. In the future we plan to investigate ways to install the agent at runtime to avoid this issue.

When the agent starts, it uses a provider-specific *adapter* to retrieve contextualization data passed by the coordinator, and to collect attributes about the node and its environment. The attributes collected include: the public and private hostnames and IP addresses of the node, as well as any other relevant information available from the metadata service, such as the availability zone. The contextualization data includes: the host and port where the coordinator can be contacted, the ID assigned to the node by the coordinator, and the node's security credentials. Once the agent has retrieved this information, it is sent to the coordinator as part of a registration message, and the node's status is set to 'registered'. At that point, the node is ready to be configured.

Configuration. When the coordinator receives a registration message from a node it checks to see if the node has any dependencies. If all the node's dependencies have already been configured, the coordinator sends a request to the agent to configure the node. If they have not, then the coordinator waits until all dependencies have been configured before proceeding.

After the agent receives a command from the coordinator to configure the node, it contacts the coordinator to retrieve the list of plugins for the node. For each plugin, the agent downloads and invokes the associated plugin script with the user-specified parameters, resolving any <ref> parameters that may be present. If the plugin fails with a non-zero exit code, then the agent aborts the configuration process and reports the failure to the coordinator, at which point the user must intervene to correct the problem. If all plugins were successfully started, then the agent reports the node's status as 'configured' to the coordinator.

Upon receiving a message that the node has been configured, the coordinator checks to see if there are any nodes that depend on the newly configured node. If there are, then the coordinator attempts to configure them as well. It makes sure that they have registered, and that all dependencies have been configured.

The configuration process is complete when all agents report to the coordinator that they are configured.

Monitoring. After a node has been configured, the agent periodically monitors the node by invoking all the node's plugins with the *status* command. After checking all the plugins, a message is sent to the coordinator with updated attributes for the node. If any of the plugins report errors, then the error messages are sent to the coordinator and the node's status is set to 'failed'.

Termination. When the user is ready to terminate one or more nodes, they send a request to the coordinator. The request can specify a single node, several nodes, or an entire deployment. Upon receiving this request, the coordinator sends messages to the agents on all nodes to be terminated, and the agents send *stop* commands to all of their plugins. Once the plugins are stopped, the agents report their status to the coordinator, and the coordinator contacts the cloud provider to terminate the node(s).

C. Plugins

Plugins are user-defined scripts that implement the application-specific behaviors required of a node. There are many different types of plugins that can be created, such as *service plugins* that start daemon processes, *application plugins* that install software used by the application, *configuration plugins* that apply application-specific settings, *data plugins* that download and install application data, and *monitoring plugins* that validate the state of the node.

Plugins are the modular components of a deployment. Several plugins can be combined to define the behavior of a node, and well-designed plugins can be reused for many different applications. For example, NFS server and NFS client plugins can be combined with plugins for different batch schedulers, such as Condor [18], PBS [26], or Sun Grid Engine [8], to deploy many different types of compute clusters. We envision that there could be a repository for the most useful plugins.

Plugins are implemented as simple scripts that run on the nodes to perform all of the actions required by the application. They are transferred from the client (or potentially a repository) to the coordinator when a node is provisioned, and from the coordinator to the agent when a node is configured. This enables users to easily define, modify, and reuse custom plugins.

Plugins are typically shell, Perl, Python, or Ruby scripts, but can be any executable program that conforms to the required interface. This interface defines the interactions between the agent and the plugin, and involves two components: parameters and commands. Parameters are the configuration variables that can be used to customize the behavior of the plugin. They are specified in the XML request document described above. The agent passes parameters to the plugin as environment variables when the plugin is invoked. Commands are specific actions that must be performed by the plugin to implement the plugin lifecycle. The agent passes commands to the plugin as arguments. There are three commands that tell the plugin what to do: *start*, *stop*, and *status*.

- The *start* command tells the plugin to perform the behavior requested by the user. It is invoked when the

```
#!/bin/bash -e
PIDFILE=/var/run/condor/master.pid
SBIN=/usr/local/condor/sbin
if [ "$1" == "start" ]; then
    if [ "$CONDOR_HOST" == "" ]; then
        echo "CONDOR_HOST not specified"
        exit 1
    fi
    echo > /etc/condor/condor_config.local <<END
CONDOR_HOST = $CONDOR_HOST
END
    $SBIN/condor_master -pidfile $PIDFILE
elif [ "$1" == "stop" ]; then
    kill -QUIT $(cat $PIDFILE)
elif [ "$1" == "status" ]; then
    kill -0 $(cat $PIDFILE)
fi
```

Figure 3: Example plugin used for Condor workers.

node is being configured. All plugins should implement this command.

- The *stop* command tells the plugin to stop any running services and clean up. This command is invoked before the node is terminated. Only plugins that must be shut down gracefully need to implement this command.
- The *status* command tells the plugin to check the state of the node for errors. This command can be used, for example, to verify that a service started by the plugin is running. Only plugins that need to monitor the state of the node or long-running services need to implement this command.

If at any time the plugin exits with a non-zero exit code, then the node’s status is set to failed. Upon failure, the output of the plugin is collected and sent to the coordinator to simplify debugging and error diagnosis.

The plugin can advertise node attributes by writing key=value pairs to a file specified by the agent in an environment variable. These attributes are merged with the node’s existing attributes and can be queried by other nodes in the virtual cluster using `<ref>` tags or a command-line tool. For example, an NFS server node can advertise the address and path of an exported file system that NFS client nodes can use to mount the file system. The *status* command can be used to periodically update the attributes advertised by the node, or to query and respond to attributes updated by other nodes.

A basic plugin for Condor worker nodes is shown in Figure 3. This plugin generates a configuration file and starts the `condor_master` process when it receives the *start* command, kills the `condor_master` process when it receives the *stop* command, and checks to make sure that the `condor_master` process is running when it receives the *status* command.

D. Dependencies and Groups

Dependencies ensure that nodes are configured in the correct order so that services and attributes published by one node can be used by another node. When a dependency exists between two nodes, the dependent node will not be configured until the other node has been configured. Dependencies are

valid as long as they do not form a cycle that would prevent the application from being deployed.

Applications that deploy sets of nodes to perform a collective service, such as parallel file systems and distributed caches, can be configured using named groups. Groups are used for two purposes. First, a node can depend several nodes at once by specifying that it depends on the group. This is simpler than specifying dependencies between the node and each member of the group. These types of groups are useful for services such as Memcached clusters where the clients need to know the addresses of each of the Memcached nodes. Second, groups that depend on themselves form co-dependent groups. Co-dependent groups enable a limited form of cyclic dependencies and are useful for deploying some peer-to-peer systems and parallel file systems that require each node implementing the service to be aware of all the others.

Nodes that depend on a group are not configured until all of the nodes in the group have been configured. Nodes in a co-dependent group are not configured until all members of the group have registered. This ensures that the basic attributes of the nodes that are collected during registration, such as IP addresses, are available to all group members during configuration, and breaks the deadlock that would otherwise occur with a cyclic dependency.

E. Security

Wrangler uses SSL for secure communications between all components of the system. Authentication of clients is accomplished using a username and password. Authentication of agents is done using a random key that is generated by the coordinator for each node. This authentication mechanism assumes that the cloud provider’s provisioning service provides the capability to securely transmit the agent’s key to each VM during provisioning.

IV. EVALUATION

The performance of Wrangler is primarily a function of the time it takes for the underlying cloud management system to start the VMs. Wrangler adds to this a relatively small amount of time for nodes to register and be configured in the correct order. With that in mind, we conducted a few basic experiments to determine the overhead of deploying applications using Wrangler.

We conducted experiments on three separate clouds: Amazon EC2, NERSC’s Magellan cloud [22], and FutureGrid’s Sierra cloud [7]. EC2 uses a proprietary cloud management system, while Magellan and Sierra both use the Eucalyptus cloud management system [24]. We used identical CentOS 5.5 VM images, and the `m1.large` instance type, on all three clouds.

A. Deployment with no plugins

The first experiment we performed was provisioning a simple vanilla cluster with no plugins. This experiment measures the time required to provision N nodes from a single provider, and for all nodes to register with the coordinator.

Table I: Mean provisioning time for a simple deployment with no plugins.

	2 Nodes	4 Nodes	8 Nodes	16 Nodes
Amazon	55.8 s	55.6 s	69.9 s	112.7 s
Magellan	101.6 s	102.1 s	131.6 s	206.3 s
Sierra	371.0 s	455.7 s	500.9 s	FAIL

Table II: Provisioning time for a deployment used for workflow applications.

	2 Nodes	4 Nodes	8 Nodes	16 Nodes
Amazon	101.2 s	111.2 s	98.5 s	112.5 s
Magellan	173.9 s	175.1 s	185.3 s	349.8 s
Sierra	447.5 s	433.0 s	508.5 s	FAIL

The results of this experiment are shown in Table I. In most cases we observe that the provisioning time for a virtual cluster is comparable to the time required to provision one VM, which we measured to be 55.4 sec (std. dev. 4.8) on EC2, 104.9 sec (std. dev. 10.2) on Magellan, and 428.7 sec (std. dev. 88.1) on Sierra. For larger clusters we observe that the provisioning time is up to twice the maximum observed for one VM. This is a result of two factors. First, nodes for each cluster were provisioned in serial, which added 1-2 seconds onto the total provisioning time for each node. In the future we plan to investigate ways to provision VMs in parallel to reduce this overhead. Second, on Magellan and Sierra there were several outlier VMs that took much longer than expected to start, possibly due to the increased load on the provider’s network and services caused by the larger number of simultaneous requests. Note that we were not able to collect data for Sierra with 16 nodes because the failure rate on Sierra while running these experiments was about 8%, which virtually guaranteed that at least 1 out of every 16 VMs failed.

B. Deployment for workflow applications

In the next experiment we again launch a deployment using Wrangler, but this time we add plugins for the Pegasus workflow management system [6], DAGMan [5], Condor [18], and NFS to create an environment that is similar to what we have used for executing real workflow applications in the cloud [12]. The deployment consists of a master node that manages the workflow and stores data, and N worker nodes that execute workflow tasks as shown in Figure 5.

The results of this experiment are shown in Table II. By comparing Table I and Table II, we can see it takes on the order of 1-2 minutes for Wrangler to run all the plugins once the nodes have registered, depending on the target cloud and the number of nodes. The majority of this time is spent downloading and installing software, and waiting for all the NFS clients to successfully mount the shared file system.

V. EXAMPLE APPLICATIONS

In this section we describe our experience using Wrangler to deploy scientific workflow applications. Although these applications are scientific workflows, other applications, such

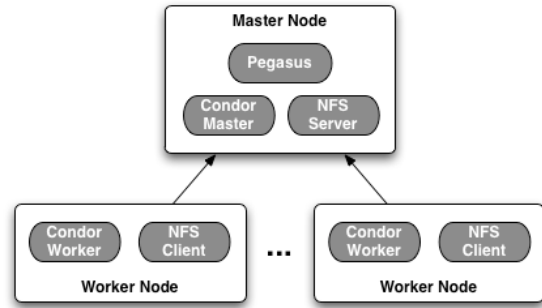


Figure 5: Deployment used for workflow applications.

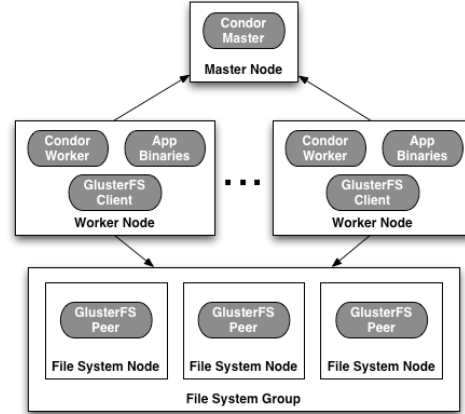


Figure 4: Deployment used in the data storage study.

as web applications, peer to peer systems, and distributed databases, could be deployed as easily.

A. Data Storage Study

Many workflow applications require shared storage systems in order to communicate data products among nodes in a compute cluster. Recently we conducted a study [12] that evaluated several different storage configurations that can be used to share data for workflows on Amazon EC2. This study required us to deploy workflows using four parallel storage systems (Amazon S3, NFS, GlusterFS, and PVFS) in six different configurations using three different applications and four cluster sizes—a total of 72 different combinations. Due to the large number of experiments required, and the complexity of the configurations, it was not possible to deploy the environments manually. Using Wrangler we were able to create automatic, repeatable deployments by composing plugins in different combinations to complete the study.

The deployments used in the study were similar to the one shown in Figure 4. This deployment sets up a Condor pool with a shared GlusterFS file system and installs application binaries on each worker node. The deployment consists of three tiers: a master node using a Condor Master plugin, N worker nodes with Condor Worker, file system client, and application-specific plugins, and N file system nodes with a file system peer plugin. The file system nodes form a group so that worker nodes will be configured after the file system is

ready. This example illustrates how Wrangler can be used to set up experiments for distributed systems research.

B. Periodograms

Kepler [21] is a NASA satellite that uses high-precision photometry to detect planets outside our solar system. The Kepler mission periodically releases time-series datasets of star brightness called *light curves*. Analyzing these light curves to find new planets requires the calculation of *periodograms*, which identify the periodic dimming caused by a planet as it orbits its star. Generating periodograms for the hundreds of thousands of light curves that have been released by the Kepler mission is a computationally intensive job that demands high-throughput distributed computing. In order to manage these computations we developed a workflow using the Pegasus workflow management system [6].

We deployed this application across the Amazon EC2, FutureGrid Sierra, and NERSC Magellan clouds using Wrangler. The deployment configuration is illustrated in Figure 6. In this deployment, a master node running outside the cloud manages the workflow, and worker nodes running in the three cloud sites execute workflow tasks. The deployment used several different plugins to set up and configure the software on the worker nodes, including a Condor Worker plugin to deploy and configure Condor, and a Periodograms plugin to install application binaries, among others. This application successfully demonstrated Wrangler’s ability to deploy complex applications across multiple cloud providers.

VI. RELATED WORK

Configuring compute clusters is a well-known systems administration problem. In the past many cluster management systems have been developed to enable system administrators to easily install and maintain high-performance computing clusters [3,9,29,32,34]. Of these, Rocks [28] is perhaps the most well known example. These systems assume that the cluster is deployed on physical machines that are owned and controlled by the user, and do not support virtual machines provisioned from cloud providers.

Constructing clusters on top of virtual machines has been explored by several previous research efforts. These include VMPlants [17], StarCluster [31], and others [20,23]. These systems typically assume a fixed architecture that consists of a head node and N worker nodes. They also typically support only a single type of cluster software, such as SGE, Condor, or Globus. In contrast, our approach supports complex application architectures consisting of many interdependent nodes and custom, user-defined plugins.

Configuration management deals with the problem of maintaining a known, consistent state across many hosts in a distributed environment. Many different configuration management and policy engines have been developed for UNIX systems. Cfengine [4], Puppet [13], and Chef [27] are a few well-known examples. Our approach is similar to these systems in that configuration is one of its primary concerns, however, the other concern of this work, provisioning, is not addressed by configuration management systems. Our approach can be seen as complementary to these systems in

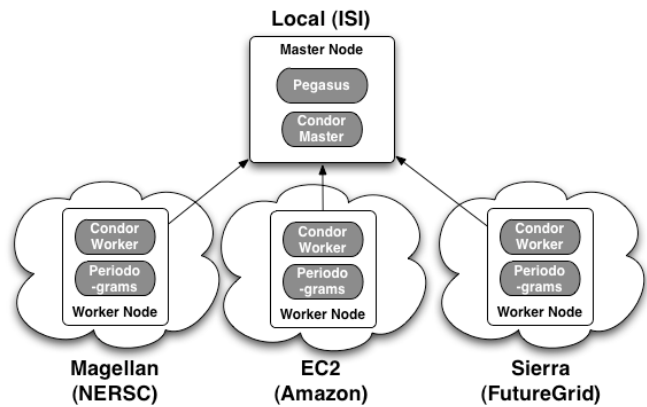


Figure 6: Deployment used to execute periodograms workflows.

the sense that one could easily create a Wrangler plugin that installs a configuration management system on the nodes in a deployment, and allow that system manage node configuration.

This work is related to virtual appliances [30] in that we are interested in deploying application services in the cloud. The focus of our project is on deploying collections of appliances for distributed applications. As such, our research is complementary to that of the virtual appliances community as well.

Our system is similar to the Nimbus Context Broker (NCB) [14] used with the Nimbus cloud computing system [15]. NCB supports *roles*, which are similar to Wrangler plugins with the exception that NCB roles must be installed in the VM image and cannot be defined by the user when the application is deployed. In addition, our system is designed to support multiple cloud providers, while NCB works best with Nimbus-based clouds.

Recently, other groups are recognizing the need for deployment services, and are developing similar solutions. One example is cloudinit.d [2], which enables users to deploy and monitor interdependent *services* in the cloud. Cloudinit.d services are similar to Wrangler plugins, but each node in cloudinit.d can have only one service, while Wrangler enables users to compose several, modular plugins to define the behavior of a node.

VII. CONCLUSION

The rapidly-developing field of cloud computing offers new opportunities for distributed applications. The unique features of cloud computing, such as on-demand provisioning, virtualization, and elasticity, as well as the emergence of commercial cloud providers, are changing the way we think about deploying and executing distributed applications.

There is still much work to be done in investigating the best way to manage cloud environments, however. Existing infrastructure clouds support the deployment of isolated virtual machines, but do not provide functionality to deploy and configure software, monitor running VMs, or detect and respond to failures. In order to take advantage of cloud

resources, new provisioning tools need to be developed to assist users with these tasks.

In this paper we presented the design and implementation of a system used for automatically deploying distributed applications on infrastructure clouds. The system interfaces with several different cloud resource providers to provision virtual machines, coordinates the configuration and initiation of services to support distributed applications, and monitors applications over time.

We have been using Wrangler since May 2010 to provision virtual clusters for scientific workflow applications on Amazon EC2, the Magellan cloud at NERSC, the Sierra and India clouds on the FutureGrid, and the Skynet cloud at ISI. We have used these virtual clusters to run several hundred workflows for applications in astronomy, bioinformatics and earth science.

So far we have found that Wrangler makes deploying complex, distributed applications in the cloud easy, but we have encountered some issues in using it that we plan to address in the future. Currently, Wrangler assumes that users can respond to failures manually. In practice this has been a problem because users often leave virtual clusters running unattended for long periods. In the future we plan to investigate solutions for automatically handling failures by re-provisioning failed nodes, and by implementing mechanisms to fail gracefully or provide degraded service when re-provisioning is not possible. We also plan to develop techniques for re-configuring deployments, and for dynamically scaling deployments in response to application demand.

ACKNOWLEDGEMENTS

This work was sponsored by the National Science Foundation (NSF) under award OCI-0943725. This research makes use of resources supported in part by the NSF under grant 091812 (FutureGrid), and resources of the National Energy Research Scientific Computing Center (Magellan).

REFERENCES

- [1] Amazon.com, Elastic Compute Cloud (EC2), <http://aws.amazon.com/ec2>.
- [2] J. Bresnahan, T. Freeman, D. LaBissoniere, and K. Keahey, "Managing Appliance Launches in Infrastructure Clouds," Teragrid Conference, 2011.
- [3] M.J. Brim, T.G. Mattson, and S.L. Scott, "OSCAR: Open Source Cluster Application Resources," Ottawa Linux Symposium, 2001.
- [4] M. Burgess, "A site configuration engine," USENIX Computing Systems, vol. 8, no. 3, 1995.
- [5] DAGMan, <http://cs.wisc.edu/condor/dagman>.
- [6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," Scientific Programming, vol. 13, no. 3, pp. 219-237, 2005.
- [7] FutureGrid, <http://futuregrid.org/>.
- [8] W. Gentzsch, "Sun Grid Engine: towards creating a compute power grid," 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '01), 2001.
- [9] Infiniscale, Perceus/Warewulf, <http://www.perceus.org/>.

- [10] G. Juve and E. Deelman, "Wrangler: Virtual Cluster Provisioning for the Cloud," 20th International Symposium on High Performance Distributed Computing (HPDC), 2011.
- [11] G. Juve, E. Deelman, K. Vahi, and G. Mehta, "Scientific Workflow Applications on Amazon EC2," Workshop on Cloud-based Services and Applications in conjunction with 5th IEEE International Conference on e-Science (e-Science 2009), 2009.
- [12] G. Juve, E. Deelman, K. Vahi, G. Mehta, B.P. Berman, B. Berriman, and P. Maechling, "Data Sharing Options for Scientific Workflows on Amazon EC2," 2010 ACM/IEEE conference on Supercomputing (SC 10), 2010.
- [13] L. Kanies, "Puppet: Next Generation Configuration Management," Login, vol. 31, no. 1, pp. 19-25, Feb. 2006.
- [14] K. Keahey and T. Freeman, "Contextualization: Providing One-Click Virtual Clusters," 4th International Conference on e-Science (e-Science 08), 2008.
- [15] K. Keahey, R.J. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa, "Science clouds: Early experiences in cloud computing for scientific applications," Cloud Computing and Its Applications, 2008.
- [16] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky Computing," IEEE Internet Computing, vol. 13, no. 5, pp. 43-51, 2009.
- [17] I. Krsul, A. Ganguly, J. Zhang, J.A.B. Fortes, and R.J. Figueiredo, "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing," 2004 ACM/IEEE conference on Supercomputing (SC 04), 2004.
- [18] M.J. Litzkow, M. Livny, and M.W. Mutka, "Condor: A Hunter of Idle Workstations," 8th International Conference of Distributed Computing Systems, 1988.
- [19] P. Marshall, K. Keahey, and T. Freeman, "Elastic Site: Using Clouds to Elastically Extend Site Resources," 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010), 2010.
- [20] M.A. Murphy, B. Kagey, M. Fenn, and S. Goasguen, "Dynamic Provisioning of Virtual Organization Clusters," 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 09), 2009.
- [21] NASA, Kepler, <http://kepler.nasa.gov/>.
- [22] NERSC, Magellan, <http://magellan.nersc.gov>.
- [23] H. Nishimura, N. Maruyama, and S. Matsuoka, "Virtual Clusters on the Fly - Fast, Scalable, and Flexible Installation," 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 07), 2007.
- [24] D. Nurmi, R. Wolski, C. Grzegorzczak, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-source Cloud-computing System," 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 09), 2009.
- [25] OpenNebula, <http://www.opennebula.org>.
- [26] OpenPBS, <http://www.openpbs.org>.
- [27] Opscode, Chef, <http://www.opscode.com/chef>.
- [28] P.M. Papadopoulos, M.J. Katz, and G. Bruno, "NPACI Rocks: tools and techniques for easily deploying manageable Linux clusters," Concurrency and Computation: Practice and Experience, vol. 15, no. 7-8, pp. 707-725, Jun. 2003.
- [29] Penguin Computing, Scyld ClusterWare, http://www.penguincomputing.com/software/scyld_clusterware.
- [30] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M.S. Lam, and M. Rosenblum, "Virtual Appliances for Deploying and Maintaining Software," 17th USENIX Conference on System Administration, 2003.
- [31] StarCluster, <http://web.mit.edu/stardev/cluster/>.
- [32] P. Uthayopas, S. Paisitbenchapol, T. Angskun, and J. Maneesilp, "System management framework and tools for Beowulf cluster," Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, 2000.
- [33] J.-S. Vockler, G. Juve, E. Deelman, M. Rynge, and G.B. Berriman, "Experiences Using Cloud Computing for A Scientific Workflow Application," 2nd Workshop on Scientific Cloud Computing (ScienceCloud), 2011.
- [34] Z. Zhi-Hong, M. Dan, Z. Jian-Feng, W. Lei, W. Lin-ping, and H. Wei, "Easy and reliable cluster management: the self-management experience of Fire Phoenix," 20th International Parallel and Distributed Processing Symposium (IPDPS 06), 2006.