CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1836

A reference model for deploying applications in virtualized environments

Enis Afgan [1], Dannon Baker [1], the Galaxy Team [2], Anton Nekrutenko [3] and James Taylor [1,*,†]

[1] *Biology Department, Math & Computer Science Department, Emory University, Atlanta, GA*
[2] *http://galaxyproject.org/*
[3] *Huck Institute for the Life Sciences, Penn State University, University Park, PA, USA*

SUMMARY

Modern scientific research has been revolutionized by the availability of powerful and flexible computational infrastructure. Virtualization has made it possible to acquire computational resources on demand. Establishing and enabling use of these environments is essential, but their widespread adoption will only succeed if they are transparently usable. Requiring changes to applications being deployed or requiring users to change how they utilize those applications represent barriers to the infrastructure acceptance. The problem lies in the process of deploying applications so that they can take advantage of the elasticity of the environment and deliver it transparently to users. Here, we describe a reference model for deploying applications into virtualized environments. The model is rooted in the low-level components common to a range of virtualized environments and it describes how to compose those otherwise dispersed components into a coherent unit. Use of the model enables applications to be deployed into the new environment without any modifications, it imposes minimal overhead on management of the infrastructure required to run the application, and yields a set of higher-level services as a byproduct of the component organization and the underlying infrastructure. We provide a fully functional sample application deployment and implement a framework for managing the overall application deployment. Copyright © 2011 John Wiley & Sons, Ltd.

Received 9 September 2010; Revised 25 April 2011; Accepted 5 July 2011

KEY WORDS: application deployment; cloud computing; reference model

1. INTRODUCTION

With the maturity of virtualization technologies and high speed networking, coupled with recent research directions[‡,§,¶], it is apparent that dispersed, independent, and individually managed infrastructures are being aggregated into larger units that are uniformly managed [1]. Resource aggregation is a likely trend for organizations to pursue for a variety of reasons including greater infrastructure management flexibility, failover, redundancy, increased resource utilization, and cost reduction [2]. This resource aggregation model also fares well with changes in resource user base, such as change of a project or group size, change in funding, change in data processing demand, and change in infrastructure management. Over the years, interest in this trend has manifested as the deployments of grids and, more recently, clouds [3].

The benefits of aggregated infrastructures are realized because resources can be housed in dense data centers, thus reducing operational costs [4, 5], and access to those can be flexibly controlled

*Correspondence to: James Taylor, Biology Department, Math and Computer Science Department, Emory University, Atlanta, GA, USA.
[†]E-mail: james.taylor@emory.edu
[‡]http://salsahpc.indiana.edu/CloudCom2010/index.html
[§]http://www.thecloudcomputing.org/2010/
[¶]http://www.cloudcomputingconf.org/

through virtualization technologies [6], which enable easier control over the layout of distributed hardware making otherwise static resources and infrastructures much more dynamic [7]. This makes it is possible for a user to acquire a greatly variable number of resources, as required by their current computational demand, and it is in direct contrast to statically managed, traditional infrastructures where a user was strictly limited by the total size of any one resource. Virtualization also allows resource providers to increase resource utilization by effectively managing the infrastructure across a wider user base [8] by providing compute resources in the form of small, preconfigured units called machine images (e.g., [9–11]). As the name implies, a machine image is a preconfigured, customized image of a machine that encompasses underlying hardware, an operating system, and desired system software. Because a machine image can be instantiated any number of times and multiple machine images can be assembled in any desired combination, the underlying infrastructure may be organized and reorganized in any number of ways, yielding unprecedented flexibility of the infrastructure.

To maximize flexibility of the infrastructure and minimize cost, many service providers offer only transient instances (e.g., [12, 13]). This means that any changes made to a live machine image (i.e., an instance) are lost at instance termination. The reason for transient images can be seen as a consequence of frequent failure [14] and high cost [15] of reliable storage, which would be required for persistent instances. Because of the need for service providers to provide cost-effective solutions to the consumers, transient instances are likely to remain a prominent feature for the long term. With such infrastructure characteristics, there is thus a need to persist desired changes beyond the life of an instance. One method for including the changes is to create a new version of the machine image. However, repeatedly creating new machine images represents a continuous disruption to the user's routine and leads to version management issues making it neither desirable nor scalable. This issue is remedied through *persistent storage resources* that are available within the infrastructure in addition to the machine images. Storage resources represent persistent storage units that transcend the life of an instance and they may be used to store any data resulting from computation performed on an instance. Once a compute instance is available, an arbitrary number of storage resources can be associated with an instance and used as needed (e.g., [12]).

With that, aggregated infrastructures based on virtualization provide all of the necessary components to compose a complete system, namely: machine images, storage, and network connectivity (e.g., [11, 13]). These units represent basic building blocks that may be used to construct complex and customized systems on demand. However, from an application and user standpoint, these building blocks are disparate units. They require users to contemplate their usage of the system and for applications to be modified to execute in the new environment. It would be beneficial to transparently coordinate these individual building blocks into a unit that gives the perception of a traditional system to both the users and applications. Such an approach would make the transition toward aggregated infrastructures easier and faster.

Taking advantage of the flexibility of aggregated infrastructures while enabling easy transition for applications and users thus imposes the following two requirements: (1) existing applications should be easily deployable on the new infrastructures and (2) users should not observe any differences in application behavior when they are used on the new infrastructures. The first requirement implies that an application should not need to be modified. Instead, a framework that hides low-level infrastructure details should be in place so that any differences between the traditional infrastructures and the newly emerging ones are abstracted. This enables applications to continue their execution as if they were running in a traditional environment. This requirement helps in promoting the adoption of the described infrastructures by not imposing additional requirements on application developers. Requirement two implies that a user should not need to change how they interact with the application because of the underlying infrastructure changes. Rather, they should transparently enjoy the flexibility enabled by the new infrastructure. This requirement helps in promoting adoption of the described infrastructures by not turning users away. Figure 1 depicts these ideas: (a) users in different research groups access a dedicated application instance over the internet as they are accustomed to; (b) the application instances appear to the users to be running on top of a dedicated infrastructure, albeit, with apparently infinite compute and storage resources; and (c) the resources are in fact virtual resources, which are allocated on demand from a large shared pool. At

DOI: 10.1002/cpe

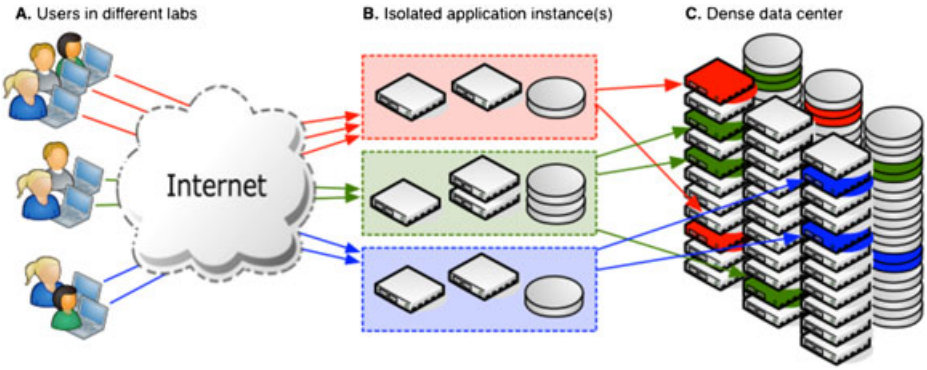

Figure 1. Simplified overview of an aggregated distributed infrastructure and its perception by users.

the same time, applications are executing as they normally would, without any special adjustments for execution in the new environment; the encapsulating framework hides and abstracts the low-level infrastructure details making it appear to the application as if it was running in a static, traditional environment.

In this paper we present an architecture and provide a framework along with a working example that satisfies the two stated requirements. The presented architecture makes use of the basic building blocks provided by an infrastructure provider and coordinates those in a fashion that enables an application to be easily deployed and controlled in the new environment. In addition, the proposed framework enables higher-level services, which are made possible by effectively utilizing the underlying infrastructure components, to be simply exposed to users. This added value has the benefit of attracting users to utilize a given application and the infrastructure. We also provide an implementation of the derived architecture in the form of an application deployment framework. This framework has been tested and is being utilized within the Galaxy project [17, 18] on Amazon's EC2 cloud infrastructure [11]. In this context, a complete and working implementation is available at http://bitbucket.org/galaxy/cloudman/ that can be simply modified to other applications or infrastructure providers.

2. MOVING FROM IAAS TO PAAS

To facilitate adoption of aggregated infrastructures, we have focused on bridging the functionality of the three basic infrastructure types: IaaS, PaaS, and SaaS. On the basis of the two requirements stated in the introduction and coupled with users' desired perception of those infrastructures, we observed a need for a solution that is rooted in and utilizes the low-level, core infrastructural components but simultaneously delivers high-level services. Because aggregated infrastructures may be implemented using a variety of methods and principles, to maintain the widest applicability potential, there is a need to choose the lowest common denominator for the application deployment environment. As a result, focusing on functionality offered by IaaS providers (as opposed to PaaS or SaaS providers), one can rely on the basic computational components that may be expected from a range of IaaS providers (as described in the previous section). This approach minimizes dependencies on expected functionality and is thus applicable to a variety of infrastructure implementations; it minimizes lock-in with respect to the infrastructure and the environment setup. However, these low-level, flexibility-granting components need to be abstracted away from the end user so their productivity is not hindered by the accidental complexities of the technology. Productivity can be improved through automation and delivery of domain-specific tools that simply utilize low-level infrastructures but do not require technical expertise. In other words, there is a benefit in relying only on low-level computational components offered by a range of IaaS providers and yet transparently delivering SaaS type of functionality to the users.

DOI: 10.1002/cpe

2.1. Architecture

In the remainder of this section, we present an architecture built on the principles outlined above. This architecture represents a step toward defining a *universal environment* for deploying a range of applications on aggregated infrastructures. The motivation for this approach is twofold: (1) provide a template for those wishing to migrate their own application to an aggregated infrastructure environment and (2) enable the underlying infrastructure to be interchanged while minimizing changes required to realize the bridge between IaaS and SaaS. By following this architecture, we believe many of the pitfalls of application deployment across aggregated infrastructures can be avoided and several benefits can be realized, as described in detail in the next section.

The presented architecture focuses on encapsulating an existing application into a deployable unit that can be instantiated by an individual user. This approach has benefits for both the application owner and the application user: the application owner is alleviated from having to provision resources on which the application runs or provide details and support about how to install a given application locally. Furthermore, by providing a completely self-contained solution within an infrastructure provider, the application owner does not need to act as a broker between users and the infrastructure provider. They are thus further abstracted from the infrastructure management details allowing them to focus on the application. At the same time, from the user standpoint, a readily deployable application unit allows them to easily and quickly gain access to the desired application. The user is thus not encumbered by having to install and maintain their own infrastructure or installation, nor limited by restrictions imposed by availability of a public deployment of the application; if the application deployment unit exists for multiple infrastructure providers, the user is also free to choose where to run the application. As a result of the self-contained application unit deployment process, the user can independently — and with no system level expertise — utilize IaaS infrastructure while enjoying SaaS usability. This approach creates a separation of concerns at a high level and allows everyone involved to focus on their domain instead of being bogged down by management and maintenance routines and requirements.

To enable packaging of an application deployment unit, the application deployment process needs to be understood. An application is deployed by transferring the entire application (source code, required data, scripts, dependencies) to a virtual machine, where it is then compiled and made available for execution. This deployment process requires steps that often involve user intervention, insight into the internal structure of the application, and familiarity with the various infrastructure technologies and tools. Moreover, once deployed, the application assumes and expects a static environment regarding data storage locations and dependencies. The devised architecture thus has the requirement to encapsulate and continue to provision individual components provided by the infrastructure in a fashion that is expected by the application.

The devised architecture is captured through separation and coordination of otherwise independent components. Figure 2 captures the three components comprising the presented architecture: the machine image, a persistent data repository, and persistent storage resources. The *machine image* is characterized by simplicity; it consists only of the basic services required to initiate the application unit deployment process. In particular, besides selecting the underlying operating system, only the core set of services should be installed on the image itself. This core set of services is comprised of only those services that are required to complete the remainder of the instance boot process. Alongside these services, a contextualization script should be included in the basic machine image. Contextualization is the process of coordinating the machine instance preparation and deployment at runtime [19]. The contextualization script included in the machine image should serve only as an access point to the instance while the contextualization details should be extracted and stored in the persistent data repository. The *persistent data repository* lives independent of the machine image and is used to provide contextualization details, in particular, boot time scripts that define which services should be loaded (note that this does not include downloading and compiling the desired software stack at each boot, see Implementation section). Lastly, *persistent storage resources* provided by the infrastructure should be used as the storage medium for the software stack, namely the application, any required tools, libraries, or datasets the application depends on, and user data. It is furthermore advisable to group the pieces of software into logical units (i.e., application and any

DOI: 10.1002/cpe

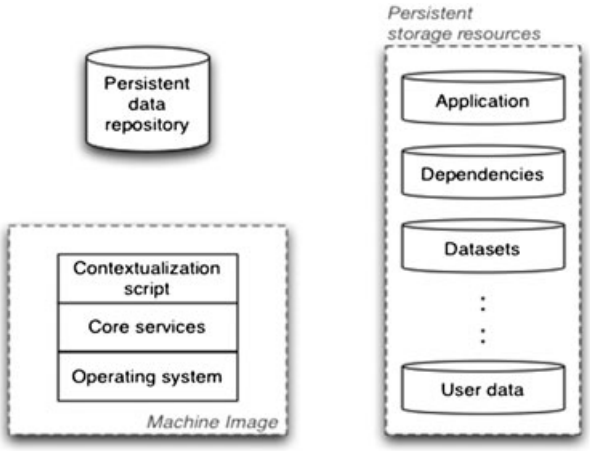

Figure 2. Layout of the basic components comprising the proposed architecture.

dependencies, required datasets, and user data) and create a persistent storage resource for each of the units.

The presented architecture is designed with modularity in mind. As described earlier, creating new versions of a machine image is undesirable, so we believe that machine image customizations should be kept to a minimum and instead the custom components composing an application deployment should reside on a medium that is easier to modify (e.g., persistent data resources in the form of an external data volume/snapshot). Then, at instance setup time during contextualization, those external resources can be associated with the running instance, which happens nearly instantaneously. Through such an approach, an otherwise static machine image can become a customized instance by assembling preconfigured persistent data resources at runtime to accommodate the needs of a changing application. The same machine image may even be reused for different applications. The *persistent data repository*, where the contextualization script resides, is realized as a content delivery service and it only needs to be accessible to the machine image at boot time; it may be provided by a service provider (e.g., S3 within Amazon) or be external to the given infrastructure. Regardless of the implementation, use of the external persistent data repository allows the application developer to modify the image instantiation process by simply updating the script that is stored in the persistent data repository and referenced by the contextualization script on the machine image at boot time.

Similar to the benefits of the persistent data repository, from the architectural standpoint, the *persistent storage resources* are external to the machine image; they are formatted as independent file systems accessible to the system as a whole. As a result, they may be updated and assembled (at contextualization time) in any desired arrangement. From the standpoint of the application being deployed, those resources are perceived as local resources available for immediate consumption (i.e., file systems). This enables the application to utilize those resources without requiring any changes in how it operates, and without the user noticing any difference. Although only a single storage resource could be used for all of the pieces required for deploying the application (e.g., dependencies, datasets), there is a considerable benefit in isolating those into separate logical units and assigning them to individual storage resources. As described in more detail in the next section, such decomposition of individual pieces provides support for updates to the individual pieces and the ability to version the same. In addition, multiple storage volumes may be requested on demand and assembled into a single file system, thus removing physical limitations of any one storage medium that may be otherwise imposed by the infrastructure provider.

Specific benefits relating to the portability, encapsulation, and management of the proposed architecture are twofold: (1) updates to the machine image contextualization process happen without updating the image itself and (2) updates to the application, any dependencies or datasets happen independently of other components. The presented architecture goes a long way toward defining and simplifying the application deployment process for virtualized infrastructures; in addition, it

DOI: 10.1002/cpe

enables a set of higher-level services otherwise not available (see next section). The architecture defines individual components but not the specific assembly of those components. This is because different applications will impose different requirements on their environment, so the needed component composition must be performed specifically for the given application. In subsequent sections we provide an example of how this was achieved and in turn provide a framework that may be reused to perform the same steps. We believe the presented architecture is applicable to a range of applications and will help in deploying almost any application into a virtualized environment.

3. SUPPORT FOR HIGH-LEVEL SERVICES

Supporting the goal of maintaining portability, the presented architecture makes use of only the basic compute infrastructure components. Coordination of these low-level components into a cohesive unit yields a framework capable of delivering high-level services or features to the end user that are a product of the devised architecture rather than explicit implementation or a given infrastructure. These features include: data persistence, managed software updates, and software versioning. The following sections discuss in detail how these features are achieved and the associated benefits.

3.1. Data persistence

Data persistence is paramount to the usefulness of a system and, because applications and users expect data persistence, it needs to be realized transparently for both. The architecture presented in the previous section utilizes persistent storage resources available within an IaaS infrastructure to achieve this. By definition, the persistent storage resources persist independently beyond the life of an instance. They may be attached to a running instance and exposed as a hard drive to the remainder of the system. Such functionality makes it easy to transparently utilize these resources from within the application. In turn, the application requires no modifications to execute in the new environment. Subsequently, through instance contextualization, these resources are assembled into a unit that the application comes to expect and the user is given the notion of constant data availability.

More specifically, individual storage resources are attached to a running instance and a file system is created on top of those. If multiple instances are being deployed into a cluster, those file systems can be shared through a network file system. As explained in the architecture description in the previous section, the individual pieces required to deploy an application are grouped into logical units (e.g., application with its dependent tools and libraries, datasets, user data). Consequently, separate storage resources are acquired and individual units are stored independently from one another. This approach separates, for example, the application core from the user data. As further elaborated in subsequent sections, this approach enables runtime composition of the desired components for a deployment and makes management of those components more isolated. From the perspective of both the system and the user, those individual components are simply different locations in the overall file system. Once a deployed instance is terminated, all resources are released. With the next deployment, through instance contextualization, the same storage resources are attached to the new instance and file systems are mounted, providing all of the expected persistent data as the application starts.

An additional feature of persistent storage resources is that point-in-time snapshots can be created from the storage resources. With such functionality, there is often no need to require all of the users to keep their versions of all the storage resources required by a deployment. Because the application, dependencies, and reference datasets are typically used in read-only mode (i.e., all of the changes and calculations performed by the user are considered user data and thus stored on the respective user data resource), snapshots of a given storage resource can be created by the application developer. These snapshots can then be made public and, at instance boot time, temporary versions of the snapshots are automatically created and attached to the user's instance. Upon instance termination, these created storage resources are deleted. Such an approach can dramatically reduce infrastructural resource requirements and make software updates easier, as discussed next.

DOI: 10.1002/cpe

3.2. Managed software updates

Through the use of contextualization, a persistent data repository, and the composition of individual storage resources we find the ability to perform easy software updates. Composing an application deployment unit as described in the presented architecture is in direct contrast to bundling, or packaging, the application and any required dependencies directly into the machine image. If the application and all of the required dependencies are bundled into the base image, thus creating a *fat image*, any subsequent changes to the boot process, application updates, or simple configuration changes mandate creation of a new machine image. Even if the process of creating a machine image is automated, thus minimizing manual effort, users of the given machine image need to be informed to use the new image. This provides a possible point of failure involving the end user, disrupts the user's routine, and reduces productivity.

The architecture presented here handles the update case much more elegantly. The persistent data repository referenced during the contextualization process defines a single start point for any instance. The boot script that resides in this repository defines the actions that, in turn, define the new instance. As a result, changes to this script are automatically and transparently deployed to all of the new instances. Correspondingly, use of the external storage resources for data storage allows composition of custom instances at runtime. By separating application deployment pieces into logical units and snapshots of storage resources, different systems may be composed by simply joining individual components. For example, if multiple versions of tools are made available as independent storage resource snapshots, at instance boot time the user can be given an option as to which version of tools to utilize on the given instance. Moreover, as the application developer releases new versions of their application or updates to application dependencies, these can be made available as individual storage resources and, through changes to the central contextualization script, transparently utilized by users for all new instances.

Figure 3 depicts this functionality: after the application developer deploys the application in compliance with the described architecture, snapshots of storage resources used by the application and dependent datasets can be created and made public. From then on, each time a user starts a new instance, new storage resources are created from the published snapshot and made available in that particular user's environment. Along with the read-only storage resources, a user data storage resource is created for each machine instance. Because the application is configured to use only user data resources for data that changes, the application and the user have the impression that everything is persistent and operating as it would in traditional environments. When a user terminates an instance, read-only storage resources are released and only user data is preserved. Once the application developer updates the application and is ready to make it public, they simply need to provide a new version of the relevant snapshot. From then on, the user will immediately gain access to the updated application version. Moreover, the update can happen transparently if that is the desired functionality.

3.3. Versioning

On the basis of the notion of snapshots, if multiple versions of the application, datasets, or dependencies exist and are available, it is possible to compose instances at runtime that are customized to an individual user. Customized instances can range from differing availability of tools within the application to a different version of the application in its entirety. These ideas are captured in Figure 4. *Default action* can be defined as using the latest versions of published snapshots. This means that a completely new instance automatically utilizes the newest snapshots to create a live instance for a user. Alternatively, if an instance has been established previously or if functionality to choose from multiple versions exists, *custom action* is enacted. A custom action can simply imply use of the same version of a snapshot as was used for the most recent instantiation of a given instance. This is important because the user data may depend on having a particular version of the application or associated datasets, thus the application should not automatically force the user to use the newest versions of snapshots.

Internally, composition of services is handled through contextualization. At instance boot time, a reference to available snapshots is read from the persistent data repository, and a set of options

DOI: 10.1002/cpe

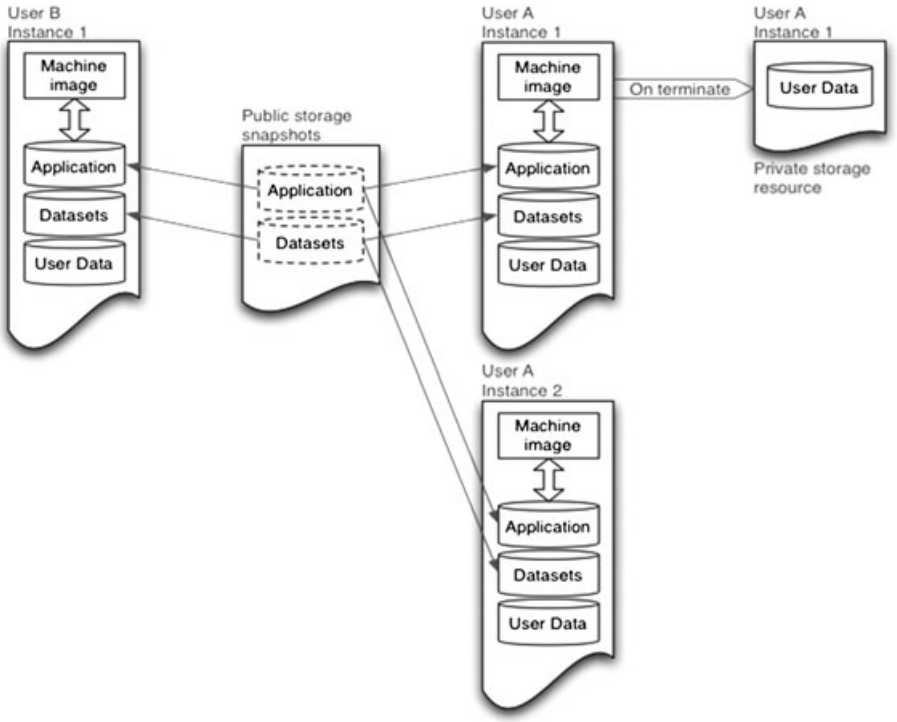

Figure 3. The application developer can manage software updates by simply publishing snapshots of relevant storage resources. At instance boot, those snapshots are used to create user's instance. Because of the separation of instance pieces into logical units, created resources can be used in read-only mode and thus released upon instance termination, thus reducing infrastructure requirements.

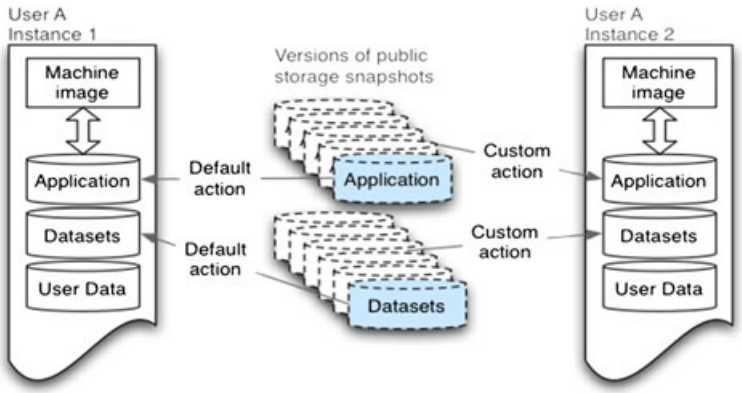

Figure 4. With multiple versions of logical units composing an application deployment, it is possible to automatically assemble customized instances at runtime and provide versioning functionality.

can be presented to the user allowing them to choose which configuration they would like. Again, because of the separation of logical units comprising an application deployment, composition of an instance from these different selections is possible. If there are specific requirements or constraints on any of the components, care should be taken to appropriately present possible options and resolve possible issues. As part of our future work, we will look into defining a component description and dependency harness that can be utilized to address this potential problem.

DOI: 10.1002/cpe

4. IMPLEMENTATION

We implemented the previously described architecture within the context of the Galaxy project[‖] to enable the Galaxy application [17, 18] to make use of aggregated infrastructures. Galaxy is a tool integration framework specifically targeting the biological domain. It focuses on accessibility, extensibility, and result reproducibility by providing an integrated analysis environment where domain scientists can interactively construct multistep analyses, with outputs from one step feeding seamlessly into the next. Any command-line tool can easily be made available through Galaxy, and the underlying data management and computational details are completely hidden from the user, even when dealing with large-scale datasets and high-performance computing resources. The environment transparently tracks every analysis detail to ensure reproducibility, and provides a workflow system for constructing reusable analysis, and an intuitive interface for constructing workflows from an existing analysis. The public Galaxy server (http://usegalay.org) is used by thousands of users every month that run hundreds of thousands of jobs; however, this is a public, shared server. To allow users to become independent of the public server, we wanted to provide a turnkey solution for transparently utilizing aggregated infrastructures so that individual researchers' need for computational and storage resources can more easily be met. Galaxy CloudMan [20] meets this need by providing a reliable solution within these aggregated infrastructures that can additionally scale at runtime as demand mandates (for more information on how to use Galaxy CloudMan see http://usegalaxy.org/cloud).

Recently, cloud computing [21] has emerged as a computational model that is rooted in the concepts of aggregated infrastructure ideas discussed throughout this paper. With many providers of cloud computing services (e.g., Amazon Web Services (AWS), RackSpace, GoGrid, and Mosso), this model presents an ideal scenario to verify the architecture described in Section 2.1. We have selected AWS as the service provider to verify our architecture because it represents a *de facto* standard in this area, showcased by proven availability and reliability of its services. In addition, the infrastructural architecture and the API set forth by Amazon and used by the implementation presented here have been accepted by other cloud infrastructure management projects (e.g., Eucalyptus [13], OpenNebula [22], and Nimbus [23]) allowing for smoother transition as those services become more prominent or infrastructures based on those managers emerge.

A customized Amazon Machine Image was created that is packaged with necessary applications and services to implement the described architecture within AWS. As described in the architecture section, this image contains only the basic boot contextualization script and essential tools, namely: the python interpreter, a message queuing system, and necessary user accounts (i.e., galaxy, postgres, and sge). All of the domain-specific tools and the Galaxy application are obtained at instance boot time from attachable storage resources (e.g., Elastic Block Store (EBS) snapshots and volumes) that are easy to associate with individual users, perform tool updates, and persist beyond lifetime of any one instance. Adopting the described approach was even more important in the context of developing Galaxy CloudMan because the same machine image is instantiated by multiple, independent users. Thus, the image needs to be flexible enough to be transparently tailored for independent users while not containing information specific to any individual user (e.g., [24]).

Internally, CloudMan is implemented as a standalone web application that acts as a manager for the instantiated deployment. To enable composition of multiple instances into a compute cluster, at the implementation level, CloudMan is represented by two services, a master and a worker. The distinction between services is determined at instance boot time, based on the given instance's role. All of the instance contextualization, master-worker communication, and status reporting is performed through a messaging system implemented using the Advanced Message Queuing Protocol standard [25] and using a RabbitMQ[**] server deployed on the master instance.

Figure 5 shows the composition and interaction of individual components described in the architecture in Section 2.1. The persistent data repository is used to store the CloudMan application and information specific to the given cluster. Initially, there is a default public repository that contains

[‖]http://galaxyproject.org
[**]http://www.rabbitmq.com/

DOI: 10.1002/cpe

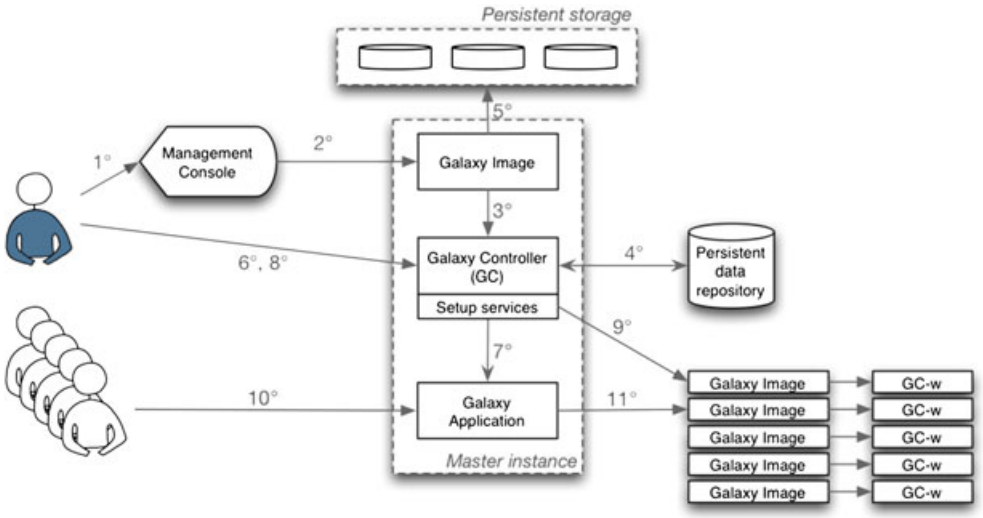

Figure 5. Composition of components composing the presented architecture to enable CloudMan execution in virtualized environment. User initiates interaction with the infrastructure through the infrastructure console manager (1) and instantiates CloudMan machine image (2). CloudMan starts as part of the boot process (3) and contextualizes itself by obtaining needed context from the persistent data repository (4). External storage resources are then attached and configured automatically by CloudMan (5). User provides needed configuration information through CloudMan's web interface (6) and CloudMan configures and starts the Galaxy application. The user then specifies the desired size of the cluster (8), which CloudMan configures by acquiring additional worker instances (9). Once configured, users may use the Galaxy application as any other instance of the application (10) and have their jobs run transparently on the backend resources (11).

only the CloudMan application code that we, the developers, maintain. Once a user instantiates a cluster, an independent repository is created for the given cluster within that user's AWS account. A copy of the CloudMan application along with a listing of which logical volumes and versions thereof are to be used is then stored in the newly created repository. The next time a user instantiates this cluster, information from their repository is used to ensure persistence and reproducibility. Within CloudMan, we decided to realize data persistence with three logical units that are placed on independent storage resources. The following logical units were created: Galaxy application and associated tools, index files needed by tools run through Galaxy, and user data. This approach was chosen because the index data does not change often while the tools do. More specifically, it has the ability to update the application and tools without needing to modify any other data. This reduces the number of snapshots we need to maintain and thus simplifies the overall system management.

With the infrastructure in place, CloudMan automates composition of the individual components at instantiation time. Descriptions of actions that are performed after the created machine image is instantiated, complementing Figure 5, are provided below:

1. User instantiates a master instance.
2. As part of the startup process, start RabbitMQ server, download CloudMan source code from either user's data repository or the public data repository and start the CloudMan web application.
3. Within CloudMan, create attachable storage resources (EBS volumes in case of AWS) from public snapshots for the Galaxy application and necessary index files (i.e., datasets) used by several Galaxy tools, attach them to the running instance, and import existing file systems.
4. Start network file system and enable sharing of relevant directories.
5. Unpack and configure the job manager (Sun Grid Engine (SGE) in the case of CloudMan).
6. Allow a user, through the CloudMan web interface, to configure their cluster by specifying the amount of persistent storage to be used for the user data.
7. Create the user data external storage resource and the appropriate file system.

DOI: 10.1002/cpe

8. Configure PostgreSQL database on the user storage resource to be used by the Galaxy application.
9. Start the Galaxy application. Once Galaxy is ready, enable access to it from the web interface.
10. The user can now start using the Galaxy application. As the need for cluster nodes increases (or decreases), through the web interface, the user may add (or remove) worker instances:

    i. As a worker instance boots, they mount NFS directories and notify the master instance that they are 'alive'; and
    ii. As instances report alive, authentication information to enable SGE to submit jobs is exchanged with the master.

Upon user-initiated termination of a given instance, CloudMan stops all relevant services, terminates worker instances, exports file systems and detaches external data volumes. As noted in Section 3, because storage resources containing tools and index files are not modified during the life of a Galaxy instance, those storage resources are deleted. The next time this instance is instantiated they will be recreated exactly as they were previously. The user data volume is unmounted and detached, but not deleted. Information about this volume is stored in a user-specific persistent data repository (an S3 bucket, in case of AWS) and is read by CloudMan automatically during the instance contextualization process.

Note that although discussed in the context of the Galaxy application, the process described here and the CloudMan code itself can be reused as a separate framework customized to jumpstart other projects wanting to achieve similar functionality. Currently, CloudMan handles composition of lower-level components in a fashion that suits the Galaxy application (e.g., composition of index files, Galaxy application, tools, and user data storage resources). However, the underlying actions of creating and connecting relevant storage resources coupled with mounting the file system or starting of relevant services and applications are generic. Thus, adjusting CloudMan to start with a persistent storage snapshot different than the one with Galaxy tools or starting an application other than Galaxy is a simple matter of parameterization. As a result, CloudMan is an implementation framework for the presented architecture and can thus be used as a comprehensive manager for controlling a cluster deployment on aggregated infrastructures.

Overall, the approach presented here, and demonstrated in the CloudMan application, enables users to gain access to a completely functional and usable service in a matter of minutes — without needing to posses any technical knowledge. In addition, because of the infrastructural flexibility coupled with the provided scaling functionality, users are able to utilize a variable amount of underlying compute resources. Because they are no longer constrained by the limits of the infrastructure, they are free to advance their research.

5. RELATED WORK

In Bradshaw *et al.* [16] and Keahey *et al.* [19] the authors presented the process of instance and cluster contextualization; they focused on the process of contextualization and context data provisioning for an instance. In our approach, we utilize the process of instance contextualization as one part of the architecture but also present a higher-level aggregation of otherwise disparate services to comprise an environment where an application can execute. Furthermore, the approach described in these works requires a centralized broker that coordinates much of the data used during the contextualization process. In our work, we provide a self-contained and standalone solution that requires no external services or brokerage, thus making it independent of a specific service or an infrastructure provider.

In Nishimura *et al.* [26] and Krsul *et al.* [27], the authors described methods for instantiating complete clusters of virtual machines and focused on the performance of the deployed solutions. However, the approaches presented encapsulated the entire application stack into the virtual machine and thus lacked the flexibility provided by contextualization. Furthermore, the focus of the approach presented in those papers were on the higher-level architecture describing and enabling entire applications to be deployed in aggregated infrastructures without requiring changes to the source or usability of a given application.

DOI: 10.1002/cpe

6. CONCLUSIONS

In this paper we describe an architecture that coordinates independent components for the deployment of an existing application within aggregated infrastructure environments. This architecture focuses on utilizing only low-level infrastructure components that can be expected from a wide range of infrastructure providers. The presented decomposition and coordination of the low-level components makes it possible to deploy an application within this framework without requiring application changes. Simultaneously, through the controlled composition of those components, users can utilize a given application without requiring them to change how they interact with it. This approach thus acts as a bridge between the low-level components that can be relied upon for application portability and specific solutions delivered to users. The overall approach provides a model, a working example, and a framework for achieving the same results across a range of applications.

Overall, key aspects of the presented architecture focus on the decomposition of individual pieces required for a complete application deployment into logical units and subsequent mapping of those onto appropriate resources. Such decomposition allows for separation of concerns and good modularity in a deployed solution. This leads to easier deployment of an application but also facilitates easier realization of higher-level features, namely managed software updates and versioning. These features have the benefit of easing application developers' management duties while allowing them to offer a more versatile application with more functionality — without requiring considerably more effort.

ACKNOWLEDGEMENTS

Galaxy was developed by the Galaxy Team: Enis Afgan, Guruprasad Ananda, Dannon Baker, Dan Blankenberg, Ramkrishna Chakrabarty, Nate Coraor, Jeremy Goecks, Greg Von Kuster, Ross Lazarus, Kanwei Li, Anton Nekrutenko, James Taylor, and Kelly Vincent. We thank our many collaborators who support and maintain data warehouses and browsers accessible through Galaxy. Development of the Galaxy framework is supported by NIH grants HG004909 (A.N. and J.T), HG005133 (J.T. and A.N), and HG005542 (J.T. and A.N.), by NSF grant DBI-0850103 (A.N. and J.T) and by funds from the Huck Institutes for the Life Sciences and the Institute for CyberScience at Penn State. Additional funding is provided, in part, under a grant with the Pennsylvania Department of Health using Tobacco Settlement Funds. The Department specifically disclaims responsibility for any analyses, interpretations or conclusions.

REFERENCES

1. Leal K, Huedo E, Llorente I. A Decentralized Model For Scheduling Independent Tasks In Federated Grids. *Future Generation Computer Systems* 2009; **25**(8):840–852.
2. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 2009; **25**(6):599–616.
3. Erlenmeyer M. *Grid and Cloud Computing: Architecture and Services*. MS, Rochester Institute of Technology: Rochester, NY, 2009.
4. Hamilton J. Cost of Power in Large-Scale Data Centers. Available at: http://perspectives.mvdirona.com/2008/11/28/CostOfPowerInLargeScaleDataCenters.aspx [28 November 2008].
5. Neumann D, Baker M, Altmann J, Rana O (eds). *Economic Models and Algorithms for Distributed Systems*. Autonomic Systems: Birkhäuser Basel, 2010.
6. Rochwerger B, Galis A, Breitgand D, Levy E, Cáceres JA, Llorente IM, Wolfsthal Y, Wusthoff M, Clayman S, Chapman C, Emmerich W, Elmroth E, Montero RS. Design for Future Internet Service Infrastructures. *Presented at the Future Internet Assembly 2009*, Prague, Czech Republic, 2009.
7. Sotomayor B, Keahey K, Foster I. Combining Batch Execution and Leasing Using Virtual Machines. *ACM/IEEE International Symposium on High Performance Distributed Computing 2008 (HPDC 2008)*, Boston, MA, 2008; 87–96.
8. Younge AJ, Laszewski Gv, Wang L, Lopez-Alarcon S, Carithers W. Efficient Resource Management for Cloud Computing Environments. *Work in Progress in Green Computing (WIPGC) Workshop at IEEE International Green Computing Conference (IGCC)*, Chicago, IL, 2010.
9. Vijaykrishnan N, Kandemir M, Irwin MJ, Kim HS, Ye W. Energy-driven integrated hardware-software optimizations using SimplePower. *International Symposium on Computer Architecture*, 2000.
10. Menon A, Santos J, Turner Y, Janakiraman G, Zwaenepoel W. Diagnosing performance overheads in the Xen virtual machine environment. *1st ACM/USENIX international conference on Virtual execution environments*, Chicago, IL, 2005; 13–23.

DOI: 10.1002/cpe

11. Amazon. Amazon Elastic Compute Cloud (Amazon EC2), 2010. Available at: http://aws.amazon.com/ec2/.
12. Amazon.com. Amazon Elastic Block Store (EBS), 2010. Available at: https://aws.amazon.com/ebs/.
13. Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D. The eucalyptus open-source cloud-computing system. *Cloud Computing and Its Applications*, Shangai, China, 2008; 1–5.
14. Schroeder B, Gibson G. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? *5th USENIX conference on File and Storage Technologies (FAST 07)*, San Jose, CA, 2007; 1–16.
15. Gibson G, Van Meter R. Network attached storage architecture. *Communications of the ACM* 2000; **43**(11):45.
16. Bradshaw R, Desai N, Freeman T, Keahey K. A scalable approach to deploying and managing appliances. *TeraGrid 2007*, Madison, WI, 2007; 8.
17. Blankenberg D, Taylor J, Schenck I, He J, Zhang Y, Ghent M, Veeraraghavan N, Albert I, Miller W, Makova K, Hardison R, Nekrutenko A. A framework for collaborative analysis of ENCODE data: making large-scale analyses biologist-friendly. *Genome Research* 2007; **17**(6):960–964.
18. Taylor J, Schenck I, Blankenberg D, Nekrutenko A. Using Galaxy to perform large-scale interactive data analyses. *Current Protocols in Bioinformatics* 2007; **19**:10.5.1–10.5.25.
19. Keahey K, Freeman T. Contextualization: Providing one-click virtual clusters. *IEEE International Conference on eScience*, Indianapolis, IN, 2008; 301–308.
20. Afgan E, Baker D, Coraor N, Chapman B, Nekrutenko A, Taylor J. Galaxy CloudMan: delivering cloud compute clusters. *BMC Bioinformatics* 2010; **11**(12):S4.
21. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M. Above the Clouds: A Berkeley View of Cloud Computing. *University of California at Berkeley UCB/EECS-2009-28, February* 2009:10.
22. Llorente IM, Moreno-Vozmediano R, Montero RS. Cloud Computing for On-Demand Grid Resource Provisioning. *Advances in Parallel Computing* 2009; **18**:177–191.
23. Keahey K, Foster I, Freeman T, Zhang X. Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. *Scientific Programming Journal, Special Issue: Dynamic Grids and Worldwide Computing* 2005; **13**(4):265–276.
24. Nishimura H, Maruyama N, Matsuoka S. Virtual clusters on the fly-fast, scalable, and flexible installation. *CCGrid*, Rio de Janeiro, Brazil, 2007; 549–556.
25. A.W. Group. AMQP - A General-Purpose Middleware Standard. ed, p. 291.
26. Nishimura H, Maruyama N, Matsuoka S. Virtual Clusters on the Fly - Fast, Scalable, and Flexible Installation. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, Rio de Janeiro, Brazil, 2007; 549–556.
27. Krsul I, Ganguly A, Zhang J, Fortes J, Figueiredo R. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. *Supercomputing, 2004*, Pittsburgh, PA, 2004; 7.

DOI: 10.1002/cpe