Elastic Site

Using Clouds to Elastically Extend Site Resources

Paul Marshall
Department of Computer Science
University of Colorado at Boulder
Boulder, CO USA
paul.marshall@colorado.edu

Kate Keahey[1,2] and Tim Freeman[1]
[1]Computation Institute, University of Chicago
[2]Argonne National Laboratory
Chicago, IL USA
keahey@mcs.anl.gov and tfreeman@mcs.anl.gov

***Abstract*— Infrastructure-as-a-Service (IaaS) cloud computing offers new possibilities to scientific communities. One of the most significant is the ability to elastically provision and relinquish new resources in response to changes in demand. In our work, we develop a model of an "elastic site" that efficiently adapts services provided within a site, such as batch schedulers, storage archives, or Web services to take advantage of elastically provisioned resources. We describe the system architecture along with the issues involved with elastic provisioning, such as security, privacy, and various logistical considerations. To avoid over- or under-provisioning the resources we propose three different policies to efficiently schedule resource deployment based on demand.**

We have implemented a resource manager, built on the Nimbus toolkit to dynamically and securely extend existing physical clusters into the cloud. Our elastic site manager interfaces directly with local resource managers, such as Torque. We have developed and evaluated policies for resource provisioning on a Nimbus-based cloud at the University of Chicago, another at Indiana University, and Amazon EC2.

We demonstrate a dynamic and responsive elastic cluster, capable of responding effectively to a variety of job submission patterns. We also demonstrate that we can process 10 times faster by expanding our cluster up to 150 EC2 nodes.

Cloud computing, Infrastructure-as-a-Service

I. INTRODUCTION

Typical computing resources, including everything from a single server to a supercomputer, offer a static and finite set of computational, network, and storage capacity to users. Initially, a resource is purchased with an estimate for its peak capacity with the hope that the average load on the resource stays well below that estimate. However, while the computational, network, and storage capacity of the resource is static, the demand is dynamic. There are times when users or system administrators need to find additional resources to meet their demands, e.g. to meet a paper deadline [3], handle the increased need for computation or storage during an experiment, or process increased Website traffic during the holiday season.

Historically, system administrators could only purchase more physical resources when the load of a particular resource increased beyond its maximum capacity. However, it is difficult to estimate the times when the demand will exceed the capacity of the resource, most often it is only realized when the system crashes under the load. This results in user frustration and possibly the loss of user data or business. Purchasing additional physical resources is a very cumbersome process and usually involves a large amount of time and money: nodes must be ordered, delivered to the site, setup and configured, added to the cluster and continually maintained. These steps can take weeks, if not months to complete. If additional resources are purchased it is likely that their regular usage will be below peak usage, leaving resources underutilized or idle while still drawing power and costing the organization money.

With Grid computing [7] it became possible to leverage existing resources at remote sites when additional capacity was required. Although many applications can leverage the Grid model, Grid computing has a number of constraints that have limited its adoption. A fundamental assumption that Grid computing makes is that the control over the mode in which remote resources are provided is with the remote site. Grid users are not typically given root on remote resources for security reasons, and thus it is not always possible to deploy custom software stacks or databases required by the application. Users also face a potentially infinite set of diverse resources that they must port their applications to and validate on. Finally, the Grid doesn't offer a way for users to gain access to Grid resources on demand.

Cloud computing enhances the mechanisms for sharing of remote resources by providing users with control over the remote resources (e.g. control over their configuration). Clouds also provide a virtualized platform for users to create and manage the software stack from the operating system to the applications. This particular type of cloud is known as an Infrastructure-as-a-Service (IaaS) cloud, as opposed to Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS) clouds. The customizability, complete control over the software stack, and on-demand access to IaaS clouds make them an attractive solution to the problem of dynamically extending the resources of a static site to adjust to changes in demand.

Elastically extending sites with cloud resources poses a number of challenges that must be addressed. First, leveraging remote resources highlights the following security concerns: how can we establish trust between the site and the added remote nodes? How can we provide an environment that would allow us to project assumptions underlying security mechanisms within the site onto a wide-area network?

Numerous logistical considerations must be addressed. In particular, remote nodes that are provided on-demand must be automatically configured to join the site and perform their required role (e.g. a cluster worker node). We must also be able to correctly identify when nodes are required, the duration for which they will be needed, how many nodes should join the site, and which cloud providers should be utilized. These decisions must balance the cost of integrating remote nodes from different cloud providers with the need to efficiently process the demand. The cost of integrating cloud nodes is not simply the monetary cost, although that may be one factor, the cost is also the associated overhead (execution, I/O, or network latency) of different cloud providers.

In our research, we propose a model to elastically extend a site by integrating remote cloud resources on demand. Our model is neither application- or cloud-specific, instead we propose an additional layer capable of monitoring the demand of applications and responding by acquiring or releasing cloud nodes. We implement and evaluate an elastic site manager, built on the Nimbus toolkit [11], to dynamically and securely extend existing physical clusters with IaaS cloud resources. Our elastic site manager interfaces directly with local resource managers, such as Torque [5]. We discuss our implementation in detail, including specifics about how we addressed the applicable challenges previously mentioned. We also develop and evaluate policies for resource provisioning on a Nimbus-based cloud at the University of Chicago (UChicago), another at Indiana University (IU), and the Amazon Elastic Compute Cloud (EC2) [2].

We demonstrate a dynamic and responsive elastic cluster, capable of responding effectively to a variety of job submission patterns. We also demonstrate that we can process 10 times faster by expanding our cluster up to 150 EC2 nodes.

The remainder of the paper is organized as follows. In Section II we discuss the general approach of extending resources within a site and present our elastic site model, and in Section III we discuss our elastic site implementation for extending a batch scheduler with IaaS resources. We evaluate our implementation in Section IV, in Section V we discuss the related work in the field, and in Section VI we hypothesize possible avenues for future work. We conclude in Section VII.

II. Approach

In order to respond to changing demand a site could be extended in numerous ways. One option would be to modify individual applications so that they are able to locate, acquire, and utilize additional resources when they need them. For example, a batch scheduler (e.g. Torque) could be modified to boot a set of Virtual Machines (VM) in the Cloud and dispatch jobs to those VMs. However, the obvious downside to this approach is the requirement that every such application or piece of middleware be modified, resulting in applications possessing redundant functionality. A second, more extensible option would be to build an additional layer, separate from the applications, which is capable of measuring their workload and acquiring or releasing cloud resources when needed. This option allows any application to be extended, creating an elastic site.

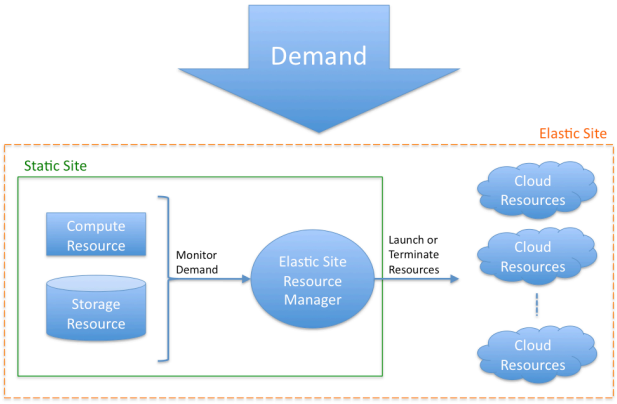

Figure 1. Elastic Site Model

We have created an elastic site model that efficiently adapts services within a site, such as batch schedulers, storage archives, or Web services to leverage elastically provisioned IaaS cloud resources in response to changes in demand as shown in Figure 1. In order for the elastic site to make intelligent decisions, various technical, logistical, and economic differences between cloud providers must be considered. Simply acquiring the maximum resources needed to handle an increase in demand may be technically feasible, however, it may not make sense financially. Cloud nodes must also be preconfigured to perform a role within the site once launched, or they must be configured dynamically as they are launched.

We focus specifically on integrating our elastic site architecture with a batch scheduler where additional worker nodes are dynamically acquired or released from a cloud based on changes in the cluster job queue. This design limits the amount of interaction a user must have with such a system. While a user would undoubtedly be aware that the system leveraged cloud resources to execute his or her jobs, the user wouldn't be required to know the specifics about the underlying cloud, nor how to integrate cloud nodes securely. A user would simply submit their jobs to the queue in order to use elastic site. In the following section we describe the process of manually adding a physical node to a cluster, much of which must be automated for elastic site.

A. *Adding a Worker Node to a Cluster*

Assuming the node has already been purchased and delivered to the site, an operating system must then be installed on the node. Once installation is complete the node will need the necessary patches applied to it and be configured properly, for example, unnecessary services should be terminated. Required software and libraries will need to be installed, most likely including the cluster's version of MPI and common scientific libraries. The node will also need file systems mounted (if necessary). At this point the cluster software will need to be installed on the node and configured to communicate with the cluster head node. If the node will not utilize a shared file system then the cluster software will need

to be configured to support the exchange of input and output data. If the cluster is using any licensed software then licenses for the node would need to be acquired.

In order to dynamically add a node to a site, similar steps must be performed. However, for IaaS cloud nodes, the vast majority of these steps can be performed offline and only once. For instance, we can create one VM image offline installing the operating system, applying patches, configuring services, and adding software and libraries. This base image can then be used on multiple clouds without modification. The remainder of the steps can only be completed when the node is actually deployed on a specific cloud, which we address in the following section.

B. Contextualization

Contextualization of the node can only be performed when the node has been deployed in a particular context. The node must be configured to communicate properly with the site, such as configuring the remote node to exchange input and output data with the cluster head node or mounting a shared file system on the remote node. Additionally, a secure context must be established between the remote nodes and the site resources.

One solution to address contextualization that has been developed by the community is the Nimbus Context Broker [13]. Generally, the Context Broker is a Web Service that facilities the secure exchange of context-specific information (e.g. IP addresses, SSH keys, etc.) between all appliances deployed together in a specific context. Each appliance contains a contextualization agent that securely communicates with the Context Broker on boot and configures the appliance to fulfill a specific role within the context. For example, a cluster of nodes may be launched in the cloud where one node is configured as a cluster head node and the remaining nodes are configured as compute nodes. The contextualization agent in each node provides the node's SSH key and IP address to the Context Broker, which then communicates it to the other nodes. Once a node receives the information for the other nodes, the contextualization agent completes the configuration of the node as either a compute node or the cluster head node, depending on the role that was specified.

C. Architecture and Policies

Elastic site dynamically adds and removes worker nodes from the cluster by monitoring the cluster job queue. As part of elastic site we have developed a queue sensor that examines the cluster job queue and maintains a complete picture of the queue, including total number of jobs in the queue, the number of running jobs, the number of queued jobs, and total job queued time. The queue sensor also collects job-specific information, such as walltime for individual jobs.

Elastic site uses various policies to determine when to boot additional VMs in the cloud or terminate them based on the information provided by the queue sensor. The policies form the core decision-making process of elastic site. The policies also use an estimated waste time that should be gathered for possible cloud providers. The estimated waste time is simply the sum of average startup times and shutdown times for a particular VM image on a cloud. This waste is considered in the decision making process implemented by policies. We have created three initial policies for elastic site:

- *On demand*: The on demand policy is very basic: when a new job is queued, the policy boots one VM. When the queue is empty the policy terminates idle VMs.
- *Steady stream*: The steady stream policy assumes that there will potentially be a "steady stream" of jobs arriving in the queue, thus it always leaves at least one VM running. By leaving one VM running this policy is able to avoid the thrashing caused by the on demand policy when a job arrives shortly after the last VM has been terminated. The steady stream policy also boots additional machines when the total queued walltime becomes greater than five times the estimated waste time of the particular cloud. However, the policy only starts one machine at a time, i.e., it waits until a VM has finished booting before it decides to boot another VM. The policy terminates any additional machines when the total queued walltime drops below three times the estimated waste time of the cloud. With this policy the cluster is able to respond to work immediately, however, it is the most conservative to adjust to changes in demand.
- *Bursts*: The bursts policy is intended for jobs that arrive in bursts. Once a burst arrives we want to boot enough machines so that the estimated waste time is balanced appropriately with the amount of work in the queue. Therefore, to calculate the number of VMs to launch, we divide the total wall time of all queued jobs by two times the estimated waste time. If the integer division results in zero then a single VM is booted.

We believe that many other job arrival patterns also fall into one of these three broad categories. For instance, sporadic job arrivals should use the on demand policy or bursts policy. The bursts policy will typically boot fewer VMs than the on demand policy, however, if each of the jobs has a lengthy runtime then the jobs should perhaps be assigned to their own VMs, as attempted by the on demand policy. Furthermore, it should be noted that these policy definitions are an initial test set and not a comprehensive set encompassing all possible job patterns. Our intent is not that these policies would prove to be the best possible method for managing VMs, but instead that they would provide a viable starting point in our examination of an elastic cluster. Specifically, in the case of the steady stream policy our choice to start additional VMs when the total queued walltime is greater than five times the estimated waste time and terminate VMs when the total queued walltime drops below three times the estimated waste time is an initial attempt to create a responsive but not wasteful policy aimed at a particular job submission pattern. We expect that more extensive examination will result in further adjustments to these bounds.

In the following section we discuss implementation specific details for our elastic site architecture.

III. Implementation

Our elastic site manager, written in Python, is built upon a Linux cluster and the Torque batch scheduler. The elastic site

manager monitors the job queue and responds by either launching a machine in the cloud, terminating one, or doing nothing as shown in Figure 2. Once a cloud machine has booted it joins the cluster as a worker node. If the cloud node operating system and libraries do not match those of the head node, applications can be linked statically, precompiled and deployed on the cloud nodes and saved into the image, or automatically compiled on the cloud node at deployment time.

We make a number of simplifying assumptions in our prototype implementation. First, cloud nodes join as independent remote nodes; we have not deployed an overlay network, allowing the nodes to communicate with each other. Thus, these nodes are only capable of running serial jobs or small parallel jobs, limiting them to the execution of small parallel programs or many serial programs, such as workflows often categorized as Many Task Computing (MTC) [16]. Second, we do not mount a shared file system on the remote nodes so all data must be transferred in and out from the node. Currently this is done via Torque's use of SCP for file transfers.

A. Leveraged Technologies

Our elastic site implementation relies on IaaS technologies such as EC2 [2] and the Nimbus Workspace Service [11] deployed on Science Clouds [19]. The Nimbus Workspace Service is responsible for deploying nodes on the Nimbus cloud, as requested by the cloud client. The Workspace Service initiates the transfer of the VM image from the storage pool to the nodes. Once the VM image has been deployed, the Workspace Service begins the boot process. As the nodes begin to boot they enter the contextualization phase.

We rely on the Nimbus Context Broker [13] to provide contextualization of the nodes. The Nimbus Context Broker provides a secure mechanism for dynamically contextualizing a set of virtual appliances. We use the Context Broker to create a trusted environment between the newly deployed cloud worker nodes and the cluster head node.

B. Adding a Resource to a Site Dynamically

Before we add a cloud resource to a site, we first create the VM image and save it in the different clouds we will be utilizing. We install all of the necessary software and libraries as well as the Torque client software. Torque is free open-source software, so there is no need to acquire licenses for the cloud nodes. For simplicity, the Torque client (pbs_mom process) is preconfigured to join the cluster head node on boot. If there is an external firewall between the cloud and the cluster head node, the necessary ports must be opened for Torque and SSH to function properly. Opening ports in an external firewall can be especially problematic if there isn't a static range of IP addresses that cloud nodes will be associated with. One alternative is to allow traffic from any IP over the necessary ports to reach the head node. A host-based firewall can then be used on the head node to open access dynamically to the cloud nodes. The elastic site manager can interface directly with host-based firewalls to open communication with the cloud nodes. The current implementation includes support for iptables.

The Nimbus Context Broker handles the transfer of the SSH keys from the booted cloud nodes to the cluster head node. Elastic site then adds the keys (and associated hostnames

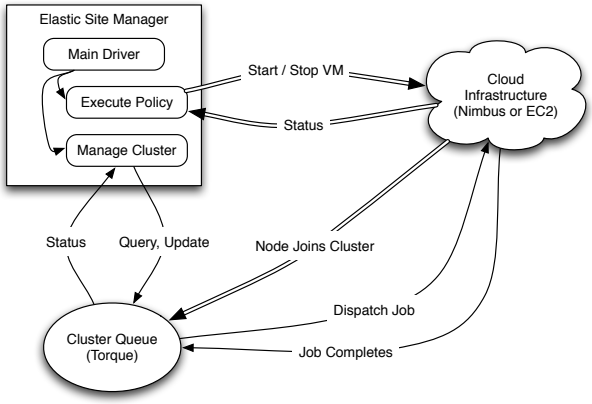

Figure 2. Elastic Site Implementation

and IP addresses) to the global ssh_known_hosts file. For simplicity, we preconfigured the cloud image to automatically trust the cluster head node.

Elastic site uses the Nimbus cloud client to launch or terminate nodes on Nimbus-based clouds or EC2. As this is an initial prototype, elastic site also uses Torque command line programs to monitor the queue and available nodes, as well as dynamically add and remove nodes from Torque.

C. Implementation of the Queue Sensor and Policy Modules

Elastic site runs in a loop periodically examining the job queue, executing a policy, and performing cluster management functions, such as terminating nodes that have been flagged for shutdown by the policy. Due to the time required to launch a VM, we have created a thread pool to launch machines in parallel. The threads (default of 10) work from a single “deploy node” queue to launch machines. The short and consistent time required to terminate a machine allows the main elastic site thread to terminate nodes serially, when needed. In order to avoid overloading the cluster head node with rapid calls to system commands (e.g. qstat and pbsnodes), elastic site sleeps for a duration specified by the cluster administrator. We have found that sleeping for 10 to 15 seconds is a reasonable amount of time to limit the impact on the head node as well as provide a very responsive and dynamic cluster.

We have implemented a queue sensor to monitor the Torque queue and gather job information. The queue sensor parses the output of Torque's qstat command and stores the information in a Python queue object that we have created. We have also created a Python cluster object, which stores all of the relevant information about the cluster, such as the number of running cloud nodes and the number of cloud nodes that are available for work. The cluster object also contains methods for manipulating the cluster, such as launching or terminating VMs.

Policies are implemented as individual Python modules. Policies have access to the cluster and queue objects and can use this representation of the system in order to determine if additional nodes are needed or if nodes may be terminated. The policies themselves are responsible for directly manipulating the cluster object by calling methods to either launch VMs or

schedule them for termination. When a node is scheduled for termination it is not terminated immediately, instead the node is flagged offline by the Torque 'pbsnodes -o' command, allowing a running job to complete. A node flagged offline is not assigned additional work by Torque, which then allows the elastic site program to terminate the node once it becomes idle. This avoids the problem of killing the node prematurely and possibly losing data. Currently, the estimated waste time is represented as a single number, in seconds, which is the sum of the time to boot a VM image on a particular cloud and the time to terminate the VM. This number can simply be a best guess to help policies make more intelligent decisions. The elastic site manager doesn't currently support a mechanism to specify numerous estimated waste times, depending on the number of nodes booting or differing VM image size, both of which may impact boot time in cloud environments.

The policy framework of elastic site is meant to be extensible, allowing administrators to customize or define their own policies to fit the needs of their users. The primary reason we choose to implement policies as Python modules instead of creating our own policy definition language was to minimize the learning curve for administrators adopting our solution. We find that very detailed and customized languages are often too complicated and frustrating for users to learn for a single purpose, diminishing the desire of users wishing to adopt our implementation of elastic site. Python is a widely used language that is easy to learn and robust enough to define any policy, no matter how simple or complicated.

IV. Evaluation

Our evaluation of elastic site consists primarily of a comparison of the three different policies in an attempt to maximize job turnaround time while minimizing thrashing (constantly launching and terminating VMs) and idle VMs (VMs running with no available work). Comparing the performance of different cloud providers, and by extension, the underlying node hardware, network, and software deployed by those providers is beyond the scope of our work. These topics have been addressed elsewhere in the literature: [6], [9], [10], [15], and [22].

The evaluation environment consists of a cluster head node at the University of Colorado at Boulder. The head node has two 2.4 GHz Intel Xeon processors with hyper-threading and 6 GB of RAM. We provide an initial comparison between the Nimbus cloud at the UChicago and the cloud at IU; however, we chose to use the UChicago cloud for the remainder of the analysis since it provided a more consistent environment when booting multiple VMs simultaneously. To demonstrate that our elastic site implementation scales beyond a handful of nodes we use Amazon EC2.

The IU cloud consists of 2 nodes. Each node has 2 Quad Core Xeon processors with 32 GB of RAM and a single SATA disk. The Nimbus cloud at the University of Chicago has 16 nodes, each with two 2.2 GHz AMD64 processors and 4 GB of RAM. Each node also has a local 80 GB IDE disk. The storage repository is exposed as a GridFTP service running on the service node. For our analysis with the Nimbus cloud at UChicago we specified a maximum of 10 cloud nodes. Thus, a policy may choose to deploy up to 10 nodes, but no more.

To demonstrate scale we use Amazon EC2 small instance machines, without specifying a maximum limit on the number of nodes available to the elastic site. Small instances contain 1 virtual core (equivalent to a 1-1.2 GHz 2007 Xeon processor), with 1.7 GB of memory, and only what Amazon denotes as "moderate" I/O.

As a metric, we define *elapsed workload time* to be the elapsed time between the arrival of the first job in the workload to the completion of the final job in the workload. As another metric we define *overhead* to be the amount of time VMs are active but not running jobs. Startup and termination time are considered overhead, as well as idle run time. Finally, we define the metric *queue wait time* to be the amount of time a job must wait in the queue before it begins execution. This includes the time from when the job was first submitted until it is dispatched by Torque to the worker node and begins running.

In addition to these metrics we consider the reactiveness of the three policies. We define the reactiveness of a policy to be its ability to respond quickly and effectively to changes in demand. In our analysis we observe the reactiveness of all three policies for three unique workloads that consist of job arrival patterns specific to the policy.

Our workloads use short running jobs because of our interest in evaluating and analyzing the overhead and reactiveness of the policies. Longer running jobs simply amortize the boot and termination overhead of the VMs, thus a two minute boot time is basically irrelevant if you are running a 24-hour job.

Jobs are submitted directly to the Torque queue on our cluster head node. A job consists of a small C program that sleeps for the desired time. The only input is a parameter passed to the program specifying the sleep time and the only output is the small error and output files generated by Torque, which are transferred back to the head node via SCP. The job walltime is set to be one minute more than the runtime of the job.

A. *Understanding the Characteristics of Launching and Terminating Cloud Nodes*

Prior to our evaluation of the policies we must first understand the basic characteristics of the cloud provider we intend to use. We are primarily interested in the amount of time required to start a VM and shutdown a VM. We have found that termination time is often under 10 seconds and relatively consistent across clouds since it involves a Web service invocation which returns after the termination process has begun. The startup time, however, varies between providers depending on the underlying hardware and network as well as the configuration of the cloud. For example, if a cloud caches a recently used VM image on the nodes it will achieve much quicker boot times on average than if the VM image must be transferred from a storage pool to the nodes for every boot. The number of nodes also impacts startup time as well as the size of the VM image. We have found that these are primarily related to the network between the nodes and the storage pool: a large VM image takes longer to transfer and multiple nodes booting cause contention on the network. Another factor that may impact boot time is the amount of configuration that must be

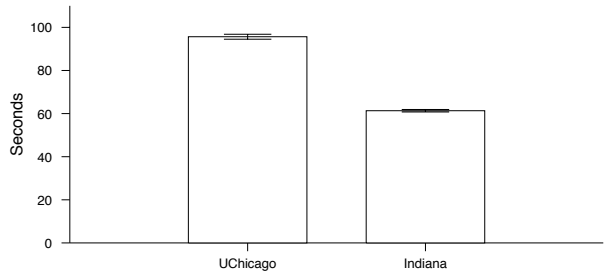

Figure 3. Comparison of startup times (with standard deviation) for 1 VM on the Nimbus clouds at the University of Chicago and Indiana University.

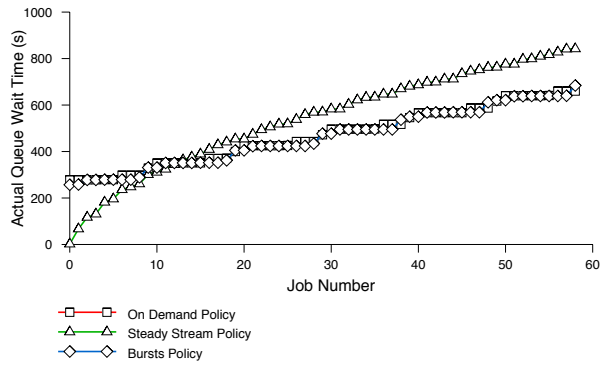

Figure 4. Actual queue wait time, in seconds. All jobs are one minute jobs with two minute walltimes, submitted simultaneously.

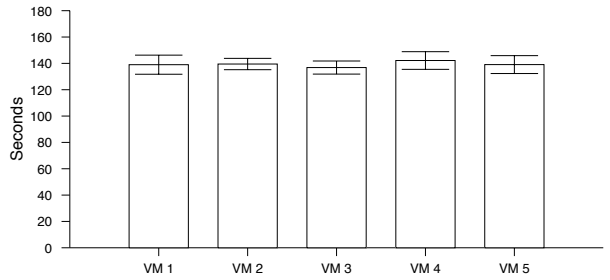

Figure 5. Startup times of 5 VMs (with standard deviation), launched simultaneously, on the Nimbus cloud at the University of Chicago.

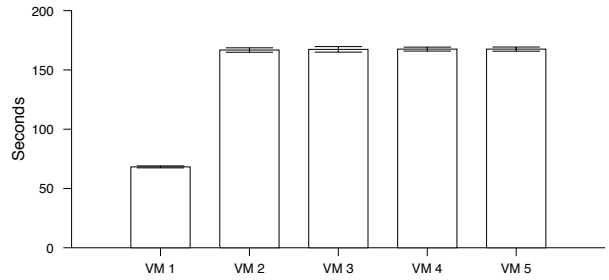

Figure 6. Startup time of 5 VMs (with standard deviation), launched simultaneously, on the Nimbus cloud at Indiana University.

done during the contextualization phase, especially if information must be exchanged between large numbers of nodes. Lastly, the demand on the cloud at a given time can also impact the responsiveness of the cloud. A heavily utilized cloud will have more network contention as many users attempt to boot nodes. The cloud may also oversubscribe nodes to meet heightened demand.

These differences in boot time are clearly demonstrated by a comparison of the boot time on the UChicago cloud vs. the IU cloud as shown in Figure 3. The IU cloud can boot a single VM approximately 34 seconds quicker than the UChicago cloud. However, interestingly, when both clouds boot five VMs, the UChicago cloud (Figure 5) slows down consistently across all five VMs whereas on IU (Figure 6) one VM still boots around 60s. The remaining four VMs on IU slow down significantly more, talking almost 20 seconds longer than all five UChicago VMs. We suspect that the unique behavior on the IU cloud is due to the underlying hardware. In particular, each of the two nodes only has a single SATA disk. On both the UChicago cloud and the IU cloud we observed termination times of 6 seconds plus or minus a second.

B. Evaluation of the Policies

We have devised two workloads to evaluate the metrics. The first workload consists of a submission of 20 60-second jobs. This workload demonstrates a potential use case for the bursts policy. The second workload consists of submitting 10 60-second jobs 30 seconds apart followed by five minutes of sleeping. Then 10 120-second jobs are submitted 10 seconds apart. This workload is considered a potential use case for the steady stream policy. As a baseline we also submit these workloads to a cluster in which, prior to submission, we launch 10 machines and integrate them into the cluster; this is referred to as the dedicated policy.

The elapsed workload time is shown in Figure 7. The dedicated policy processes both workloads in the least amount of time; this is to be expected since there are 10 VMs booted and available when the jobs are submitted. The slight differences between the on demand policy and the bursts polices are interesting: both policies attempt to boot enough VMs to process the jobs with the on demand policy simply booting one VM per queued job (until 10 VMs have been launched) whereas the bursts policy weighs the total queued walltime with the estimated waste time for launching a VM. Because 20 jobs are submitted in workload 1 the on demand policy boots the maximum number of VMs, 10, and the bursts policy only boots 6 VMs yet processes the workload in less time. Booting more VMs takes additional time so the bursts policy is able to start processing the workload before the on demand VMs finish booting. The elapsed workload time for these policies is reversed under workload 2. In this case both policies are able to process the first set of jobs and terminate the VMs, when the second batch of 10 120 seconds arrive the on demand policy boots 10 VMs, one machine for each job,

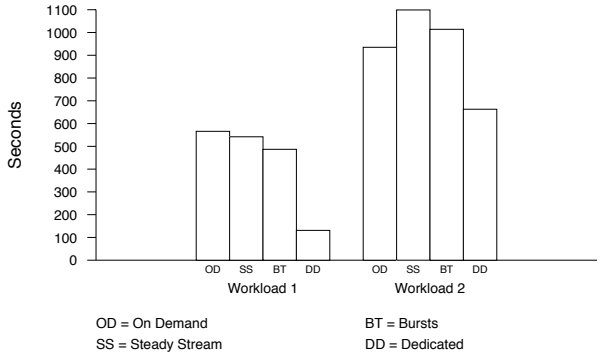

Figure 7. Elapsed time (seconds) of the workload, from submission of the first job to completion of the final job. For the dedicated policy 10 VMs are started prior to submission and run for the duration of the test.

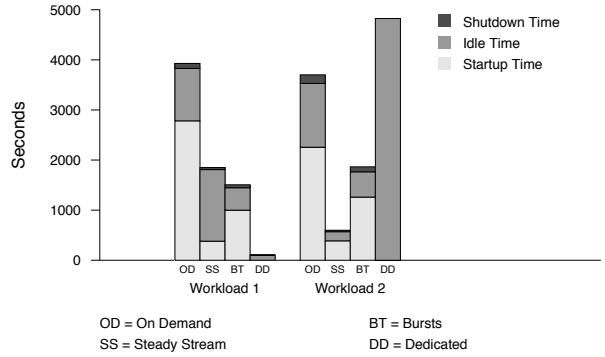

Figure 8. Total overhead per policy. Overhead shown is the sum of the overheads for all VMs associated with each workload and policy.

and the bursts policy boots seven VMs. Because of the longer runtime, 10 VMs are able to process the workload faster.

Figure 8 shows the corresponding overhead for the two workloads. The overhead presented in the figure is the sum of the overhead for all machines utilized. It should be noted that the dedicated case for workload 1 is somewhat misleading. Overhead is only calculated as the overhead that is incurred during the actual test, which is the duration from the time the first job is submitted to the time the final job completes. Understandably, the dedicated policy, with 10 running VMs, is able to quickly process the entire workload with very minimal overhead. Workload 2 is a more accurate representation of the overhead associated with a dedicated set of always-on machines where there may be large periods of time that numerous machines remain idle.

Perhaps the most interesting comparison is between the steady streaming policy overhead and run time for workload 2. This is a perfect example of the need to balance job turnaround time with system overhead. Even though the steady stream policy achieves minimal overhead, it takes the most time to process the workload. The reason for this is due to the fact that the steady stream policy is the most conservative policy when adjusting to changes in demand, thus, to process the workload it uses the least number of VMs, which reduces unnecessary overhead booting extra VMs, but it also means that the workload takes longer to process. Though none of our policies out perform the dedicated policy, we are able to achieve lower levels of overhead than the dedicated policy as demonstrated by workload 2.

Figure 4 shows the queue wait time for the three policies for a test similar to workload 1, we submitted an additional 40 jobs to show the contrast between the three policies. Initially jobs have the shortest queue wait times when using the steady stream policy since one machine is constantly running, awaiting work. Both the on demand and bursts policies must first boot nodes before jobs can begin executing. This demonstrates the conservative aspect of the steady stream policy. While the bursts and on demand policy boot enough machines to address the total demand in the queue, the steady stream policy slowly processes the jobs and boots one VM at a time until it determines that no more machines are needed. Thus, in the case of the steady stream policy we see a point where the queue wait time of the jobs exceeds the wait time of those being processed by the on demand and bursts policy.

C. Reactiveness of Policies

To examine the reactiveness of the policies we devised three reactivity tests, each with unique job submission patterns. The first test emphasizes the sporadic nature of job arrivals addressed by the on demand policy. Ten jobs, each with a runtime randomly chosen between one and ten minutes are submitted at random times over the course of an hour. The second test, aimed at the steady stream policy where jobs arrive in a relatively consistent and steady manner, consists of a 15 minute interval where one to ten minute jobs arrive three minutes apart, followed by a 15 minute period with only two one to ten minute jobs. The final 30 minutes of the test is comprised of one-minute jobs arriving once per minute. The third test is comprised of bursts of job submissions: 40 one to five minute jobs are submitted, followed 20 minutes later 15 ten minute jobs, which are followed 20 minutes later by 60 one minute jobs.

Figure 9 clearly shows the ability of the on demand policy to react to the queue by matching the number of VMs with the number of jobs. The one to two minute delays between jobs and VMs is the time that the VM is booting. Figure 10 demonstrates the ability of the steady stream policy to use a minimum number of VMs to constantly process jobs, while booting additional VMs to process additional jobs when needed. Finally, in Figure 11 the bursts policy boots the necessary number of VMs to process bursts of jobs, terminating the VMs as the queue drains. Due to the large number of jobs submitted the policy boots 10 machines for each bursts and then terminates them when the burst of jobs has been processed. In Figure 11, the running VMs in the second burst of jobs overlap with jobs from the third burst because the VMs were flagged to be terminated prior to the arrival of the third set of jobs. Once a VM has been flagged for termination (i.e. the policy determines that the VM should be terminated

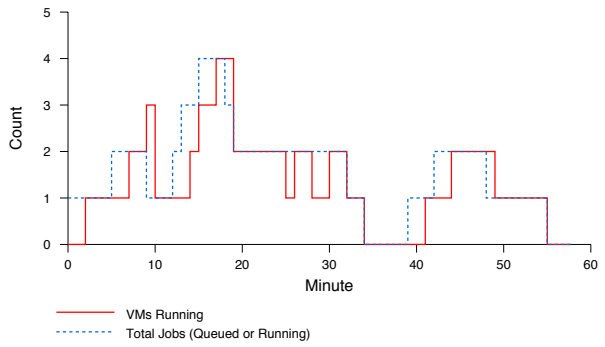

Figure 9. Reactiveness of the On Demand Policy

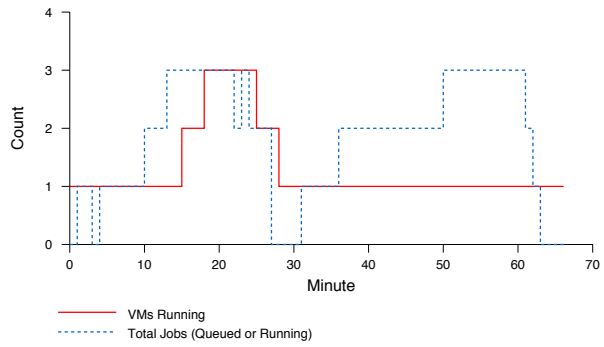

Figure 10. Reactiveness of the Steady Stream Policy

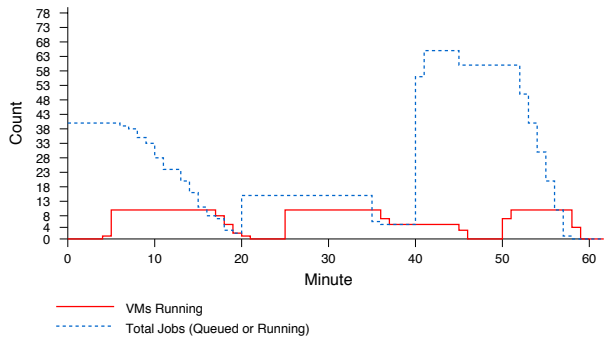

Figure 11. Reactiveness of the Bursts Policy

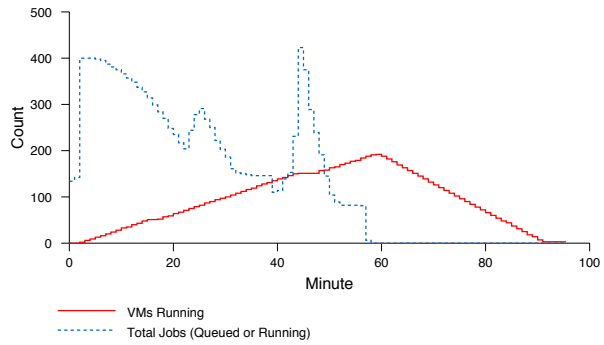

Figure 12. Reactiveness of the Bursts Policy on Amazon EC2

when it has finished running its assigned jobs), our current system does not allow the VM to be "unflagged." Instead, the machine is terminated and another VM must be booted in its place if there is enough work remaining to justify additional resources.

D. Scalability of Elastic Site

To demonstrate the scalability of our elastic site implementation we submit the third reactivity test with 10 times the number of jobs to an elastic cluster with EC2 compute nodes. The workload consists of 400 one to five minute jobs submitted initially and 20 minutes later 150 ten minute jobs are submitted, followed 20 minutes later by 600 one minute jobs. We do not limit the number of VMs available to the elastic site manager. We used the bursts policy for this test.

In switching to EC2 we made one change to our default configuration: we reduced the number of VM launching threads in the thread pool to six, down from 10, in order to reduce the strain on our cluster head node. Ten threads were manageable when we only used the Nimbus cloud client and it was only possible to boot a total of 10 VMs. However, the addition the EC2 Java-based tools and the possibility to launch machines continually, for the duration of the test, produced a noticeable and lasting increase in load on the head node.

Prior to test submission we launched a handful of VMs on EC2 to gather startup and shutdown times for EC2 nodes. EC2 exhibited much more inconsistent boot times than our Nimbus clouds. We launched a handful of VMs and observed boot times varying from 74 seconds to 205 seconds. Termination invocation times were consistently around three to four seconds. Our evaluation with EC2 (Figure 12) shows that elastic site able to process 10 times the number of jobs within 60 minutes, similar to our much smaller test case on the UChicago cloud. However, EC2 grows up to 151 VMs in order to accomplish this. We see an increase in approximately 15 times the number of VMs to process 10 times the work over the 60-minute interval.

V. Related Work

Related work can broadly be grouped into two categories: solutions that dynamically adapt to changing demand by acquiring or releasing resources and solutions that integrate with cloud computing resources to provide additional compute or storage capacity when needed.

VioCluster [18] is a solution that adjusts dynamically to demand by borrowing and lending machines between different clusters. Typically, these clusters would be standard physical clusters where borrowed machines are provided as VMs on top of available physical nodes. Thus it is possible for a domain to be comprised of physical and virtual nodes. The

implementation is based upon Portable Batch System (PBS)-based clusters and User Model Linux (UML). Violin provides a network overlay allowing the VMs of a remote domain to communicate directly with physical nodes in the local domain. VioCluster calculates whether to borrow machines (or whether it has machines available to lend) by counting the number of requested nodes in the queue and subtracting the number of available machines in the cluster. If the number is positive then this is the number of machines that need to be borrowed, if the number is negative then it is the number of machines the cluster has available to lend. Elastic site doesn't borrow or lend machines; instead we only acquire nodes from the cloud when needed. The primary advantage of utilizing the cloud is its ability to offer a seemingly infinite number of resources, on demand, to anyone at anytime. There is no need to identify other independent clusters, which may or may not contain available nodes.

Ruth et al. [17] create an adaptive environment of VMs that is able to adjust the environment based on measuring the current load within the VMs. Their adaptation manager can make per-host CPU and memory adjustments, migrate VMs to different nodes, or move an entire virtual environment to another set of resources. Our approach extends existing resources with cloud resources based on overall demand, which differs from their focus on creating individual, autonomous, adaptive environments capable of dynamically adjusting to load within the VMs.

Murphy et al. [14] create a system to dynamically provision virtual organization clusters. The system is similar to our implementation in that they monitor a job queue and spawn VMs to process the jobs. The authors' solution creates virtual clusters on Grid resources to process Condor jobs. Our work differs in several key aspects. We leverage cloud resources to extend site resources. Our model is generic and can be applied to any datacenter service even though our implementation specifically extends the Torque scheduler with cloud resources. We also create a handful of policies to dynamically respond to demand in an attempt to maximize job turnaround time and minimize overhead. The authors' create and evaluate a single policy that corresponds to our on demand policy.

Assuncao et al. [4] focus on scheduling strategies for individual jobs and the effect of those scheduling strategies on the overall cost of utilizing a cloud to provide compute resources in addition to existing site resources. Thus, depending on the scheduling strategy and job requirements, jobs are placed in a site queue or cloud queue. This differs from our approach where we schedule the actual creation and termination of VM instances in response to changes in a single queue. We allow Torque to dispatch the jobs on the resources. We do not consider the effect of individual job scheduling strategies on our system; this is an item to be considered in future work. Our policies focus solely on launching and terminating cloud VM instances in an attempt to maximize job turnaround time while minimizing the associated VM overhead.

Evangelinos et al. [6] use on demand Amazon EC2 clusters as a platform to execute climate models. The authors create an interactive application capable of spawning EC2 nodes to execute the models. This is an example of an application-specific extension to integrate support for a specific cloud. Our approach is more general, elastic site can be extended to monitor demand for any number of underlying applications as well as support a wide variety of clouds. Once elastic site supports a particular cloud provider, that provider can be made available to any application that can be monitored by elastic site.

Recently, Sun Microsystems has added support for Amazon EC2 into Sun Grid Engine (SGE) [20]. This is a scheduler specific approach to dynamic cloud integration since only the demand of a single scheduler is monitored. Though our current implementation only supports the Torque resource manager, it can easily be extended to monitor any number of underlying schedulers or applications.

In May of 2009 Amazon released Amazon CloudWatch [1], which enables EC2 users to monitor their usage on EC2. It also provides an auto-scaling feature allowing users to dynamically acquire or release EC2 instances depending on the load. However, this feature is specific to Amazon EC2, whereas our elastic site implementation can integrate extensibility sensors from multiple sources and work with multiple cloud implementations. Our solution also creates a secure context among the resources by leveraging the Nimbus Context Broker.

VI. Future Work

Our current implementation is only an initial prototype focused specifically on Torque-based clusters, Nimbus clouds, and Amazon EC2. In future work we will add support for additional resources managers and cloud providers. We will also address our assumption that the cloud nodes are isolated from each other and unable to perform collectively by leveraging network overlay solutions. Our implementation only supports the exchange of input and output data via Torque's SCP capacities, future work could involve the creation and evaluation other methods to support data exchange between local resources and cloud resources, for example, a remote shared file system could be made available to cloud nodes.

Our initial policy definitions are not comprehensive and it is likely that they will be modified to improve their efficiency for many types of job patterns. For example, one method might be to increase the responsiveness of a particular policy on clouds with quick VM boot and termination times. The evaluation of our initial policies also provide insight into other factors that should be thoroughly examined in future work, such as more effective methods to identify and integrate a cloud's estimated waste time in cases where it varies widely, e.g. EC2. Our policies do not currently consider economic factors, such as Amazon's charge-by-the-hour model where an EC2 instance running for 5 minutes costs the same as one running for 59 minutes. Policies should take this into account and leave a VM that's been charged for an hour running for the duration of the hour, even if there is no available work. Currently, the selection and configuration of policies is done manually. A more effective method would be to create a layer on top of the policies capable of evaluating incoming demand and dynamically choosing the appropriate policy on the fly. Similarly, users could provide estimations for parameters like waste time; this would allow policies to make more intelligent

decisions. Future work will also focus on the impact of job scheduling strategies to more effectively manage the placement of jobs on compute nodes.

More generally, our elastic site model will be extended beyond typical cluster resources to support other services provided within a site, such as Web services and storage archives. Recent research also [12] focuses on bridging multiple cloud providers and using the resources together as a single entity, which would allow elastic site to provide a single extension to a site leveraging resources from multiple cloud providers.

VII. Conclusions

In this work we propose a model of an elastic site capable of responding to changes in demand by leveraging cloud resources. We create an implementation of this model that seamlessly and securely extends Torque clusters with Nimbus-based clouds and Amazon EC2 on demand. We devised three policies, on demand, steady stream, and bursts to respond intelligently to changes in demand by acquiring or terminating cloud resources.

We evaluate the policies under different workloads and examine their reactiveness to different job submission patterns. We also demonstrate that we can process 10 times faster by expanding our cluster up to 150 EC2 nodes.

Acknowledgments

This work was supported by NSF CSR award #527448 and, in part, by the MCS Division subprogram of the Office of Advanced Scientific Computing Research, SciDAC Program, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. We would like to thank David LaBissoniere for his help and advice about the internals of the Nimbus toolkit and Nimbus Context Broker as well as his assistance with the configuration, deployment, and management of nodes on Amazon EC2. We would also like to thank John Valdes for setting up the initial Torque worker node on the Nimbus cloud as well as all of his help and advice configuring Torque nodes in a dynamic and secure manner. Also, thanks to Joe Rinkovsky at Indiana University for his assistance with the Nimbus cloud at IU.

References

[1] Amazon CloudWatch. Amazon, Inc. [Online]. Retreived February 7, 2010, from: http://aws.amazon.com/cloudwatch/

[2] Amazon Web Services. Amazon.com, Inc. [Online]. Retreived February 7, 2010, from: http://www.amazon.com/aws/

[3] Argonne National Laboratory Press Release. [Online]. "Nimbus and cloud computing meet STAR production demands," Retreived February 7, 2010, from: http://www.anl.gov/Media_Center/News/2009/news090402.html

[4] Assuncao, M.D., A.D. Costanzo, and R. Buyya, "Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters," High Performance Distributed Computing, 2009.

[5] Bode, B., D. Halstead, R. Kendall, Z. Lei, W. Hall, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. Usenix, 4th Annual Linux Showcase and Conference, 2000.

[6] Evangelinos, C., C. Hill. "Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2," The First Workshop on Cloud Computing and its Applications (CCA'08), October 2008.

[7] Foster, I., C. Kesselman, S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In the International Journal of High Performance Computing Applications, Vol. 15, No. 3, pages 200-222, 2001.

[8] Foster., I. Globus Toolkit Version 4: Software for Service-Oriented Systems. In IFIP International Conference on Network and Parallel Computing, pages 2–13, 2005.

[9] Gavrilovska, A., S. Kumar, K. Raj, V. Gupta, R. Nathuji, A. Niranjan, and P. Saraiya, "High-Performance Hypervisor Architectures: Virtualization in HPC Systems," In 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2007).

[10] Huang, W., J. Liu, B. Abali, and D. K. Panda, A Case for High Performance Computing with Virtual Machines. In Proceedings of the 20th Annual International Conference on Supercomputing, Queensland, Australia, 2006.

[11] Keahey, K., I. Foster, T. Freeman, and X. Zhang. Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. Scientific Programming Journal, vol 13, No. 4, 2005, Special Issue: Dynamic Grids and Worldwide Computing, pp. 265-276.

[12] Keahey, K., M. Tsugawa, A. Matsunaga, J. Fortes, "Sky Computing," Internet Computing, IEEE , vol.13, no.5, pp.43-51, Sept-Oct 2009.

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

[14] Murphy, M., B. Kagey, M. Fenn and S. Goasguen "Dynamic Provisioning of Virtual Organization Clusters" 9th IEEE International Symposium on Cluster Computing and the Grid, Shanghai, China, May 2009.

[15] Napper, J. and P. Bientinesi. Can cloud computing reach the Top 500?. In Proceedings of the Combined Workshops on Unconventional High Performance Computing Workshop Plus Memory Access Workshop, Ischia, Italy, May 18-20, 2009.

[16] Raicu, I., and I. Foster. Many-Task Computing for Grids and Supercomputers. IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), 2008.

[17] Ruth, P., J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. IEEE International Conference on Autonomic Computing, 2006.

[18] Ruth, P., P. McGachey, D. Xu. VioCluster: Virtualization for Dynamic Computational Domains, Cluster Computing, 2005. IEEE International, pages 1-10, Sept. 2005.

[19] Science Clouds. [Online]. Retreived February 7, 2010, from: http://www. scienceclouds.org/

[20] Sun Grid Engine. Sun Microsystems. [Online]. Retreived February 7, 2010, from: http://www.sun.com/software/sge/

[21] TeraGrid. [Online]. Retreived February 7, 2010, from: http://www.teragrid.org/

[22] Walker, E. Benchmarking Amazon EC2 for high-performance scientific computing, Retreived February 7, 2010, from: http://www.usenix.org/publications/login/2008-10/openpdfs/walker.