A Flexible Elastic Control Plane for Private Clouds

Upendra Sharma
IBM Watson
usharma@us.ibm.com

Prashant Shenoy
Dept. of Computer Science
Amherst MA 01003
shenoy@cs.umass.edu

Sambit Sahu
IBM Watson
sambits@us.ibm.com

ABSTRACT

While public cloud computing platforms have become popular in recent years, private clouds—operated by enterprises for their internal use—have also begun gaining traction. The configuration and continuous tuning of a private cloud to meet user demands is a complex task. While private cloud management frameworks provide a number of flexible configuration options for this purpose, they leave it to the administrator to determine how to best configure and tune the cloud platform for local needs. In this paper, we argue for an adaptive control plane for private clouds that simplifies the tasks of configuring and operating a private cloud such that each control plane service is adaptive to the workload seen due to end-user requests. We present a logistic regression model to automate the provisioning and dynamic reconfiguration of control plane services in a private cloud. We implement our approach for two control plane services—monitoring and messaging—for OpenStack-based private clouds. Our experimental results on a laboratory private cloud testbed and using public cloud workloads demonstrates the ability of our approach to provision and adapt such services from very small to very large private cloud configurations.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance

Keywords

Cloud computing, dynamic provisioning, logistic regression

1. INTRODUCTION

Cloud computing has become popular in recent years for running Internet and enterprise applications due to its pay-as-you-go pricing model and ability to elastically allocate resources. While public cloud platforms have attracted much attention, the design of private clouds—cloud platforms that are operated by enterprises for their own internal use—have begin gaining traction. Today a number of private cloud management frameworks are available, ranging from commercial offerings from IBM and VMWare to open-source frameworks such as OpenStack, CloudStack and OpenNebula. Despite the availability of these platforms, the task of configuring, managing and operating a private cloud remains challenging. Most private cloud management frameworks expose a range of flexible configuration options and settings to support various deployment architectures. However, they leave it to the system administrator to determine a deployment architecture and configuration settings that are best suited for local needs. In particular, most private cloud management frameworks implement a *control plane* for managing various cloud services such as monitoring, messaging, allocation of compute and storage resources and VM image management (see Fig. 1). The task of configuring each service, allocating sufficient resources to service end-user requests, and continuously tuning the service to adjust to changing needs is left to the administrator.

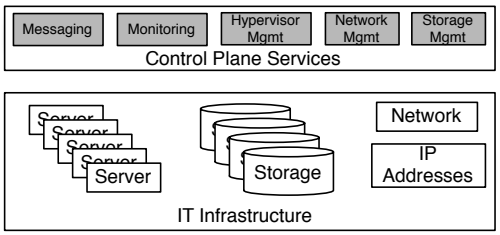


Figure 1: Architecture of private cloud and its control plane.

In this paper, we argue for an adaptive control plane for private clouds that simplifies the tasks of configuring and operating a private cloud. Such an adaptive control plane must simplify the initial setup and configuration of each control plane service and ensure that each service is responsive to the workload seen due to end-user requests. Further, as the demands imposed by the private cloud vary over time, the control plane must adapt the service to changing workloads.

Cloud platforms have long supported the notion of *elasticity* for end-user applications. Elasticity implies that the resources (such as the number of VMs or servers) allocated to the application is automatically adjusted to match the variations in the incoming workload. In this work, we propose that the *control plane of the cloud must itself be elastic* and adjust the resources allocated to various control plane services automatically to changing needs—just as it does for end-user applications.

Thus our paper focuses on the design of a flexible, adaptive control plane that automates the initial configuration of each control plane service to match the needs of a private cloud of a desired size and elastically provisions resources for these services as their

CAC'13, August 5–9, 2013, Miami, FL, USA.

workload demands change over time. In designing our elastic control plane, we make the following contributions.

First, we model the interactions of each control plane service with end-user VMs and between themselves and develop a logistic regression based model to estimate capacity needed to sustain a certain workload with a certain SLO. A key benefit of using logistic regression over other techniques is that it does not require a large training set to model the behavior of the service. Our adaptive control plane then uses this model to determine how many nodes (or VMs) are needed to service the expected workload. In the event the control plane service needs to be replicated, it also determines whether these replicas should be federated or clustered to meet the desired SLO in the most efficient manner. Such approach greatly simplifies the initial setup of each control plane service by the administrator.

Second, since the workload seen by a control plane service may vary or grow over time, our control plane implements elasticity of each service. We present reactive and proactive elasticity mechanisms that can dynamically provision additional capacity for each service on-the-fly. Our proactive approach combines our logistic regression model with workload forecasting techniques to proactively allocate resources to each elastic control plane service.

Prototype implementation and experimental validation: Third, we implement a prototype of our flexible elastic control plane for an OpenStack-based private cloud and demonstrate its efficacy for two essential control plane services: monitoring and messaging. Our experimental results on a laboratory private cloud testbed and using public cloud workloads demonstrates the ability of our approach to provision and adapt these services for private clouds ranging from very small to very large configurations. We also demonstrate the ability of our dynamic reconfiguration approach to elastically provision capacity to these services on-the-fly.

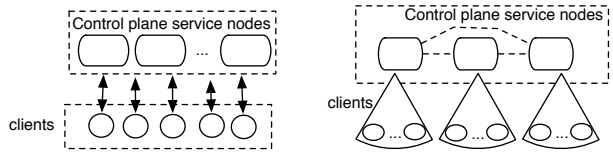
2. BACKGROUND

Private Clouds: A private cloud consists of infrastructure resources like compute, storage and network and allows its users to create virtual resources on-demand. Private clouds implement similar functionality as public clouds like Amazon EC2, except that use they infrastructure owned by an enterprise to implement cloud functionality for internal use. A number of open source cloud management platforms are available to establish and operate a private cloud, namely OpenStack [11], CloudStack, Eucalyptus, OpenNebula etc. These assume a cluster of linux machines and provide a control plane to manage the cloud infrastructure and perform management tasks, like hypervisor management, user management, messaging, monitoring, image management, etc. as depicted in Figure 1. Each such management task is performed by a control plane service that runs in one or more virtual machines. In this work we have chosen OpenStack as our cloud management system of study; this is primarily because it offers a rich set of control plane services and has become a popular choice amongst the open source community [8].

Problem Formulation: Consider an organization that wishes to deploy a private cloud on a cluster of size N. Most private cloud management frameworks are designed to work with as few as tens of hosts/machines to very large clusters consisting of thousand machines, but for successful and efficient operation, the cloud management system has to be configured according to the size of the cluster. To do so, the administrator must configure each control plane service and provision sufficient capacity so that it can service the control plane workload generated by the management tasks in a cluster of that size.

In the simplest case, each control plane service will run on a single virtualized node. A single node per service setup is adequate for a small to medium size clusters. However, as the cluster size grows, a single node setup will become a bottleneck. For instance, consider a monitoring service, which performs two major tasks, recording the monitored metrics for all resource as well serving queries regarding the same. A single node deployment of the monitoring service may easily handle the monitoring data from a cluster containing a few tens to a few hundred nodes. However, if the cluster grows to a thousand machines or more, the amount of monitoring data that is generated by the clients will overwhelm the single node monitoring service.

To scale the control plane in such scenarios, the service will need to be replicated on multiple nodes and the incoming workload to the service will need to be distributed across the replicas of the service. Typically replication can be done in one of two ways: (i) by employing clustering, where a group of replicas of the control plane service *collectively* serve the requests made to it, or (ii) by employing federation, which *partitions* the workload across multiple instances of the control plane service. In the clustering approach, all replicas collectively serve all the requests as a single logical entity – as shown in Figure 2a. In federation, each service instance services a subset of clients and forwards only the necessary requests to the other – as shown in Figure 2b. Both clustering and federation approaches partition the workload but clustering based approach also allows high availability while federation does not.



(a) Clustering (b) Federation

Figure 2: Clustering and Federated approaches

Given such a private cloud management system and the control plane services, an IT administrator is faced with a two fold task of appropriately configuring each control plane service and ensuring that there is sufficient capacity to service the requests. Manual configuration and capacity allocation is a challenging task as a large number of interdependent services are involved. We, thus, have the problem of configuring and provisioning each service so that the task of deploying the control plane service can be automated.

While there are rules of thumb on how to configure these control plane services, it is not apparent which approach to use to scale up and in what situations. In addition, it is challenging to determine how many instances to provision for a private cloud of a certain size. Our approach is to design a flexible control plane service that automates this task by solving two sub problems: (i) Given the size of cluster, say N, choose which approach is suitable, i.e. single node, clustering, or federation for each service. (ii) Determine and provision sufficient number of nodes if the service is replicated.

Dynamic Provisioning: The initial setup of the control plane and its various services is based on an estimate of the workload likely to be seen by each control plane service. However, the workload observed by control plane services may change over time either due to imperfect initial estimates of client workloads or due to incremental growth of the managed infrastructure or even a sudden change in managed workload. For instance the administrator may increase the monitoring resolution from 15 min to 1 min, causing

an order of magnitude increase in the monitoring data. In such situations, some services required to be reconfigured by dynamically increasing (or decreasing) the capacity of the control plane service. Thus the *control plane must itself be adaptive and elastic*—it needs the ability to dynamically reconfigure a control plane service by provisioning new capacity for the service when the specified SLOs can no longer be met. While the problem of dynamic provisioning of application VMs has been well studied [15, 13, 19, 21, 14, 20, 17], elasticity and provisioning of control plane services of a private cloud has not received much attention. As we argue in this work, prior methods such as queuing models for provisioning of application VMs are not suitable in this context, primarily because models often can not account for software artifacts that limit the application capacity from scaling. Secondly, models are often specific to a software with a specific topology type and are very expensive to develop. Instead we exploit the particular nature of control plane service interactions to model a control service and design elasticity mechanisms that are tailored for such scenarios.

System Model: Each control plane service is assumed to be composed of multiple software components; these components can be deployed in dedicated virtual machines – we refer to them as *component nodes* of a control plane service. In this work we assume that all the component nodes of a control plane service are identical (thus we also address a component node as a replica). This is not a limitation of our approach but a simplification, which we have adopted for ease of exposition of our approach. A fully functional control plane service is assumed to be created by arranging these component nodes in a single node, clustered or federated configuration – as shown in Figure 2. We assume that each component node has an associated SLO it and the administrator must pick a configuration and number of nodes such that there is enough capacity to serve the request seen by the service. Further, it is assumed that the SLO violations of each service can be monitored by logging the performance seen by control plane requests.

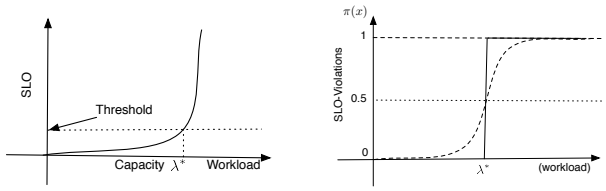
3. MODELING AND CONFIGURING CONTROL PLANE SERVICES

Since each control plane service can be clustered, federated or run on a single node, we model service as a set of one or more identical components (referred as component nodes). A component node is assumed to service two types of requests, namely *external* requests from infrastructure nodes or other services and *internal* requests from the other component nodes of the same service. Let λ_c and λ_n denote the average workload due to external client requests and internal nodes requests, respectively. We also assume that each control plane service needs to meet a performance threshold to meet an Service Level Objective (SLO). SLO of a control plane can be specified using a threshold on application performance metric (e.g. latency) or on a resource utilization metric, for instance 80% of CPU utilization. Administrators must estimate and provision sufficient resource capacity to a control plane service to avoid violating the SLO. We automate this task of configuring and provisioning the control plane service by determining whether a single node or clustered or federated configuration is best suited for the control plane service and how many nodes are necessary to provide the desired capacity.

Our approach comprises of deriving an analytical model to determine the capacity needed and an algorithm to dynamically reprovision when the workload increases beyond the capacity. We gather empirical data of system performance by offline empirical profiling; it aids in accounting for i) software artifacts which limit the applications capacity and ii) performance variation due to various hidden factors, like shared resource allocation, hardware etc.

3.1 Analytical model

Control plane service uses different resources, namely memory, CPU, network etc. The performance of a control plane service can be affected by many factors, including its configuration, workload variations, resource utilization, and also artifacts of the involved software components as well as those of the system. We present a probabilistic model, based on logistic regression, to estimate the capacity needed by a control plane service to service a particular workload.



(a) (b)

Figure 3: Intuition of SLO violation curve

Let λ_T be the be the total estimated workload and let k be the number of replicas ($k \geq 1$) needed to service this workload. Thus, we must estimate the number of replicas k required by a control plane to service a workload of requests arriving at rate λ_T for a given SLO. Our approach consists of gathering empirical data of SLO violations of each node/replica of the control plane service and use these observations to build a probabilistic model/function of SLO violations given the observed workload at the node. We then use this model/function to determine the max load λ_c^* that can be serviced by a single node; given this capacity of a single node, we can estimate the number of nodes, i.e. k, for a specific configuration (i.e. clustered, federated).

We now determine a function that relates λ to the SLO. More formally, let Y be a binary random variable, which represents presence/absence of an SLO violation and λ be the total workload observed by a node (i.e. $\lambda = \lambda_c + \lambda_n$). We, then, wish to estimate the conditional expectation of SLO violation, i.e. $E(Y|\lambda)$. There are a number of sophisticated non parametric techniques which can estimate conditional probabilities but these techniques often require a large amount of training data to create reliable models.

Logistic regression [7] is an alternative that does not require a large number of training samples to determine the conditional expectation. Let $\pi(\lambda)$ denote the conditional expectation $E(Y|\lambda)$, when assuming a logistic distribution. The specific form of logistic distribution we use is:

$$\pi(\lambda) = \frac{e^{(\beta_0+\beta_1\lambda)}}{1+e^{(\beta_0+\beta_1\lambda)}}, \tag{1}$$

where, β_0 is the intercept parameter and β_1s is the slope parameter. We re-write (1) to obtain a linear equation in λ:

$$g(\lambda) = \ln\left(\frac{\pi(\lambda)}{1-\pi(\lambda)}\right) = \beta_0 + \beta_1\lambda. \tag{2}$$

The parameters β_0 and β_1 can be estimated using logistic regression; they are maximum likelihood estimates of $\pi(\lambda)$ – expectation of SLO violation for a given λ.

Using (2), we can compute the value of λ for a given probability of SLO violation, say $=\lambda^*$. For instance, let us suppose we want to

compute the capacity λ^* for a conservative threshold on probability of SLO violation, say 0.5; this essentially means that whenever $\lambda \geq \lambda^*$ there is more than 50% chance of SLO violation (as shown in Figure 3b). Thus equating $\pi(\lambda) = 0.5$ in equation 2 yields

$$\beta_0 + \beta_1 \lambda^* = 0 \quad (3)$$

Estimating β_0 and β_1 requires some real observations of workloads and SLO of a control plane node in a real setting. For that we perform offline empirical profiling of control plane services in different topologies as outlined in the next section.

3.2 Workload Estimation

The workload λ seen by a node of each control plane service has two components, namely requests from the clients (λ_c) and requests from the other replicas/nodes of the same service (λ_n), i.e. $\lambda = \lambda_c + \lambda_n$. Intra service workload (λ_n) is a function of client workload (i.e. $\lambda_n = f(\lambda_c)$) and the exact form of the function depends on the configuration.

We can use knowledge of the control plane to provide the function f. For instance, in a federated setup the clients are partitioned into smaller groups and each partition is serviced by one node/replica. Thus λ_n is a fraction of λ_c, i.e. $\lambda_n = \delta\lambda_c$. Similarly, in the case of clustering, the intra service workload depends on the size of the cluster and also on the way it has been implemented. This means that if a clustered configuration implements information exchange via broadcast then the messages received by each node will equal the size of the cluster; now, if the service uses multicast transmission to implement the same then only one message need to be sent, however,if the implementation adopts unicast transmission then the number of outgoing messages will be equal to the size of the cluster. Thus for a cluster of size n we will have $\lambda_n = 2(n-1)\lambda_c$ if the unicast is adopted, while in the case of multicast based implementation it will be $\lambda_n = n\lambda_c$.

On the other hand if nothing is known about the control plane service then we can treat the control plane service as a black box and estimate λ_n as function of λ_c by regressing over the empirical profiling data, i.e.

$$\lambda_n = \alpha_0 + \alpha_1 \lambda_c. \quad (4)$$

For the purpose of computing initial estimate of control plane's capacity, and also for performing empirical profiling, we require an estimate the client workload, i.e. λ_c, and the workload generated by a single client, say λ'_c. We make use of rules of thumb or prior experience for the same; for instance, if it is known that for each monitored machine a monitoring service records an average of 25 metrics at a granularity of 1-sec, then $\lambda'_c = 25$. Now, if the monitoring node services n clients then the average client workload observed by a single monitoring node will be $\lambda_c = n \times 25$ and the total client workload observed by the whole monitoring service for a cluster of size N will be $\lambda_T = N \times \lambda'_c$.

3.3 Provisioning Algorithm

Having modeled the control plane service and estimated the workload parameters, we compute the number of nodes required for service as follows:

Step 1: First we use the training data to compute the βs in (2) using logistic regression. Using the values of βs we compute a conservative capacity of a control plane node in terms of workload which it can handle, i.e. λ^*, using (3).

Step 2: Next we estimate the maximum client workload a control plane node can service, say λ_c^*. Since observed internal workload (λ_n) is a known function, $f()$, of client workload (λ_c) we estimate the capacity of the service in terms of number of clients that can be serviced, say λ_c^*, by solving the following equation for λ_c^*:

$$\lambda_c^* + f(\lambda_c^*) = \lambda^*$$

Step 3: We estimate the capacity of a control plane service, i.e. total number of control plane nodes (say k), required to service a cluster of size N, i.e. $k = \lceil \lambda_T / \lambda_c^* \rceil$.

Step 4: If the above steps indicate that a single node is not sufficient to handle the workload, i.e., the k is found to be greater than 1, then we must determine whether to employ clustering or federation for the replicated service. To judiciously choose between them, the above steps are repeated for clustering and federation by using the appropriate function $f()$ for each configuration as derived in Section 3.2. We then choose the configuration that is more efficient, i.e., yields a smaller k. The final step then provisions the estimated capacity k for that configuration, i.e. clustered or federated.

4. ELASTIC RECONFIGURATION

Our provisioning algorithm provides a technique to determine the appropriate configuration (e.g., single node, clustered or federated) and the capacity k needed to service the estimated workload. Since the initial provisioning is based on an estimate of the workload likely to be seen, the actual workload may be different or may grow over time. Hence our control plane implements elasticity for each service by enabling them to be re-provisioned as and when needed. For example, if the administrator changes the frequency of monitoring each node from 5 minutes to 1 minute, there will be a five-fold increase in monitoring data, which may require the monitoring service to be reprovisioned if any node gets saturated due to this change. Such elastic reprovisioning and reconfiguration involves two steps: i) *When* to trigger dynamic reprovisioning? ii) *How* to migrate from current configuration to new one?

When to trigger? Elastic reprovisioning can be triggered reactively or proactively. Reactive reprovisioning is triggered when the control plane detects SLO violations for a particular service, while proactive reprovisioning is triggered when future workload forecasts indicate SLO violations are likely in the near future.

Reactive: The control plane monitors each service and reacts to observed SLO violations by invoking re-provisioning. In this simplest case, the control plane can gradually increase the number of replicas allocated to a service in steps until the SLO violations stop (e.g., increase the number of replicas by one node at a time step until the violations stop). A better approach is to use the recent history of the workload seen by the service re-run the provisioning algorithm from the previous section. Doing so will yield a new k for the number of replicas needed by the service and the control plane can simply start $k - k'$ new replicas, where k' is the current number replicas for the service.

Proactive: Proactive provisioning involves combining workload forecasting with the provisioning algorithm from the previous section to anticipate SLO violations before they occur and take corrective action. To do so, we can employ a workload forecasting technique to predict the expected workload Δt time units into the future. If the predicted workload is higher than the peak estimate used for the currently provisioned capacity, then SLO violations are likely and the control plane will invoke the provisioning algorithm from the previous section with the new workload forecast. While any workload forecasting method can be used by the control plane, we currently employ time-series forecasting. Similar to the approach used in [15], we obtain a time series of workload observations, model the workload as an ARIMA time series [4], and use standard ARIMA-based forecasting to predict the workload for a fixed time interval Δt into the future. This prediction is used by

the provisioning algorithm to compute a new capacity k and additional replicas are spawned by the control plane for the service.

How to migrate to new configuration? There are two main steps in migrating the control plane service from old configuration to new configuration, namely i) redeployment and ii) redistribution of the clients across the new configuration. *Redeployment* involves deploying the necessary additional VM replicas for the service. Application topologies can be encoded in an Open Virtualization Format (OVF) [6], which can be used by external deployment scripts to provision the replicas. Most common cloud management platforms support OVF making it a good implementation choice. In this work our re-deployment task provisions newly computed capacity and and inter-connects the deployed components forming the same configuration pattern; however, selecting and switching to a different configuration pattern is a relatively easy extension of this work.

Redistribution: Once new replicas have been provisioned, the workload has to be redistributed across new and old replicas equally. This involves identifying the clients of the service and changing their configurations to append new replicas to the list of available replicas for the service, and perhaps specifying the preferred replica to use for the service.

5. PROTOTYPE IMPLEMENTATION

This section describes the prototype of our elastic control plane.

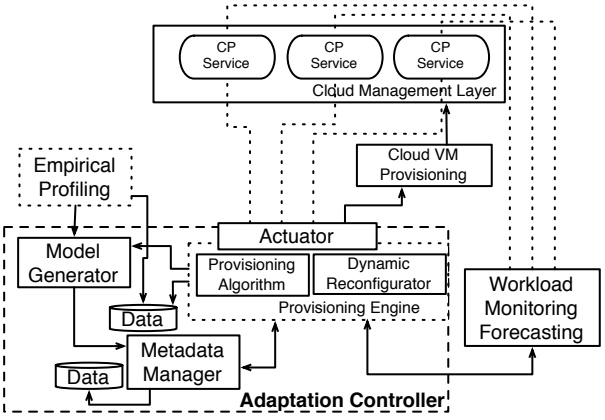


Figure 4: Architecture of our elastic control plane

5.1 System Architecture

Our prototype depends on monitoring of the system and performance metric of each of the control plane service nodes. Monitoring of systems and resources is a standard practice followed in all large system deployments and besides this our approach does not put any additional load on the control plane service. Other components of our prototype, namely *adaptation controller* and *monitoring and foreasting* components, are hosted on a dedicated VM and implemented in python; details of each these components are (see Figure 4):

Model Generator takes the empirical profiling data for each control plane service and generates a model (set of βs) and stores it in the *Metadata manager*. We have used the STATA 10's implementation of logistic regression [2] to obtain our models.

Metadata manager stores models for each control plane service, their current configuration and capacities. We have implemented it as a python class, which stores all the information in in-memory data structures with an option to persist the models on disk.

Workload Monitoring and Forecasting Engine collects time-series monitoring data of all the virtual machines as well as of those of the control plane services. It stores all the results in a database, which can be queried. We have implemented this as a part of monitoring service of OpenStack using Ganglia. We have used STATA 10 for implementing the ARIMA forecaster [18].

Configuration and Provisioning Engine implements the provisioning algorithm. It takes the generated model from *Metadata manager* and computes the number of replicas needed for a configuration. In case of change in configuration, it provisions new replicas using the *Actuator* module and updates the details of new configuration to Metadata manager. It also performs *dynamic reconfiguration* by constantly evaluating the SLO metric and by computing the change in average client workload. As a solution to the less frequent situation where the model requires re-learning, the *provisioning engine* queries and collects the cases of SLO violations and updates the learning data. It then re-estimates the model parameters and updates the records in *metadata manager*.

Dynamic reconfigurator: This component exposes two interfaces, *redeploy* and *redistribute*. We provide an implementation for each control plane service. Currently we have implemented a plugins for monitoring and messaging services.

Actuator: This is module is a part of configuration and provisioning engine. It performs the task of deploying new virtual machines of each control plane service. After deployment it executes the necessary scripts in each replica of the control plane service for creating the correct configuration. The actuator also looks up the dependent clients and alter's their configuration so that the client workload is evenly distributed across all the replicas. It essentially keeps a fixed number of clients for each replica.

5.2 Private cloud management system

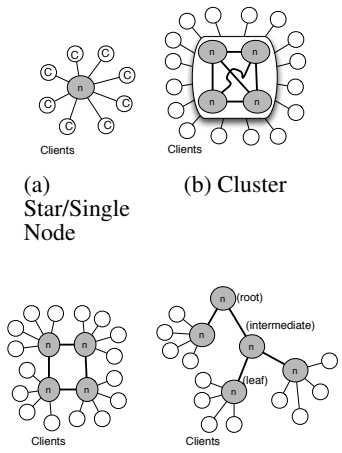
We use OpenStack as our private cloud management system. OpenStack comprises four main components: compute (Nova), image repository (Glance), authentication (Keystone), and storage (Swift). Nova, Glance, and Keystone provide hypervisor management, image management and authentication, respectively, while Swift provides an object-store service [10]. We use *Nova* as an example to expose some of the design details of scalable cloud service components. Nova has multiple control plane services which together provide the the functionality of compute and storage management. Nova's various services communicate with each other via message queuing [16]. Here we use an open-source message queuing system, namely RabbitMQ [12].

Although monitoring is a key component of a cloud platform, OpenStack currently lack a full fledged monitoring service. We implement our own prototype monitoring service for OpenStack to mimic Amazon's CloudWatch monitoring service in the EC2 public cloud. We build monitoring for OpenStack by integrating two open-source monitoring systems into OpenStack: Ganglia and Nagios [9]. Ganglia is used to monitor nodes and VMs and archive monitored data in its database while Nagios is used to set simple triggers on monitored data; for instance, we can use Nagios to report if the average resource utilization of a group of nodes exceeds a threshold.

5.3 Empirical profiling

As the first step towards learning the parameters we perform empirical profiling of each control plane service in both clustered and federated configurations. In order to empirically profile a component node of a service in a particular configuration, we start with

a single node configuration and systematically increase the client workload on the service (i.e. λ_c) until we observe SLO violations. We, then, repeat the same procedure with the next bigger graph of the same configuration and so on; at each step we record the average intra service workload as well (i.e. λ_n)



(a) Star/Single Node (b) Cluster

(c) Federated - Ring (d) Federated - Heirarchical

Figure 5: Profiling configurations; grey nodes represent control plane service nodes and the white nodes are its clients

Single node: this is the smallest configuration – Figure 5a. We assume the average workload generated by a single client, i.e. λ_c and for the purpose of increasing λ_c we simply increase the number of clients until we observe SLO violations. *Cluster configuration*: We start with a cluster of size 2 and distribute the clients between the two nodes. We, then gradually increase the workload on both the nodes, distributed evenly, until we observe SLO violations on any of the nodes. We measure both the average number of client requests per sec, λ_c, and the average number of intra service requests per sec, λ_n. We repeat the same experiment for larger cluster sizes. This helps us gather data necessary for capturing the impact of cluster size as well.

Federated configuration: A federated configuration partitions its clients between component nodes and can be hierarchical or non (e.g., a ring). For nonhierarchical kind of federated configuration, the method of profiling is like that of clustered configuration, i.e. we start with smallest possible configuration and profile till it is saturation and then increase the number of component nodes by one and repeat the same procedure. In the case of hierarchical configuration, we have to differentiate between nodes, i.e. leaf nodes, intermediate nodes and root node. This makes the profiling a little more involved. We start with a tree of depth=1, i.e. with a root node and single leaf node; Then, similar to cluster topology we increase the client workload on the leaf node while keeping the client workload of root node to zero until leaf node's saturation. We keep the root node's client workload to zero and gradually increase the λ_n by adding more leaf nodes. We repeat the same experiment with a tree of depth two, where there is a leaf node and intermediate node and then a leaf node. We empirically profile the intermediate node.

Determining SLO metric: SLO metric of each control plane service should selected such that a faithful operation of the management service can be ensured by monitoring an easily observable value threshold of the metric. In situations where the SLO metric is not directly observable, administrators can empirical profile the service in a closely controlled environment and choose that metric (or a set of metrics) which are strongly correlated with SLO. Empirical profiling also assists administrators to heuristically determine a conservative threshold value for the monitored metrics, which can be used to trigger dynamic provisioning much before the actual SLO violation happens.

We manually perform empirical profiling by gradually increasing the workload and record SLO violations. This data is used to obtain the model of the management service. Offline empirical profiling is not a limitation of the approach as administrators often perform empirical profiling before deploying a large-scale system; in addition to this, the empirical profiling data can be used as a starting point of the dynamic provisioning algorithm.

6. EXPERIMENTAL EVALUATION

This section describes our experimental setup and our experiments to test the efficacy of our elastic control plane.

6.1 Experimental Setup

We have used OpenStack as our private cloud management system and have experimented with two of its control plane services, namely monitoring and messaging.

Monitoring: As explained earlier, we have implemented the monitoring control plane service of OpenStack using Ganglia. Ganglia consists a monitoring agent, *gmond*, which gathers and broadcasts monitored data using UDP multicast/unicast. The monitored data is pushed to a metadata server, *gmetad*, for archival; *gmetad* stores data in a Round Robin Database (RRD) and leverages *rrdtool* to extract and graph the monitored data using Apache web-server and php technology. For our experiments, we defined SLO of our monitoring subsystem as a threshold on the percentage of data loss due to unreliable message delivery or system saturation. We have used Ganglia 3.3.6 on Ubuntu to create each node of our *monitoring* control plane service by deploying both a *gmetad* and a *gmond* daemon on it. This server is responsible for gathering all data from the monitored nodes. Client workload generated by a single client (i.e. λ_c') is dependent on number of metrics being monitored and the frequency at which they are monitored. We have considered three types of monitoring workloads for our experiments. Each of these workloads involve monitoring 25 metrics but at different monitoring frequencies, i.e. 1-sec, 5-sec and, 15-sec.

To simulate large clusters than available in our testbed, we created client workload generators that emulate the data sent by ganglia on real nodes. Our clients generate and send synthetic gmond 2.x data packets to gmond. For each monitored node we sent 25 separate metrics at the pre-configured monitoring frequency. On the *monitoring node* the metrics are saved in separate files and folders, where files are named using the metric's name and the folder is named using the host name.

Messaging: OpenStack's message queuing subsystem is the backbone of this scalable private cloud management system. All control plane services of the compute cloud of OpenStack (i.e. Nova) communicate with each other via blocking and non-blocking RPC calls using the message queuing system [16]. We use RabbitMQ 2.8 as OpenStack's message queueing system for our experiments. We experimented on a private cloud created over 12 Intel Xeon (X3430) machines each with 8 GB RAM and 500 GB SATA Disk. The machines were installed with Ubuntu 12.04 and we created the private cloud using OpenStack Essex release. Each RabbitMQ node possesses both queue-management capabilities as well as router capabilities. We installed each such node on single core VMs with 6GB of RAM. The clients were deployed on the hosts described above. On each VM as well on each host we set the limit to num-

ber of open files to 81920 (80 K). In order to synchronize the clock we used NTP 4.2.6.

Since RabbitMQ is memory bound and stops receiving messages when memory gets saturated, for our experiments, we define a SLO as a high threshold of the memory utilization of the RabbitMQ node (E.g., 75% utilization threshold). To generate the workload, we assume that each client, i.e. Nova-compute and Nova-volume generate one VM and volume creation and deletion request every hour. We assume that each client node is of 64 cores and thus each node create one VM as well as volume creation request every second. Since a VM creation requires 5 messages and VM-deletion requires 6 messages and equal for volume creation and deletion, thus $\lambda_c' = 22$.

6.2 Empirical Profiling and Capacity Estimation

In this section we empirically profile two control plane services, namely monitoring and messaging, in different configurations, namely single node, cluster and federated. We then use the profiling data and the results developed in Section 3 to compute the maximum capacity of a configuration. Finally, we test our dynamic reconfiguration approach on monitoring subsystem deployed in a federated topology.

6.2.1 Single Node Configuration

Monitoring: We created a single node configuration of monitoring control plane service by using the *m2.xlarge* instance of EC2 as our *monitoring node*. We conducted profiling in three different monitoring granularities, i.e. 1-sec, 5-sec and 15-sec.

We have simulated n_c client nodes by sending the monitored data of $n_c \times 25$ metrics[1] to the *monitoring node* from 5 client machines running the client workload simulator. The λ_c observed by the monitoring node in the three respective monitoring granularities is $n_c \times 25$, $5 \times n_c$, and $5 \times n_c/3$. The observed data-loss at the *monitoring node* for each of the three different monitoring granularities is recorded for training the model. The SLO plots for each of the experiment are shown in Figure 6, where each point on the graph is an average of more than 50 samples.

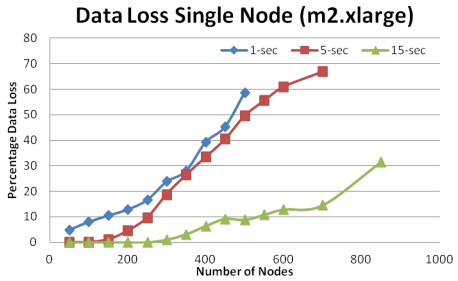
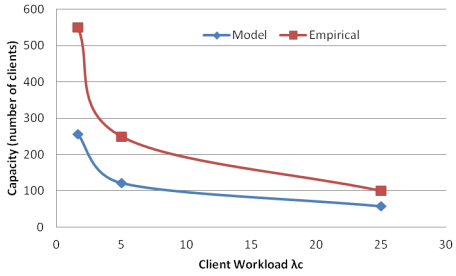


Figure 6: Data loss in a single node configuration

We used the complete profiling data to compute the capacity model of a single node configuration in three different settings. We estimate the parameters of the model using logistic regression and compute capacity using (3). Figure 7b summarizes the results of profiling of a single node configuration with the three different monitoring workloads as a table.

It can be seen from Figure 7a that capacity does not vary linearly in λ_c and that the model provides a conservative estimate of capacity with the data generated by empirical profiling.

[1]Amazon cloudwatch monitors 25 metrics for an instance and its volume



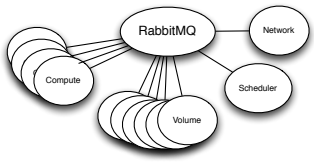
(a) Estimated and observed capacities as a function of λ_c

	1-sec ($\lambda_c = 25$)	5-sec ($\lambda_c = 5$)	15-sec ($\lambda_c = 5/3$)
Capacity	58	122	256

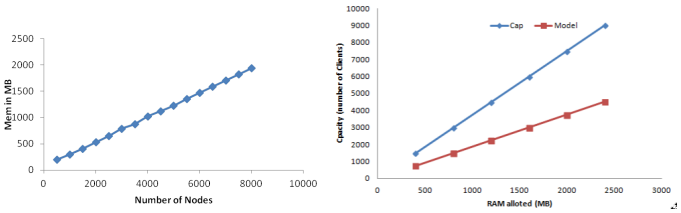
(b) Estimated Capacities

Figure 7: Empirical and estimated capacities of single node monitoring configuration with monitoring node on an *m2.xlarge* instance type.

Messaging: We conducted the experiment with a single node configuration by using a single core VM but with varying amount RAM to RabbitMQ, i.e. starting from from 400MB to 2.4GB. For each RAM configuration, we gradually increase the number of compute and volume nodes, which increases the message traffic via the message queue. We, then, measure memory utilization of the RabbitMQ node. The results are shown in Figure 8.



(a) Single node setup



(b) Memory utilization (c) Capacity

Figure 8: Memory utilization and average message latency observed in a single node configuration of RabbitMQ

In each experiment we gradually increased the number of compute nodes, keeping a single scheduler and a single network node. Increasing scheduler and network is not a recommended configuration in openStack. In each experiment we scaled up the number of clients in batches of 250 clients. We stop adding clients when the memory utilization reaches 75% of the total allowed memory to RabbitMQ server. We found that the memory utilization linearly increases with number of clients, as shown in figure 8b. To generate a model of single node configuration, we conducted experiments

where we varied the RAM from 400MB to 2400MB and the results are shown in figure 8c. It can be observed that the capacity scales linearly with RAM for workload generated by OpenStack clients and the model captures a conservative estimate of the same.

Conclusion: Empirical profiling effectively captures the software artifacts. In addition the model allows us to capture that knowledge and generate conservative estimates.

6.2.2 Federated configuration

For a federated configuration we conducted an experiment with a tree of depth two, as shown in figure 9a (which means a tree of depth one for control plane nodes). In this configuration there are multiple monitoring nodes each of which gathers the data from their individual group of monitored nodes, called clusters. It is often useful for administrators to have a summary of monitored metrics at cluster level. In our case the root node pulls the summary statistics data every t_r-seconds. This places additional load on the leaf monitoring nodes and thus impacts data loss.

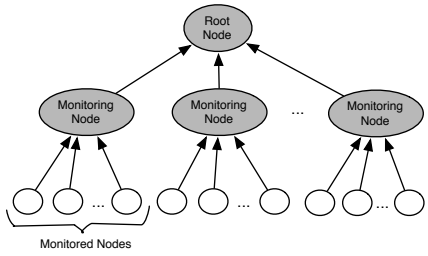
In a hierarchical configuration there are three types of nodes, namely a leaf metadata node (subjected to both client and intra-service workload) and a root node (with only intra service workload); we profiled each of these nodes. For leaf metadata nodes, we generated the client workload in the same manner as for single-node configuration profiling. However, for generating intra service workload (λ_n), we setup the root node to pull data from the leaf metadata nodes at three different granularities, i.e. 15-sec, 30-sec and, 1-min. This is because the higher level nodes in the tree collect only summary statistics of the lower level nodes and thus the resolution is often quite low. For each resolution, we measure SLO while gradually increasing λ_n. The variation in the SLO metric with increase in workload, for both leaf as well as root metadata node, is shown in Figure 9b and 9c respectively. We have used an average workload of 25 metrics per monitored client, thus $\lambda_c = 25/t_l$, where t_l is the monitoring granularity of the leaf metadata node. Similarly average workload generated by a leaf metadata node for its parent node is $\lambda_n = 150/t_r$, where t_r is the monitoring granularity of the root metadata node. We conducted nine profiling experiments and developed capacity models for each of them. As expected, we find that the data loss characteristics of the leaf monitoring nodes are very similar to those of a single node configuration except only slightly less (shown in Table 1a). However, as the root metadata node's monitoring granularity increases to 30-sec and 60-sec the impact becomes nearly negligible.

Table 1c summarizes the maximum capacities of a tree topology of depth one with nine different settings of monitoring granularities. The total capacity of each of the nine configurations is computed by multiplying capacities of leaf and root nodes. This is because of the fact that we assume $\lambda_n = 150/t_r$ (a constant). Note that an approximate functional form of $\lambda_n = f(\lambda_c)$ is estimated using the knowledge of the monitoring service. Since the root node collects only the averaged values from each child metadata nodes, it computes to $\lambda_n = \lambda'_c \times 6/t_r$.

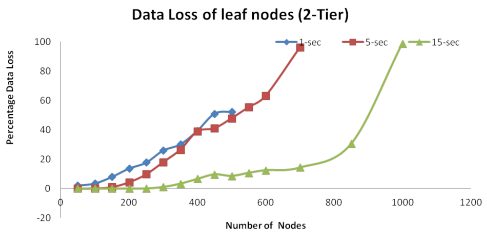
Conclusion: Empirical profiling assists in capturing the application artifacts. This coupled with our modeling approach helps in estimating maximum capacity of any configuration.

6.2.3 Cluster Configuration

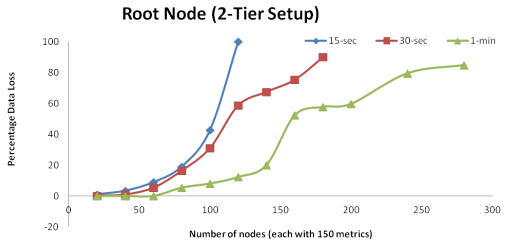
RabbitMQ supports a clustering configuration, where each broker node in the cluster has all a replica of all the data necessary for operation. This means that any queue can be accessed from any broker node, however, the queues and its messages are not replicated and thus it saves unnecessary excess communication. We



(a) A federated configuration



(b) Data loss on leaf monitoring node; root node monitoring at 15-sec granularity



(c) Data loss on root monitoring node

Figure 9: Data loss in a federated configuration

experimented with three types of cluster node configurations and the results are shown in Figure 10.

We conducted two set of experiments: First with by hosting RabbitMQ on a single core VM but with 2.4GB RAM, second with a RabbitMQ server with single core VM and with 0.4GB RAM. For both the experiments we varied the size of cluster, say k, and for each such cluster we gradually increased the number of clients till SLO violations were observed. The second experiment was conducted with a limited RAM to study the asymptotic behavior of the configuration and also to test if the model can capture this knowledge faithfully.

In the case of clustering configuration, the intra service workload (i.e. λ_n) scales linearly with λ_c. The linear function is such that it also depends on the size of cluster, say k. So we estimated the following function from our empirical profiling data $\lambda_n = \alpha_0 + k\alpha_1\lambda_c$.

We estimate the capacity of the clustered setup using our logistic regression. Results of empirical observations for the cluster with 2.4GB RAM are shown in figure 10d. We observed that this data creates a model which depicts a linear growth. To study the impact of increasing cluster size on the capacity we conducted the same experiment but with much less amount of RAM to the RabbitMQ nodes, i.e. 0.4GB. Since λ_n is not linearly dependent on λ_c (because cluster size k is also a variable), we ran a multiple logistic regression with λ_c and λ_n as our independent variables and SLO as the dependent variable. Capacity in terms of number

RootNode/LeafNode	15-sec $\lambda_n = 10$	30-sec $\lambda_n = 5$	1-min $\lambda_n = 2.5$
1-sec ($\lambda_c = 25$)	56	58	58
5-sec ($\lambda_c = 5$)	118	133	133
15-sec ($\lambda_c = 1.67$)	272	284	285

(a) Leaf metadata node capacity

	15 -sec	30-sec	1-min
Capacity	28	32	55

(b) Root metadata node capacity

RootNode/LeafNode	15-sec	30-sec	1-min
1-sec	1568	1856	3190
5-sec	3304	4256	7315
15-sec	7616	9088	15675

(c) Capacity of a hierarchical configuration pattern

Table 1: Empirical capacity of federated monitoring configuration deployed as a tree of depth of two; monitoring node on an *m2.xlarge* instance type.

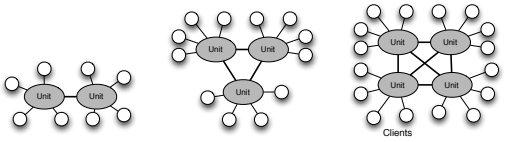
of clients which a node can handle reduces to the following form: $\lambda_c^* = 22.25/(0.01 + k \times 0.003)$. We plot the capacity for each k using this result and the capacity curve is shown in figure 10e. The figure depicts that the capacity of a clustered configuration starts to saturate as cluster size increases. Thus after some point in time it will not be useful to scale using clustered configuration.

Conclusion: Capacity of a clustered configuration starts to saturate as the size of the cluster increases. Also the model provides better estimates of SLO violation with more number of independent parameters, namely λ_c and λ_n.

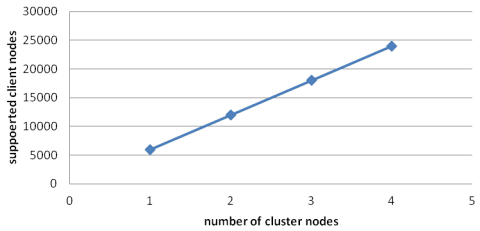
6.3 Dynamic Provisioning

To showcase the efficacy of our dynamic provisioning approach we conducted two experiments: First where we perform reactive provisioning, i.e. we trigger provisioning at SLO violations. Second, where we use a model and a forecaster that causes proactive provisioning based on forecasted workload. We have experimented with monitoring service deployed in a federated topology, more precisely, we started with a tree with one root metadata node and one leaf metadata node. We setup the leaf node with monitoring granularity of 1-sec, while the root node at 15-sec granularity. The capacity model of this configuration is already evaluated in Table 1a and 1b. We used this model, which reports a capacity of 56 clients for leaf node and 28 clients for root node. In both the cases, namely reactive and proactive, we conducted the experiment in following manner: i) We started with the two node tree with 10 clients attached to the leaf metadata node. The leaf metadata node was configured to monitor at 1-sec monitoring granularity, while the root metadata node at a 15-sec granularity. ii) We increased the workload in units of 10 clients (i.e. $n_c+ = 10$) after every 5-minutes.

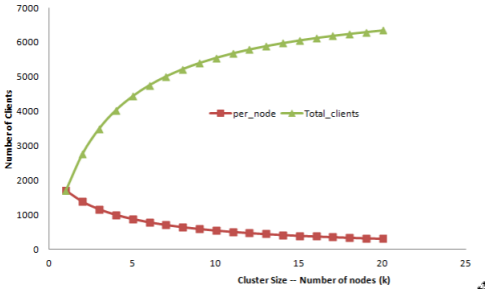
In the case of *reactive*-provisioning experiment, first reconfiguration process triggered when the n-c connections reached 50 nodes because the SLO got violated (shown in Figure 11). The new configuration at this point contains a second leaf node (n_2). We distribute the client workload equally between n_1 and n_2. As we gradually increase the workload each of the reconfigurations are triggered by SLO violation and leads to increase in capacity by one additional node. We used this SLO violation data from the reactive experiment as additional training data to re-learn the capacity model of leaf metadata nodes. Our new model reported a much conservative estimate of capacity, i.e. 33 client nodes for each leaf



(a) Two Node (b) Three Node (c) Four Node



(d) Configuration Capacity (RAM=2.4GB)



(e) Configuration Capacity (RAM=0.4GB)

Figure 10: Various cluster configurations and their empirically estimated capacity.

metadata node. We used this capacity model and a single step perfect forecaster (i.e. provides perfect forecast of workload at the next time instant) to evaluate the *proactive* provisioning approach. As mentioned, this approach triggers provisioning when the forecasted workload is more than the capacity of current configuration; thus in our experiment the provisioning happened at time instants when the number of clients reach 30, 60, 90 and 130; this is because when the workload reached 30-client node, the forecaster predicted a workload of 40-clients but the monitoring system's capacity was only 33-clients thus it provisioned an extra node in advance. We observe that proactive provisioning approach observes a slightly higher average data loss (i.e. 5.4%) at 90 clients than that observed by the reactive approach at 100 client workload. We believe that such randomness is because Ganglia uses UDP and the routers in EC2 would be observing higher workload at that instant, which could contribute to increase in data loss.

Conclusion: i) Our approach of dynamic adaptation is effective in both the situations, namely reactive *and* model-drive*. ii) Models for specific workloads can be simple but effective.*

7. RELATED WORK

Dynamic provisioning: There is a large body of related work in the area of dynamic capacity provisioning in data centers or compute clusters [15, 13, 19, 21, 14]. Much of this work is dynamic provisioning of the deployed web applications using analytic models, while our work considers dynamic provisioning of control plane services in multiple configurations, namely clustered and federated.

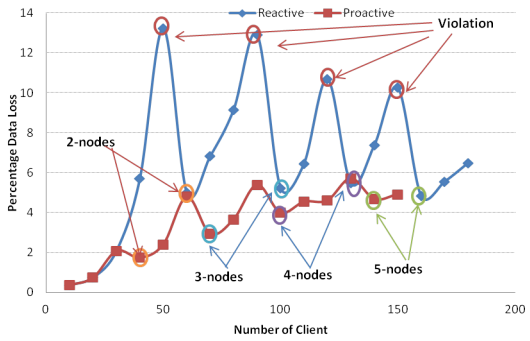


Figure 11: Dynamic provisioning of monitoring service

Generic system performance model based on ensemble of tree augmented bayesian networks has been developed by Zhang et al in [20] to capture the performance behavior of a system application under changing workload conditions. Watson et al. in [17] develop a probabilistic performance model for virtual machines with the objective of capturing the effect of statistical multiplexing in clouds and impact of other measurable factors to provide performance guarantees expressed in percentiles. In our work, we have used a logistic regression based approach to model a control plane service for performing dynamic provisioning.

Cloud Benchmarking Many researchers have conducted empirical evaluation of cloud platforms; Researchers in [5] benchmark Amazon EC2 to quantify CPU, disk and network performance of the provisioned virtual machines. Sharada et al. in [1] evaluate different virtualization technologies by running database workloads in a virtualized environment. Cooper et al. in [3] propose benchmark for the data storage subsystems popular in clouds, namely Hadoop, Cassandra, HBase and compare their bench marking results with shared MySQL implementation. Unlike much of the prior work, in this work we benchmark individual control plane services with different configurations.

8. CONCLUSION AND FUTURE WORK

In this paper we considered the problem of configuring and managing a private cloud and argued that the control plane of such cloud platforms must themselves be elastic to support dynamic control workloads. We presented a logistic regression model to automate the provisioning and dynamic reconfiguration of control plane services in a private cloud. We presented reactive and proactive methods for implementing the provisioning of elastic control plane services. We implemented our approach for two control plane services—monitoring and messaging—for OpenStack-based private clouds. Our experimental results on a laboratory private cloud testbed and using public cloud workloads demonstrated the ability of our approach to provision and adapt such services from very small to very large private cloud configurations.

Acknowledgements: We acknowledge the anonymous reviewers for their valuable suggestions. Upendra Sharma was supported in part by an IBM Graduate fellowship. This research was supported in part by an IBM OCR award and NSF grants CNS-1117221, OCI-1032765, CNS-0916972, CNS-0855128

9. REFERENCES

[1] S. Bose, P. Mishra, P. Sethuraman, and R. Taheri. In *Performance Evaluation and Benchmarking*, pages 167–182. Springer-Verlag, 2009.

[2] M. L. Buis. predict and adjust with logistic regression. *Stata Journal*, 7(2):221–226(6), 2007.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. SoCC '10, pages 143–154, New York, NY, USA, 2010.

[4] M. David, M. Richard, E. M. Errol, and J. Richard A. Hay. *Interrupted Time Series Analysis*. SAGE Publications, Inc., 0 edition, 1980.

[5] J. Dejun, G. Pierre, and C.-H. Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. ICSOC/ServiceWave'09, pages 197–207, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Dmtf - open virtualization format specification. `http://dmtf.org/sites/default/files/standards/documents/DSP0243_1.0.0.pdf`, 2 2009.

[7] D. W. Hosmer and S. Lemeshow. *Applied logistic regression (Wiley Series in probability and statistics)*. Wiley-Interscience Publication, 2 edition, 2000.

[8] Q. Jiang. Open Source IaaS Community Analysis. `http://www.qyjohn.net/?p=2233`.

[9] D. Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[10] Amazon Simple Storage Service. `http://www.amazon.com/s3`.

[11] OpenStack Cloud Software. `http://www.openstack.org`.

[12] J. Russell and R. Cohn. *Rabbitmq*. Book on Demand, 2012.

[13] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. NSDI'05, pages 71–84, Berkeley, CA, USA, 2005. USENIX Association.

[14] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proc. of the ACM SIGMETRICS Conf.*, Banff, Canada, June 2005.

[15] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM TAAS.*, 3:1:1–1:39, March 2008.

[16] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89, Nov. 2006.

[17] B. J. Watson, M. Marwah, D. Gmach, Y. Chen, M. Arlitt, and Z. Wang. Probabilistic performance modeling of virtualized resource allocation. ICAC '10, pages 99–108, New York, NY, USA, 2010. ACM.

[18] R. A. Yaffee. Forecast evaluation with stata. United kingdom stata users' group meetings 2010, Stata Users Group, 2010.

[19] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. ICAC '07, Washington, DC, USA, 2007.

[20] S. Zhang, I. Cohen, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. DSN'05, pages 644–653, Washington, DC, USA, 2005. IEEE Computer Society.

[21] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova. 1000 islands: Integrated capacity and workload management for the next generation data center. ICAC'08, pages 172–181, 2008.