A Pluggable Autoscaling Service for Open Cloud PaaS Systems

Chris Bunch Vaibhav Arora Navraj Chohan Chandra Krintz
Shashank Hegde Ankit Srivastava
Computer Science Department
University of California, Santa Barbara, CA
{cgb, vaibhavarora, nchohan, ckrintz, hegde, ankit} @ cs.ucsb.edu

I. Abstract

In this paper we present the design, implementation, and evaluation of a pluggable autoscaler within an open cloud platform-as-a-service (PaaS). We redefine high availability (HA) as the dynamic use of virtual machines to keep services available to users, making it a subset of elasticity (the dynamic use of virtual machines). This makes it possible to investigate autoscalers that simultaneously address HA and elasticity. We present and evaluate autoscalers within this pluggable system that are HA-aware and Quality-of-Service (QoS)-aware for web applications written in different programming languages. Hot spares can also be utilized to provide both HA and improve QoS to web users. Within the open source AppScale PaaS, hot spares can increase the amount of web traffic that the QoS-aware autoscaler serves to users by up to 32%.

As this autoscaling system operates at the PaaS layer, it is able to control virtual machines and be cost-aware when addressing HA and QoS. This cost awareness uses Spot Instances within Amazon EC2 to reduce the cost of machines acquired by 91%, in exchange for increased startup time. This pluggable autoscaling system facilitates the investigation of new autoscaling algorithms by others that can take advantage of metrics provided by different levels of the cloud stack.

II. Introduction

Cloud computing is a utility-oriented and service-based computing methodology that offers users many attractive features. Foremost, it simplifies the use of large-scale distributed systems through transparent and adaptive resource management. Infrastructure-as-a-Service (IaaS) vendors offer users access to virtual machines on-demand, on which users configure, install, and deploy software. Amazon Web Services, Microsoft Azure, and Google Compute Engine provide IaaS services that follow this model, offering virtual machines that run different operating systems.

Platform-as-a-Service (PaaS) clouds operate at a higher level of abstraction, offering users a fully managed software stack. Google App Engine is a popular public PaaS vendor that provides automated hosting of web applications written in the Python, Java, and Go programming languages. Google App Engine is able to scale web servers automatically by restricting the runtime stack. Hosted applications cannot read or write to a local filesystem, providing a stateless application server.

While cloud IaaS and PaaS systems have seen sizable increases in usage, they have also suffered from a number of outages, with some lasting several days [7] [8]. IaaS vendors recommend utilizing resources across multiple datacenters and autoscaling products to provide fault detection, recovery, and elasticity. Yet to make these offerings general-purpose, the metrics that these products can autoscale with are limited to VM-level metrics (e.g., CPU and memory usage).

As a motivating example, consider a typical application utilizing an IaaS and a LAMP stack. Once this application becomes popular, the system administrator needs to manually scale this application out, which requires them to become experts at scaling load balancers, application servers, and database nodes. By contrast, if the application itself runs at the PaaS layer, then the burden of autoscaling is removed from the developer and is placed onto the PaaS vendor.

We mitigate the problem of autoscaling by reinterpreting high availability (HA) under the veil of elasticity, and proposing a *pluggable* autoscaling service that operates within at the PaaS layer. Operating at the PaaS layer enables the autoscaling service to use high-level, application-specific metrics as well as low-level, cloud-specific metrics. Furthermore, because the autoscaling service operates at the PaaS layer, it can perform both inter-VM scaling and intra-VM scaling. Additionally, we elect to utilize the Google App Engine PaaS, so that our autoscaling service can operate on the one million active applications that currently run on Google App Engine [9].

This work targets the AppScale cloud system, but the techniques detailed here are extensible to other PaaS systems. AppScale, originally detailed in [6] and extended in [2][5], is an open source implementation of the Google App Engine APIs, enabling any application written for Google App Engine to execute over AppScale. AppScale is extensible to other application domains; in [3], it was extended to support high-performance computing (HPC) frameworks, including MPI and X10 [4]. AppScale runs over the public Amazon EC2 IaaS as well as the private Eucalyptus IaaS.

We begin by detailing the design of our pluggable autoscaling service and its implementation within the open source AppScale PaaS. We then evaluate autoscalers that implement support for HA, Quality-of-Service (QoS), and cost awareness, discuss related work, and conclude.

III. Design

This work redefines HA as the acquisition of virtual machines to keep services available to end-users, making it a special case of elasticity (the acquisition and release of virtual machines). We discuss how we use this idea within a cloud PaaS to provide HA via elasticity and our implementation of this idea in the open source AppScale PaaS. We then detail the pluggable autoscaling system that AppScale enables, along

Role Name	Description	Implemented Via
Load Balancer	Routes users to application servers.	`haproxy`
AppServer	Runs Google App Engine applications written in Python, Java, and Go.	Modified AppServer
Database	A persistent datastore for application data.	*Pluggable* [2]
AppCaching	A transient key-value cache for commonly accessed data.	`memcached`
Service Bus	A FIFO queue service.	VMWare RabbitMQ
Metadata	A datastore optimized for small items ($<$1KB).	Apache ZooKeeper
AppController	Starts and stops all other roles for its node.	Ruby daemon

TABLE I: A listing of the roles within the AppScale PaaS and the open source technologies that implement them.

with a number of autoscalers that can be used within AppScale to provide HA, Quality-of-Service (QoS), and cost awareness for hosted applications.

A. Role System

One goal of our work is to use elasticity to implement HA. To support this aim within a cloud PaaS, it is necessary to support HA for the full software stack that a cloud PaaS provides for its users. The approach that we take within the AppScale PaaS is what we call a *role system*, where each part of the software stack is designated a unique *role* that indicates what responsibilities it takes on, how it should be "started" (configured and deployed), and "stopped" (its tear-down process). The roles supported by the AppScale PaaS, their functionality, and the open source software packages that implement these roles are detailed in Table I.

Roles are started and stopped on each node by a Ruby daemon named the AppController. Users detail the "placement strategy" (a map indicating which nodes run each set of roles) for their AppScale deployment and pass this information to an AppController. The AppController then sends this information to all other AppControllers in the system and starts all the roles for its own node. Because the AppController itself is "role-aware", start and stop scripts can take advantage of this to enforce dependencies between roles.

B. Using Role Metadata to Support Pluggable Autoscaling

Storing metrics about every role within the AppScale PaaS enables any AppController to gather metrics about the global state of the AppScale deployment. As the AppController role is responsible for starting and stopping all other roles within its node, we extend it here to make it also responsible for maintaining all roles within its node. Furthermore, we make AppControllers responsible for maintaining HA within the AppScale PaaS as a whole. Specifically, after each AppController starts all the roles necessary for its own node, it creates a persistent connection with the Metadata service, so that it if that node ever fails, the other AppControllers will be notified of its failure.

This work extends the AppController to spawn a new thread, known as the *autoscaler*, that is responsible for making scaling decisions. It can view metrics about any role and any node via the Metadata service, and can use AppController functions to spawn and configure new nodes. Specifically, the types of metrics that are available to the autoscaler are:

- Application-level metrics: Information about hosted Google App Engine applications, including their API usage (e.g., datastore, caching, e-mail). The number of application servers serving each application is also available, as well as the average request latency.
- PaaS-level metrics: Information about the virtual machines hosting AppScale. This includes role-specific statistics (e.g., Load Balancer usage, Metadata usage) and historical data about previous scheduling decisions (e.g., the times/dates of previous node failures).
- IaaS-level metrics (if running on an open IaaS): Information about the physical machines that run the open IaaS. For Eucalyptus, this includes usage information on its Cloud Controller, Cluster Controller, Storage Controller, and Node Controller services.

IV. Framework Instantiations

The pluggable autoscaling system designed here can make scaling decisions based on application, PaaS, and IaaS-level statistics. We next detail how certain combinations of these metrics can be utilized to implement autoscaling algorithms to serve complementary use cases within the AppScale PaaS.

A. HA and QoS-Aware Autoscalers

One autoscaler supported within AppScale is HA. This autoscaler polls the Metadata service for a list of all the nodes that have registered itself as being alive, and looks for persistent connections named after each of those nodes. If any of those connections are missing (e.g., because a node has failed), then the autoscaler polls the Metadata service to see which roles that node was hosting and returns that information to the AppController's main thread. The AppController then spawns nodes to take the places of each failed node.

Another autoscaler that is supported within AppScale is QoS enforcement. This autoscaler service polls the Metadata service for data reported by the Load Balancer role about how many requests have been served in the last t seconds (a customizable value that defaults to 10 seconds) for each AppServer and how many are currently enqueued over the last t seconds. It then uses an exponential smoothing algorithm to forecast how many requests to expect and how many requests will be enqueued for the next t seconds. If either expectation exceeds a customizable threshold (defaulting to 5), then the autoscaler adds an AppServer within the system. If there is enough CPU and memory free on any node currently running, then the AppServer is added on a currently running node.

If there is not enough CPU and memory free on any currently running node, the autoscaler reports that a new node needs to be spawned to host an AppServer role. This autoscaler considers both intra-VM scaling (scaling within a node) and inter-VM scaling (scaling among nodes), in that order. Intra-VM scaling decisions are considered every minute, while inter-VM scaling decisions are considered every 15 minutes.

B. A Cost-aware Autoscaler

As AppScale operates at the PaaS layer, it is responsible for the acquisition and utilization of IaaS resources. Therefore, we have the opportunity to provide an autoscaler that can make decisions that are cost aware. For example, Amazon EC2 charges users on a per-hour basis. If the QoS-aware autoscaler described previously were to decide that resources it acquires are no longer needed, it would terminate them without realizing that keeping the resources until the end of the hour is free under the Amazon pricing model, and that there is no gain from terminating them before this hour price boundary.

We therefore augment the HA-aware, QoS-aware autoscaler used within AppScale to also be cost-aware in the following ways. Whenever a resource would normally be terminated by the QoS-aware autoscaler, it is instead relieved of all of its roles and the node becomes a hot spare, which can then be utilized by the HA-aware autoscaler to quickly respond to a node failure or by the QoS-aware autoscaler to quickly respond to increased web traffic. As we always run the HA-aware autoscaler before the QoS-aware autoscaler, the HA-aware autoscaler gets priority over these machines.

Amazon EC2's standard offering provides users with instances in an on-demand fashion. However, they do also offer an auction-style product, Spot Instances (SI), which users acquire by placing bids. If the bid that the user places is above the market price for a particular instance type, then the user gets the instance. If the market price ever rises above the user's bid, then the resource is reclaimed. As these instances can cost substantially less than the standard, on-demand instances, we propose a cost-aware autoscaler, which is able to automatically place bids and utilize SIs for both the HA autoscaler and the QoS autoscaler. To avoid losing instances to rising market prices, the cost-aware autoscaler searches through a history of successful bid prices and bids 20% above the average SI price paid. We evaluate the performance and cost impacts on the AppScale PaaS in Section V-C.

V. Experimental Evaluation

We next empirically evaluate our proposed autoscalers within AppScale. We begin by presenting our experimental methodology and then discuss our results.

A. Methodology

To evaluate the pluggable autoscaler system put forth by this work, we use sample Google App Engine applications provided by Google. We use implementations of the standard Guestbook application written in Python and Java. Upon each web request, the application queries the database for the most recent posts and displays them to the user. We host this application on AppScale with a single load balancer, application server, and database node.

B. Experimental Autoscaler Results

To evaluate the QoS autoscaler, we use the Apache Benchmark tool to dispatch 40,000 web requests to the Python and Java guestbook applications (70 concurrently), and measure how long it takes for AppScale to serve these requests. The average results of five runs of this experiment are shown in Figure 1. The first bar in each graph measures the time for AppScale to process the 40,000 web requests without the QoS autoscaler, as a baseline set of values.

The second bar in each graph uses the QoS autoscaler and only considers inter-VM scaling. It performs significantly better for the Python Guestbook application, but not for the Java Guestbook application. For both applications, the high request rate cause the QoS autoscaler to quickly acquire more nodes to run AppServer roles, which in turn allows more requests to be served at a time. However, the Java AppServer is faster due to the performance difference between the Java and Python languages, and the Java AppServer is able to use multithreading, while the Python AppServer is limited to one thread. Although the QoS autoscaler attempts to alleviate this problem by adding AppServers within a virtual machine, it is only able to do it up to a limit.

Paradoxically, the faster Java AppServer processes the 40,000 web requests before the newly spawned AppServers can have a significant impact (hence the similarities between Java QoS-off and Java QoS-on). To reduce the spawning time of these AppServers and increase their impact, we add a hot spare to the AppScale deployment before running Apache Benchmark. The results, shown in the third bar, detail a significant improvement for both the Python and Java Guestbook applications when a hot spare is used. The constant presence of a hot spare increases the cost to run the AppScale deployment, but is far less than the costs of business lost due to downtime.

Finally, the fourth bar utilizes the QoS autoscaler to only perform intra-VM scaling. It performs similarly to the scenario where the inter-VM scaler is utilized with a hot spare, and incurs a lower monetary cost (due to not needing the hot spare). Work is ongoing to consider the performance implications of utilizing the inter-VM and intra-VM scalers simultaneously.

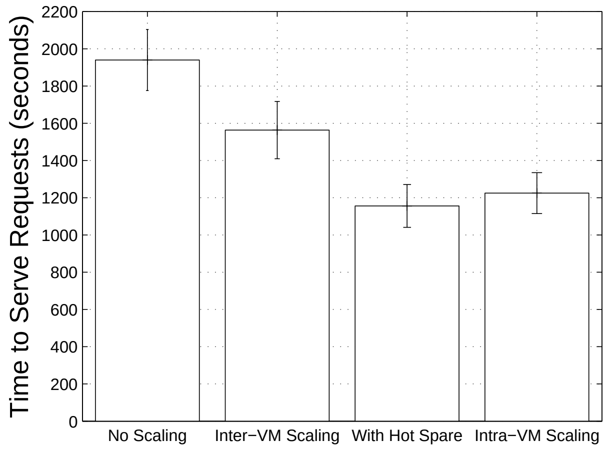
C. Experimental Metrics Results

We next move on to the gathering and reporting of metrics not traditionally considered by autoscaling algorithms, and their use in ongoing research into autoscalers used by the pluggable autoscaling solution proposed here. One metric that is simple yet powerful for a PaaS-layer autoscaler to measure is virtual machine startup time compared to cost incurred. Here, we use our cost-aware scheduler to acquire Amazon EC2's on-demand instances and Spot Instances (SIs) automatically, and report the time taken for the instances to boot up and the monetary cost incurred for one hour's use of these machines.

Table II shows two clear trends. First, on-demand instances can be acquired quickly, with low variance in both the time and cost incurred to utilize these machines. Second, the SIs take an order of magnitude longer to acquire, but cost an order of magnitude less. This makes SIs an ideal target to be used as hot spares within the AppScale PaaS, to be aggressively spawned and used to reduce the amount of time needed to recover from failures or ensure a higher QoS.

VI. Related Work

This work proposes and implements a pluggable autoscaling solution that can be utilized for fault tolerance as well as elasticity. The fields of fault tolerance and elasticity have been



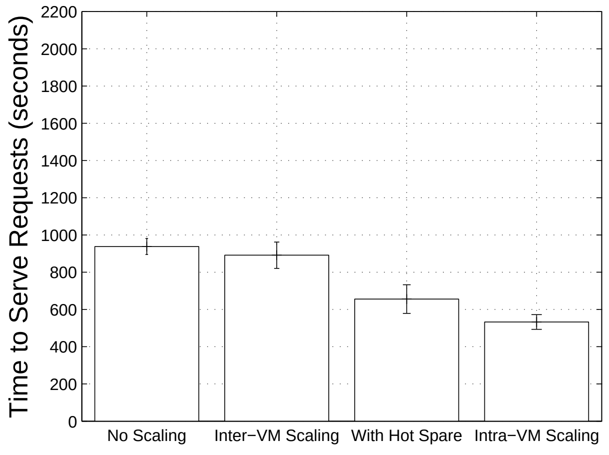


Fig. 1: Average time for AppScale to serve 40,000 web requests to the Python (Left) and Java (Right) Guestbook applications. We consider the case when the QoS autoscaler is off (the default before this work), when it is on (our contribution), when we proactively start a hot spare, and when we only scale within VMs. Each value represents the average of five runs.

Instance Type	Time to Acquire Instances (sec)	Monetary Cost ($)
On-Demand	37.03 ± 1.36	0.3200 ± 0.0000
Spot	411.31 ± 103.61	0.0299 ± 0.0002

TABLE II: Time and monetary cost incurred for the cost-aware scheduler to utilize Amazon EC2 on-demand and spot instances. These results reflect the average of ten runs, with the `m1.large` instance type in the AWS East Coast region.

well-studied, and a number of research efforts are conceptually similar to the work proposed here.

[1] and [10] focus on developing new elasticity techniques for web applications. Two main differences exist between these works and the pluggable autoscaling system proposed here. First, this work is the first that we know of to actually run within a cloud PaaS. [1] runs within the Amazon EC2 cloud IaaS, and similarly to this work, employs hot spares. While this work focuses on using hot spares to decrease the startup time for new nodes, [1] uses hot spares to mitigate potential SLA violations that can occur if their adaptive autoscaler puts too much workload on their own nodes. In contrast, [10] targets machines running over the Xen hypervisor, and does not use it as a research platform, opting instead to focus on elasticity algorithms customized for the three-tier web deployment strategy. Second, these efforts do not seek to provide a pluggable autoscaling solution for researchers to experiment with. They seek to provide novel autoscaling algorithms, and thus do not compete with the system proposed here. Work is ongoing to adapt the algorithms proposed in these works as autoscalers within the AppScale PaaS.

VII. Conclusion

We contribute an open source, pluggable autoscaler that runs at the cloud PaaS layer. By realizing HA as being part of maintaining an elastic cloud PaaS, we are able to provide an extensible autoscaling solution that adds both HA-awareness as well as QoS-awareness for web applications. We find that utilizing hot spares within our system can ensure a higher QoS to end users, with an increased performance of up to 32% for the applications tested.

We also contribute a cost-aware autoscaler that is able to automatically save users 91% for the instances utilized in the AppScale PaaS for the HA-aware or QoS-aware autoscalers, albeit with an increased time needed to respond to failures or low QoS. We contribute all of these autoscalers to the open source AppScale code base, found at `http://appscale.cs.ucsb.edu`.

References

[1] Bodik, P., Griffith, R., Sutton, C., Fox, A., Jordan, M. I., and Patterson, D. A. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds* (New York, NY, USA, 2009), ACDC '09, ACM, pp. 1–6.

[2] Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., Li, Y., and Nomura, Y. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing* (Jul. 2010).

[3] Bunch, C., Drawert, B., Chohan, N., Krintz, C., Petzold, L., and Shams, K. Language and runtime support for automatic configuration and deployment of scientific computing software over cloud fabrics. *Journal of Grid Computing 10* (2012), 23–46. 10.1007/s10723-012-9213-8.

[4] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. 40* (October 2005), 519–538.

[5] Chohan, N., Bunch, C., Krintz, C., and Nomura, Y. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing* (Jul. 2011).

[6] Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., and Wolski, R. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICST International Conference on Cloud Computing* (Oct. 2009).

[7] Final Thoughts on the Five-Day AWS Outage. http://www.eweek.com/c/a/Cloud-Computing/Final-Thoughts-on-the-FiveDay-AWS-Outage-236462/.

[8] Heroku: Widespread Application Outage. https://status.heroku.com/incident/151.

[9] Google I/O 2012 Keynote Transcription. http://oakleafblog.blogspot.com/2012/07/google-io-2012-day-2-keynote-by-urs.html.

[10] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., and Wood, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst. 3*, 1 (Mar. 2008).