AppScale Design and Implementation

Navraj Chohan Chris Bunch Sydney Pang
Chandra Krintz Nagy Mostafa Sunil Soman Rich Wolski
Computer Science Department
University of California, Santa Barbara

UCSB Technical Report Number 2009-02

January 27th, 2009

Abstract

We present the design and implementation of AppScale, an open source extension to the Google AppEngine (GAE) Platform-as-a-Service (PaaS) cloud technology. Our extensions build upon the GAE SDK to facilitate distributed execution of GAE applications over Xen-based clusters, including Infrastructure-as-a-Service (IaaS) cloud systems such as Amazon's AWS/EC2 and Eucalyptus. AppScale provides a framework with which researchers can investigate the interaction between PaaS and IaaS systems as well as the inner workings of, and new technologies for, PaaS cloud technologies using real GAE applications.

1 Introduction

Cloud Computing is a term coined for a recent trend toward service-oriented cluster computing. From a user's perspective, it is an attractive utility-computing paradigm based on Service-Level Agreements (SLAs), which is experiencing rapid uptake in the commercial sector. Following the lead of Amazon Web Services [2], a number of information technology vendors have since developed "utility," "cloud," or "elastic" product and/or service offerings [23, 1, 20, 11, 8]. Specific feature sets differ, but all cloud computing infrastructures share two common characteristics: they rely on operating system virtualization (e.g., Xen, VMWare, etc.) for functionality and/or performance isolation and they support per-user or per-application customization via a service interface (typically implemented using high-level language technologies, APIs, and web services).

This highly customizable, service-oriented methodology offers many important features. Foremost, it simplifies the use of large-scale distributed systems through transparent and adaptive resource management, and simplification and automation of configuration and deployment strategies for entire systems and applications. In addition, cloud computing enables arbitrary users to employ potentially vast numbers of multicore cluster resources that are not necessarily owned, managed, or controlled by the users themselves. By reducing the barrier to entry on the use of such distributed systems, cloud technologies encourage creativity and implementation of applications and systems by a broad and diverse developer base.

Extant cloud offerings, however, limit both innovation and scientific advance within the technologies of the cloud fabric itself because of their restricted nature (proprietary infrastructures and closed-source implementation for open interfaces). Although there exist a small number of open source cloud offerings [13, 9, 10, 6], these systems are difficult to use and extend or do not readily reflect the functionality of extant commercial settings. One exception is EUCALYPTUS [17], an open source implementation of the Amazon Web Services Elastic Computing Cloud (EC2). EUCALYPTUS enables users to deploy an Amazon EC2 cloud on their own Xen-enabled cluster resources with little effort. EUCALYPTUS provides transparency of the EC2 API, i.e., extent tools designed to work with EC2 (e.g., Rightscale [21], Elastra [7], etc.) cannot differentiate between an EC2 and a EUCALYPTUS installation.

In the spirit of EUCALYPTUS, we have designed and implemented an open source research framework, called *AppScale*, that implements the open interfaces of another important cloud offering, Google AppEngine (GAE) [11]. GAE makes available scalable abstractions via a set of interfaces that Google has implemented using its vast and highly scalable proprietary infrastructure. AppScale emulates the functionality of the GAE cloud. Specifically, AppScale implements the Google AppEngine open API and provides an infrastructure and toolset for distributed execution of GAE applications over Xen-enabled and IaaS systems (including EC2 and EUCALYPTUS). Moreover, by building on extant cloud and web-service technologies, AppScale is familiar, easy to use and extend (e.g. with new cloud services, databases, scheduling policies), and executes real, extant GAE applications using local (non-proprietary) cluster resources.

AppScale consists of multiple components that automate deployment, management, scaling and fault tolerance of the system and GAE applications. AppScale integrates, builds upon, and extends existing web service, high-level language, and cloud technology to provide a system that researchers and developers can employ to investigate new cloud technologies or the behavior and performance of extant applications. Moreover, AppScale deployment requires no modifications to GAE applications. AppScale is not meant to (and will not) compete with, outperform, or scale as well as, proprietary cloud systems, including GAE. Our intent is to provide a framework that enables researchers to investigate how such cloud systems operate and behave using real applications. Moreover, by facilitating application execution over important, lower-level cloud offerings such as EUCALYPTUS and EC2, AppScale also enables investigation of the interoperation and behavior of multiple cloud fabrics (PaaS and IaaS) in a single system.

In the sections that follow, we detail the background and motivation for this work and present the design and implementation of AppScale. We then evaluate an AppScale deployment under load for a number of different applications. We find that AppScale is easy to use, deploy, and extend, and supports execution of existing GAE applications efficiently. Moreover, its modular design facilitates a wide range of different deployment compositions and strategies that facilitate varying levels of fault tolerance and scalability, and

provides a framework that can be easily extended with new cloud technologies (e.g. database systems, scheduling policies, performance monitors, etc.).

2 Background and Motivation

Cloud computing [1, 2, 11, 24], as it is implemented in its commercially successful forms today, refers to a computing paradigm in which computational and storage capabilities are contracted for, through a well-defined programmatic interface that is exercised via a network connection. There are three broad classes or "styles" of cloud computing that are differentiated as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). The first describes systems in which high-level functionality (e.g., SalesForce.com [23], which provides customer relationship management software as an on-demand service) is hosted by the cloud and exported to thin clients via the network. The main feature of SaaS systems is that the API offered to the cloud client is for a complete software service and not programming abstractions or resources.

PaaS refers to the availability of scalable abstractions via an interface (e.g., Google's AppEngine [11] and Microsoft's Azure [4]) from which network accessible applications can be constructed. Developers of PaaS systems typically write their applications using scripting languages (e.g. Python) using an non-scalable SDK for debugging. To deploy an application, users manually upload program code and the data to which it is to be applied to into the PaaS cloud for execution. Thus PaaS clients are able to take advantage of high-quality, professionally maintained software infrastructure (such as Google's internal and proprietary implementation of BigTable [5]) through an interface that allows the cloud to manage resource usage. These systems restrict application functionality (e.g. no socket, HTTP(S)-only communication, execution limited to short responses to web requests, limited file system and database access) to protect system stability and to guarantee response times.

IaaS describes a facility for provisioning computational, storage, and network capacity under contract from a service provider. Typically, the main abstraction for this provisioning takes the form of a set of virtualized operating system stacks (e.g. Linux virtual machines) interconnected by a private layer 2 Ethernet that also connects to one or more storage abstractions (put/get storage, formattable file systems, etc.) Thus an IaaS allocation granted to a client or customer appears to be an isolated cluster "machine" that the client can fully configure and control. Amazon's AWS (Elastic Cloud Computing (EC2) and Simple Storage System (S3)) [2] is, at present, the most popular example of an IaaS-style computational cloud [17]. An EC2 request results in the provisioning of one or more Linux VMs (each of which has an externally routable IP address) to which the user is given root access via ssh. The VMs appear to be connected to a virtual private

Ethernet that supports limited layer 2 network functionality. Charging is done per VM occupancy hour. In addition, both storage options carry additional usage fees.

EUCALYPTUS is an open-source software service ensemble designed to support IaaS cloud computing research [17]. To support deployment in an academic environment, EUCALYPTUS includes an overlay mode that enables users to install it on one or more Xen-enabled clusters without modification to existing "baseline" operating system installation. In addition, the initial interface to EUCALYPTUS is compatible with Amazon's EC2 to the extent that commercial tools designed to work with EC2 (e.g., Rightscale [21], Elastra [7], etc.) cannot differentiate between an EC2 and a EUCALYPTUS installation. EUCALYPTUS allows researchers to deploy, on their own cluster resources, an open-source Web-service-based software infrastructure that presents a faithful reproduction of the Amazon EC2 functionality in its default configuration.

The goal of our work is to develop and make available a cloud technology with which researchers can investigate PaaS systems. Our system, called *AppScale*, is an open source implementation of Google's AppEngine (GAE) interface that executes over Xen-enabled and IaaS systems (including EC2 and EUCALYPTUS). AppScale thus facilitates investigation of PaaS-IaaS integration and interoperation and provides a framework with which we (and other researchers) can investigate tools, services, optimizations, and other techniques for modern and extant cloud technologies. By building on extant cloud and web-service technologies, AppScale is familiar and easy to use and extend, and executes existing GAE applications (providing researchers with real programs and distributed applications with which they can investigate and evaluate their contributions) using local, private resources. Further, AppScale provides a pathway for distributed GAE applications to execute with little developer effort in real IaaS systems (EC2 and EUCALYPTUS).

3 AppScale

AppScale is a multi-language, multi-component framework for executing GAE applications on virtualized cluster systems. Figure 1 overviews the AppScale design.

AppScale consists of a toolset (the AppScale Tools), three primary components, the AppServer (AS), the database management system, and the AppLoadBalancer (ALB), and an AppController (AC) for inter-component communication. AppServers are the execution engines for GAE applications which interact with a Database Master (DBM) via HTTPS for data storage and access. Database Slaves (DBSs) facilitate distributed, scalable, and fault tolerant data management. The AppController is responsible for setup, initialization, and tear down of AppScale instances, as well as cross component interaction. In addition, the AppController facilitates deployment of and authentication for GAE applications. The ALB serves as the head node of an AppScale deployment and initiates connections to GAE applications running in ASs. The

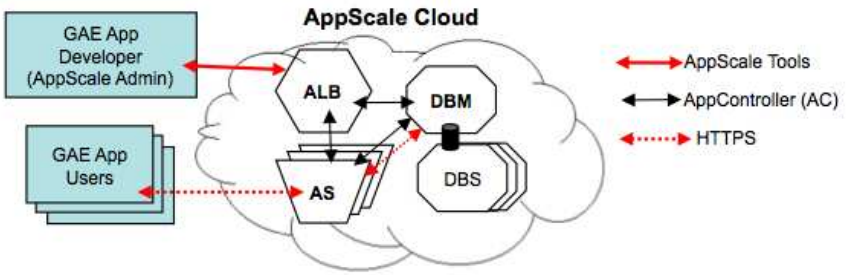

Figure 1: Overview of the AppScale design. The AppScale cloud consists of an AppLoadBalancer (ALB), a Database Master (DBM), one or more Database Slaves (DBS), and one or more AppServers (AS). Users of GAE applications interact with ASs; the developer deploys AppScale and her GAE applications through the head node (i.e. the node on which the ALB is located) using the AppScale Tools. AppControllers (AC) on each node interact with the other nodes in the system; ASs interact with the DBM of the system via HTTPS.

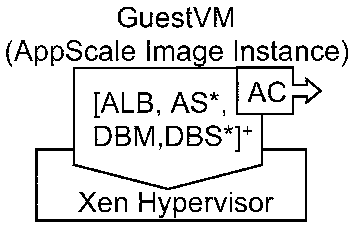

Figure 2: An AppScale Node. A node is an instance of a guestVM image, it implements an AC for cross-component interoperation and communication as well as at least one of the AppScale components: ALB, DBM, AS, or DBS; there can be multiple ASs and DBSs in a deployment. A single node implements one or more components. The guestVM executes over a Xen virtual machine monitor (VMM). The VMM can execute one or more guestVMs using the same hardware resources.

AC of the head node also monitors and manages the resource use and availability of the deployment. All communications across the system are encrypted via the secure socket layer (SSL).

A GAE application developer interacts with an AppScale instance (cloud) remotely using the AppScale Tools. Developers use these tools to deploy AppScale, to submit GAE applications to deployed AppScale instances, and to interact with and administer AppScale instances and deployed GAE applications. We distinguish developers from *users*; users are the clients/users of individual GAE applications.

An AppScale deployment consists of one or more virtualized operating system instances (guestVMs) as depicted in Figure 2. GuestVMs are Linux systems (*nodes*) that execute over the Xen virtual machine monitor or a Xen-based system such as Amazon's EC2 and Eucalyptus. For each AppScale deployment, there is a single AppLoadBalancer (ALB) which we consider the head node, one or more AppServers (AS), one Database Master (DBM) and one or more Database Slaves (DBSs). A node can implement any individual component as well as any combination of these components; the AppScale configuration can be specified by the developer via command line options of an AppScale tool.

We next detail the implementation of each of these components. To facilitate this implementation we

employ and extend a number of existing, successful, web service technologies and language frameworks.

3.1 AppController (AC)

The AppController (AC) is a SOAP client/server daemon written in Ruby. The AC executes on every node and starts automatically when the guestVM boots. The AC on the head node starts the ALB first and initiates deployment and boot of any other guestVM. This AC then contacts the ACs on the other guestVMs and spawns the components on each node. The head node AC first spawns the DBM (which then starts the DBSs) and then spawns the AppServers, configuring each with the IP of the DBM (for HTTPS database access by the GAE applications).

The AC on the head node also monitors the AppScale deployment for failed nodes and for opportunities to grow and shrink the AppScale deployment according to system demand and developer preferences. The AC periodically polls (currently every 10 seconds) the AC of every other node for a "heartbeat" and to collect per-application behavior and resource use (e.g. CPU and memory load). When a component fails, the AC restarts the component, respawning a node if necessary.

Although in this paper we evaluate the static default deployment of AppScale, we can also use this feedback mechanism to spawn and kill individual nodes of a deployment to respond to system load and performance. Killing nodes reduces resource consumption (and cost, if resources are being paid for) and consists of stopping the components within a node and destroying the guestVM. We can also spawn nodes to add more AppServers or Database Slaves to the system. We are currently investigating various scheduling policies, feedback mechanisms, and capability to interact with the underlying cloud fabric to modify service level agreements. AppScale currently supports starting and stopping of any component in a node and automatic spawning and destroying nodes.
The interface for the AC is shown in Table 1.

3.2 AppLoadBalancer (ALB)

The AppLoadBalancer is a Ruby on Rails [22] application that employs a simple HTTP server (nginx [16]) to select between three replicated Mongrel application servers [14] (for head-node load balancing). The ALB distributes initial requests from users to the AppServers (ASs) of GAE applications. Users initially contact the ALB to request a login to a GAE application. The ALB provides and/or authenticates this login and then selects an AS randomly. It then redirects the user request to the selected AS. The user, once redirected, continues to use the AppServer to which she was routed and does not interact further with the ALB unless she logs out or the AppServer she is using becomes unreachable.

API Method	Description (All methods take the secret as an argument)
AC to AC	
set_parameters	Sets the primary AppScale state upon AppScale deployment. Params: appscale_locations, database_credentials, app_name
status	Returns the CPU and memory load of the system
kill	Called by appscale-terminate-instances, backs up any state information needed and shutsdown the instance
done	Returns a boolean value corresponding to completion of a component startup
update	Called by appscale-upload-app, receives and deploys a new GAE application
change_component	Shuts down the executing component(s) and restarts others (as specified by the appscale_locations value)
AC to DBM	
upload_app	Commits a new GAE application to the database (encoded using base64), sets its metadata.
get/set_user_data	Given a username, gets/sets the user's metadata (administrator flag)
get/set_meta_data	Given an GAE application name, gets/sets the applications's metadata (size, locations, etc.), uploads an application for set
get/set_app_data	Given an GAE application name, gets/sets the applications's metadata (size, locations, etc.), create a user entry for set if none
does_user_exist	Given a username, returns a boolean value
does_app_exist	Given an GAE application file name, returns a boolean value
reset_password	Resets a users password in the metadata for a given user and password

Table 1: AppController API

3.3 AppServer (AS)

An AppServer is an extension to the Python program `dev_appserver.py` distributed freely as part of the Google AppEngine SDK for GAE application execution. `dev_appserver.py` enables GAE developers to test and develop their AppEngine applications (written in Python) on their localhost prior to uploading them to Google's internal proprietary infrastructure. The program uses a flat file for database access via protocol buffers [19]. The datastore that the application employs once uploaded to Google's infrastructure is BigTable, a highly scalable and reliable, distributed, non-relational, column-oriented datastore that leverages the scalable Google File System (GFS) [5].

Our extensions to `dev_appserver.py` enable execution of GAE applications on any Xen-enabled cluster to which the developer has access, including EC2 and Eucalyptus. AppServers can also be used without Xen but manual configuration is required. We extend `dev_appserver.py` with a generic datastore interface through which any database technology can be used. Currently we have implemented this interface to HBase and Hypertable, two open source implementations of Google's BigTable that execute over the distributed Hadoop File System (HDFS) [12].

We intercept the protocol buffer requests from the application and route them over HTTPS to/from the DBM front-end called the *PBServer*. The PBServer implements the interface to every datastore available and routes the requests to the appropriate datastore. The interaction is simple but fully supported by a

number of different error conditions, and includes:

- Put: add a new item into the table (create table if non-existant)
- Get: retrieve an item by ID number
- Query: SQL-like query
- Delete: delete an item by ID number

Our other extensions facilitate automatic invocation of ASs and authentication of GAE users. The AC of the node sets the location of the datastore (passed in from a request from the head node AC), upon AS start. The AS also stores and verifies the cookie secret that we use to authenticate users and direct the component to authenticate using the local AppController (AC).

An AS executes a single GAE application at time. To host multiple GAE applications, AppScale uses additional ASs (one or more per GAE application) that it isolates within their own AppScale nodes or that it co-locates within other nodes containing other AppScale components.

3.4 Data Management

In front of the Database Master (DBM) sits the PBServer. This Python program processes protocol buffers from a GAE application and makes requests on its behalf to read and write data to the datastore. As mentioned previously, AppScale currently supports HBase and Hypertable datastores. Both execute over HDFS within AppScale which performs replication, fault tolerance, and provides reliable service using distributed Database Slaves. The PBServer interfaces with HBase using Thrift and with Hypertable using SWIG [25] for cross-language interoperation.

The AC on the DBM's node provides access to the datastore via these interfaces to the other ACs and the ALB of an AppScale system. The ALB uses the datastore to store GAE applications that a developer has uploaded and the ACs use the datastore to authenticate the developer as well as the users of a GAE application.

3.5 AppScale Tools

The developer employs the AppScale tools to setup an AppScale instance and to deploy GAE applications over AppScale. The toolset consists of a small number of Ruby scripts that we named in the spirit of Amazon's EC2 tools for AWS. The tools facilitate AppScale deployment on Xen-based clusters as well as EC2 and Eucalyptus. The latter two systems require credentials and service-level agreements (SLAs) for the use, allocation (killing and spawning of instances) of resources on behalf of a developer; the EC2 tools (for either IaaS system) generate, manage, distribute (to deployed instances), and authenticate the credentials

Tool	Description/Parameters
appscale-run-instances	Deploys an AppScale instance, including optionally a GAE application -help: (Optional), takes no arguments, displays the usage of the tool -max: (Optional), specifies the maximum number of guestVMs to spawn -min: (Optional), specifies the minimum number of guestVMs to spawn -table: (Optional), the database type to use for this AppScale instance -file: (Optional), name of tar file containing the GAE application -ips: (Optional), name of yaml file containing the IPs to use Xen-only deployment
appscale-upload-app	Uploads a GAE application into a running AppScale instance -file: specifies name of tar file containing the GAE application
appscale-describe-instances	Queries the AC for CPU and Memory utilization on the AC and AS's. No parameters
appscale-reset-pwd	Resets the root user's (developer) password. Requests the root username. Requires that the secret be available on the local machine No parameters
appscale-terminate-instances	Cleans up and then destroys all instances created with appscale-run-instances; has no effect for Xen-only deployment No parameters

Table 2: AppScale Tools. For each tool, the second column describes its functionality and arguments

throughout the cluster. The AppScale tools sit above these commands and make use of them for credential management in IaaS settings.

Table 2 overviews the AppScale Tools. The `min` and `max` flags default to 1 and 4, respectively. Currently, `table` values can be *hbase* (default) or *htable*. `file` must name a valid gzipped tar file and has no default. If the file is not specified, the tool deploys the AppScale instance which the waits for the user to upload an GAE application using appscale-upload-app.

In a Xen-only setting, no credential management is necessary; the tools employ ssh keys for cluster management. In this setting, however, the developer must start the AppScale instance using Xen-tools manually and specifying the IP addresses of the instances to the appscale-run-instances tool. The `ips` flag is only valid for a Xen-only deployment and takes a yaml file that lists the IP addresses for each component of the system. For each IP address, there is a list of comma-separated component names (ALB, AS, DBM, or DBS). The number of entries in this file must match the `max` flag setting.

To employ AppScale over Eucalyptus or EC2, the developer (i) generates their X509 certification using Eucalyptus or EC2, (ii) downloads the ec2-tools from Amazon's AWS page and sets the appropriate environment variables, and (iii) executes appscale-run-instances. The tools enable developers to start an AppScale system, to deploy and tear down GAE applications, to query the state and performance of an AppScale deployment or application, and to manipulate the AppScale configuration and state. There is currently no limit on the number of applications that can be uploaded.

3.6 Tolerating Failures

There are multiple ways in which AppScale is fault tolerant. The AppController executes on all nodes. If the AC fails on a node with an AS, that AS can no longer authenticate users for a particular GAE application but authenticated users proceed unimpeded. Users that contact an ALB to re-authenticate (acquire a cookie) are redirected to a node with a functioning AS/AC to continue accessing the application. If the AC fails on the node with the ALB, no new users can reach any GAE applications deployed in the AppScale instance and the developer is not able to upload additional GAE applications; extant users however, are unaffected. This scenario (AC on the ALB node failure) is similar to AC failure on the DBM node. In this scenario (AC on the DBM node failure), ASs and users are unaffected.

The database system continues to function as long as at least one DBS is available. Similarly, the system is tolerant to failure of the PBServer (DBM front-end). If the PBServer fails on the DBM, the ASs will temporarily be unable to reach the database until the AC on the node restarts the PBServer. The ASs are not able to continue to execute (GAE applications will fail) if the DBM goes down or becomes unreachable. In this scenario, the ALB will restart the DBM component but unless the data from the original DBM is available to restore, the restart is similar to restarting AppScale.

Although, coupling multiple components per node reduces the number of nodes (resource requirements) and potentially better utilizes underlying resources, it also increases the likelihood of failure. For example, if all components are located in a single node, node failure equals system failure. If the node containing the ALB and DBM fails, the system fails. In these scenarios, component failure does not equal node failure however; the AC in the head node will attempt to restart components as described previously. The DBM issues 3 replicas of tables for DBSs to store, thus user data is available on failure of any individual DBS component. We are investigating the various failure scenarios and techniques for tolerating them within a deployed AppScale system as part of ongoing and future work.

3.7 AppScale Performance Monitoring

The dynamic monitoring system by the AC on the head node identifies unreachable components and nodes, and restarts them. In addition, this monitoring system collects the load (CPU and memory) and component behavior/activitiy on the system periodically (every 10s) to identify opportunities for shrinking and growing the AppScale deployment. In addition, AppScale implements a performance profiling system that enables researchers and developers to measure performance of the system and to identify bottlenecks and opportunites for optimization in an AppScale deployment or for an particular GAE application.

The AppScale profiling system consists of a set of extensions to the OProfile [18] and XenOProf [26],

lightweight (1-2% overhead) hardware performance monitor (HPM) sampling systems. We extended these open source systems to collect time-series data (as opposed to summaries) and independent collection for guestVMs and the administrator domain (dom0). We also extended the open source Ruby and Python runtimes to collect method-level information so that we are able to attribute performance data to all code regions throughout the AppScale distributed system (at the thread level). We rely on the UNIX tool `ntpdate` to synchronize the clocks across the cluster. We are currently working on tools to integrate, manipulate, and visualize profiles across the AppScale system to better understand and evaluate cloud application behavior.

3.8 AppScale Availability and Default Deployment

We distribute AppScale as a single Linux image and the AppScale Toolset. The image contains the code for the implementation of all of the components and a 64-bit Linux kernel and Ubuntu distribution. The packages included in this distribution include Ruby 1.8, Python 2.6, Rails 2.0.2, Sun Java 1.5.0, the Boost 1.36.0 libraries, NTP for time synchronization, Mongrel 1.1.5, nginx 0.6.32, god 0.7.11, Thrift, HBase 0.2.0, and Hypertable 0.9.0.12 and its required dependencies. The system is available from `http://appscale.cs.ucsb.edu/`; all new programs that we have contributed carry the Berkeley Software Distribution (BSD) License.

Figure 3 depicts our default deployment. By default, the AppScale system consists of four nodes (guestVMs). One node implements the ALB and the DBM, the other three nodes each implement an AS and a DBS. The default database system is HBase. We next experiment with and empirically evaluate this deployment.

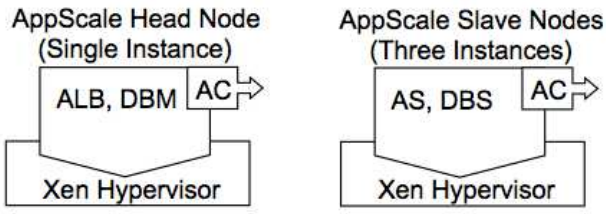

Figure 3: Default AppScale Deployment. A default deployment consists of one head node that implements the ALB and the DBM and three slave nodes that each implement an AS and a DBS. By default, the AppScale database implementation is HBase.

4 Evaluation

We next present the basic performance characteristics of AppScale default deployment. We note that we have not optimized AppScale in any way and that this study presents a baseline from which we will work to improve

Benchmark	Description	LOC Python/JavaScript	Transactions in Loop
cccwiki	user-defined webpage creation	289/10948	74
guestbook	presents last 10 signatures on a page; users can sign as well	81/0	9
shell	an interactive Python shell via a webpage	308/6100	14
tasks	to-do list manipulation	485/1248	44

Table 3: Benchmarks Statistics. For each benchmark, Column 2 is its description and Column 3 is its number of lines of code (Python/JavaScript). Column 4 is the number of transactions in the Grinder user loop that we use to load the system in our experiments.

the performance and scalability of the system over time. Our goal with AppScale to provide a research framework for the community, thus, we and others will likely identify ways to improve its performance over time. AppScale will never outperform or compete with the Google AppEngine proprietary implementation (nor is it meant to). We simply provide a framework with which to investigate existing open source GAE applications, services, and execution characteristics using local cluster resources.

4.1 Methodology

For our experimental methodology, we investigate four open source GAE applications made available as Google AppEngine Samples (`http://code.google.com/p/google-app-engine-samples/`). The applications are Python programs and Python/JavaScript programs. We overview them and their basic characteristics in Table 3. The cccwiki and tasks applications require the user to log in. Each application uses the AppScale datastore for all data manipulation. We record a user session that we replay for an increasing number of users repeatedly using the Grinder load testing framework (`http://grinder.sourceforge.net`) and its extensions [15].

For each experiment, we investigate two metrics, **(i) the total number of transactions completed over a five second interval**, and **(ii) the average response time for transactions that start during the interval**. Specifically, each *Grinder user* repeatedly executes a series of transactions (Table 3 Column 3). The *user* repeats this loop for 160 seconds. Grinder adds three users every five seconds to load the system.

For each five second interval in the 160 seconds of each test, we count the number of transactions that complete in that interval (for transactions completed per interval). For average response time, for each five second interval of the 160 seconds, we compute the average response time for the transactions that *started* in that interval. We repeat each experiment five times and compute the average and standard deviation for each interval across all of the runs.

Our cluster consists of quad-core 2.66GHz machines with 8GB RAM connected via gigabit Ethernet.

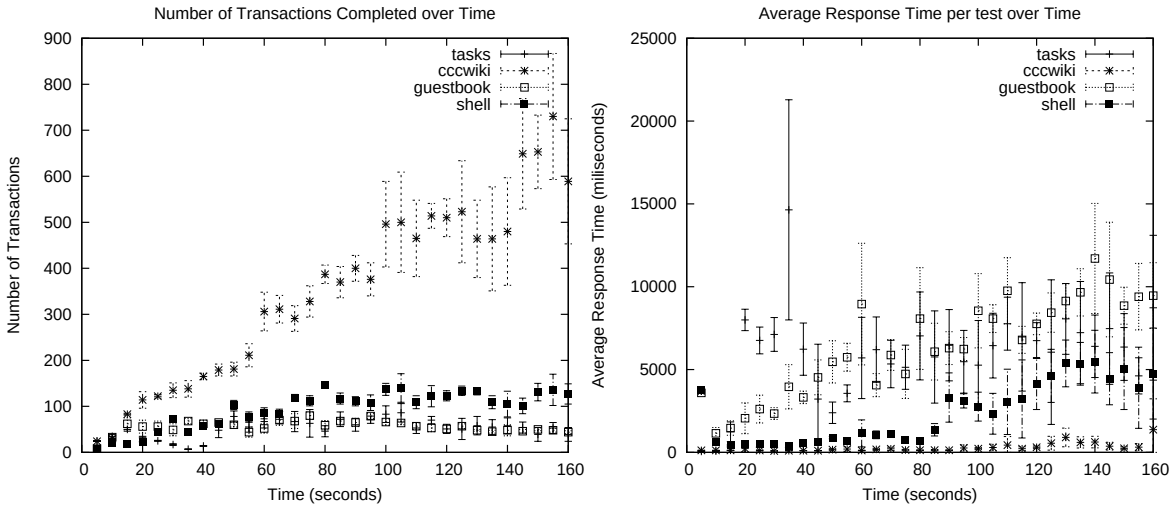

Figure 4: Application performance under stress: Transactions over time (left) and average response time (right). The x-axis is time in seconds; Grinder introduces three additional users for load every 5 seconds. In the left graph each point is the number of transactions that completed in that interval, on average across five runs (y-axis). In the right graph, each point is the average response time across the transactions that began in that interval, on average across five runs (y-axis).

We employ three of these machines for Grinder load generators. The machines are synchronized and each Grinder instance introduces a single user every five seconds. We specify the number of machines we use for the AppScale deployment with each experiment below.

4.2 Experimental Results

We first present data for each application, executed in isolation over AppScale, over time and increasing load. For this experiment, we employ the default AppScale configuration: one head node (ALB+DBM) and three slave nodes (AS+DBS each) with each node/guestVM on its own machine. Each of the three Grinder machines accesses the AS of one slave node.

Figure 4 shows the results. The left graph is transactions over time (higher is better), the right graph is average response time (lower is better). Each graph plots a point every five seconds. The x-axis is time and load: Grinder adds three additional users every 5 seconds. In the left graph each point, is the number of transactions that completed in that interval, on average across five runs (y-axis). In the right graph, each point is the average response time across the transactions that began in that interval, on average across five runs (y-axis).

All of the applications except guestbook tend to grow in the number of transactions as load increases. Guestbook's transaction count decreases after 100 seconds. This is because each guestbook posting increases the size of the database table. Our current (naive) implementation of database queries is to return the entire table to the node so that we can apply any filters at the GAE client side. As the database grows, each call is

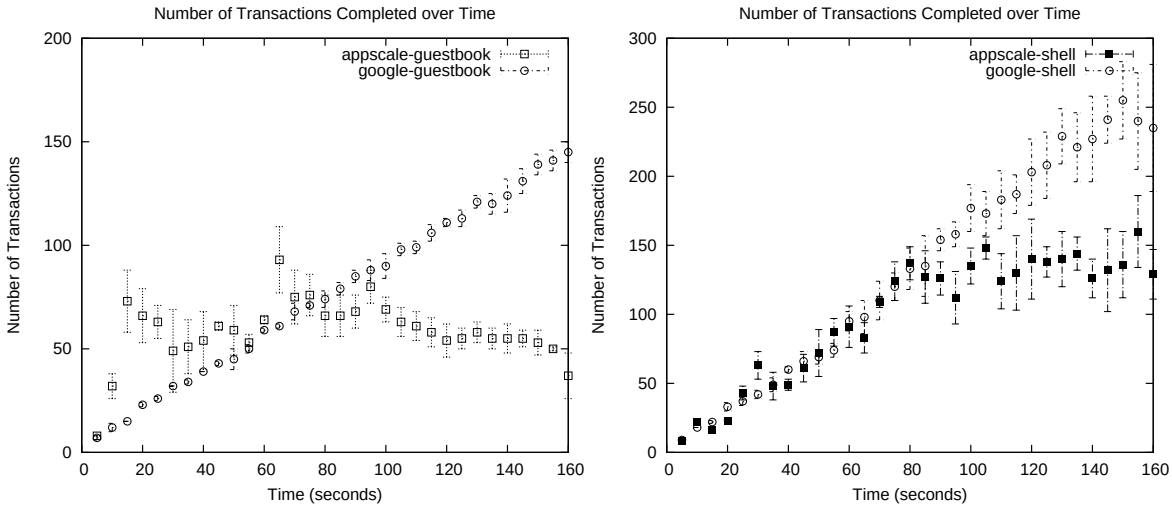

Figure 5: Transactions over time under increasing load (3 users per 5 seconds) for two applications – guestbook (left graph) and shell (right graph) – when hosted by Google and AppScale.

more expensive. We are currently extending our query process to return only the individual entries required, to address this issue. Cccwiki scales much better because each transaction only modifies an existing page, altering an entry in the table, as opposed to creating a new entry as guestbook does.

We also evaluated the difference between executing the four guestVMs on a single (quadcore) machine versus on individual machines. We find that we achieve very similar results for both for transactions completed and response time. This is interesting since it shows that the overhead of virtualization and co-location of virtual machines on these systems is not the performance bottleneck at this point. We find that in some cases the single machine case outperforms the distributed case due to network communication. This indicates that it may be beneficial to consider co-location of interoperating AppScale components for some behaviors and applications.

Finally, we investigate how AppScale performs relative to the Google proprietary infrastructure to better understand our baseline performance. We consider guestbook and shell applications since neither require the user to log in. We execute these applications using a Google AppEngine account. Figure 5 shows the results for transactions completed over time. AppScale transaction counts are more variable and do not scale for guestbook as load increases. Shell over AppScale scales up to a time/load of 80s. Google transaction counts scale perfectly. For response times (not shown) for guestbook Google consistently responds in 290-330ms regardless of load. For shell, Google's response time is more variable but still within a similar range. Shell performs more computation per request than guestbook. Google therefore starts to deny resources to the application at 150 seconds due to resource consumption limitations.

5 Related Work

The open source offering most similar to AppScale is AppDrop [3]. AppDrop is a simple Ruby-on-Rails application that emulates and hosts AppEngine applications on Amazon's EC2. AppDrop is a proof-of-concept that GAE applications can be executed in an environment other than that of Google.

There are multiple differences between AppScale and AppDrop. First, AppDrop (and any GAE applications that execute using it) is hosted entirely using a single guestVM image, which places significant limitations on IaaS usage/accounting, performance, scalability, and fault tolerance. The AppDrop progenitor uses his own EC2 account to host GAE applications on behalf of GAE developers. Thus, AppDrop is responsible for all EC2 charges and resource use as well as any "bad behavior" by the GAE applications. Each AppScale instance and its GAE applications is deployed and "owned" by each individual GAE developer.

AppDrop implements the flat file database integrated in Google's AppEngine test program (`dev_appserver.py`) for its datastore. This system is not distributed, scalable, or fault tolerant. AppDrop also employs a secondary database (implemented using Rails ActiveRecord and PostGreSQL) to store and retrieve the user's session data. AppScale uses the same distributed and fault tolerant database infrastructure as it does for its GAE applications and facilitates any database to be "plugged into" AppScale. AppScale currently integrates HBase and Hypertable as distributed, fault tolerant datastore options.

6 Conclusions

We present AppScale, an open source, PaaS cloud computing research framework that emulates the Google AppEngine-based cloud offering. AppScale is easy to use and to extend and automatically deploys itself and GAE applications over Xen-based cluster resources and IaaS clouds such as Amazon EC2 and Eucalyptus. AppScale implements a number of different components that facilitate deployment of GAE applications using local (non-proprietary resources). Moreover, AppScale provides a framework with which cloud researchers and application developers can investigate new techniques (services, tools, schedulers, optimizations), and the performance and behavior of these techniques, and for real (GAE) applications.

References

[1] 3Tera home page. `http://www.3tera.com/`.

[2] Amazon Web Services. `http://aws.amazon.com/`.

[3] AppDrop. `http://jchris.mfdz.com`.

[4] Microsoft Azure Service Platform. http://www.microsoft.com/azure/.

[5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Symposium on Operating System Design and Implementation*, 2006.

[6] J. Chase, D. Irwin, L. Grit, J. Moore, and S. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pages 90–100, 2003.

[7] Elastra Inc.. http://www.elastra.com.

[8] Engine Yard Inc. http://engineyard.com/.

[9] Enomalism Elastic Computing Infrastructure. http://www.enomaly.com.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 1997.

[11] Google AppEngine. http://code.google.com/appengine/.

[12] Hadoop. http://hadoop.apache.org/core/.

[13] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines. In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, November 2007.

[14] Mongrel. http://mongrel.rubyforge.org.

[15] P. Nagpurkar, W. Horn, U. Gopalakrishnan, N. Dubey, J. Jann, and P. Pattnaik. Workload characterization of selected jee-based web 2.0 applications. *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on Workload Characterization (IISWC)*, pages 109–118, Sept. 2008.

[16] Nginx. http://www.nginx.net.

[17] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus : A technical report on an elastic utility computing architecture linking your programs to useful systems. In *UCSB Technical Report ID: 2008-10*, 2008.

[18] Oprofile. http://oprofile.sourceforge.net.

[19] Protocol Buffers. Google's Data Interchange Format. http://code.google.com/p/protobuf.

[20] Rackspace Inc. http://www.rackspace.com/.

[21] Rightscale Inc. http://www.rightscale.com/.

[22] Ruby on Rails. http://www.rubyonrails.org.

[23] Salesforce Customer Relationships Management (CRM) System. http://www.salesforce.com/.

[24] Sun Microsystems Utility Computing. http://www.sun.com/service/sungrid/index.jsp.

[25] SWIG Software Development Tool. http://www.swig.org/index.php.

[26] Xenoprof. http://xenoprof.sourceforge.net.