Cloudstone: Multi-Platform, Multi-Language Benchmark and Measurement Tools for Web 2.0

Will Sobel, Shanti Subramanyam*, Akara Sucharitakul*, Jimmy Nguyen, Hubert Wong,
Sheetal Patil*, Armando Fox, David Patterson
*UC Berkeley and *Sun Microsystems*

Abstract

Web 2.0 applications place different demands on servers than their Web 1.0 counterparts: many-to-many user relationships, richer GUI's, and user-contributed content vs. unidirectional "publishing to the masses." Simultaneously, the definitive arrival of pay-as-you-go "cloud computing" and the proliferation of development stacks for software-as-a-service presents a different collection of degrees of freedom in deployment and tuning. To help explore this new space, we identify a number of non-obvious challenges and caveats to performing "apples-to-apples" comparisons of Web 2.0 application deployments. To help explore this space quantitatively, we offer Cloudstone, an open-source suite distributed as a set of virtual machine images comprising three implementations of a Web 2.0-representative application (Rails, PHP, Java EE) and a Markov-based distributed workload generator and data collection tools. To illustrate its usefulness we present preliminary measurements on Amazon Elastic Compute Cloud and Sun's Niagara 2 enterprise server, discussing the challenges of comparing platforms or software stacks and how Cloudstone can help quantify the differences.

1. Why We Need New Workloads

In the last five years, existing Web benchmarking tools (*ab, httperf,* SPECWeb) and applications (RuBiS, PetStore) have become less relevant to current practice in three ways. First, Web 2.0 application functionality has changed the characteristics of workloads that servers must handle. Second, the definitive arrival of "pay-as-you-go" cloud computing has brought fast growth and large scale within the reach of independent developers, making the corresponding concerns of benchmarking, stress testing and scalability much more broadly applicable. Lastly, debate continues over the actual performance differences between different development stacks, and we currently lack tools to investigate such questions systematically.

1.1. Web 2.0 Workloads are Different

Following Tim O'Reilly's widely-cited article [5] , we distinguish dominant architectures and features of "Web 1.0" applications (c.1995-2005) from those of "Web 2.0" applications (c.2005-present), noting their effect on application server workloads and deployment architectures.

One-to-many vs. many-to-many: the "mass customization" of Web 1.0 presents the same content heavily customized to each user, but since different users' activities and profiles rarely affect each other, a natural scaling strategy involves partitioning by user ID. In contrast, the social networking features of Web 2.0, in which each user's actions and preferences affect many other users in her network, suggest no obvious static partitioning as a scaling strategy.

User-contributed content: Whereas Web 1.0 focused on publishing to users, Web 2.0 users publish *to each other* via blogs, photostreams, tagging (Digg, Del.icio.us, etc.), collaborative filtering (e.g. Amazon book reviews and recommendations), etc. This changes the read/write ratio and write patterns compared to Web 1.0 applications.

Richer user experience: The quest for improved interactivity for Web applications has led to heavy use of technologies such as AJAX (Asynchronous JavaScript And XML), in which JavaScript code communicates with the server in the background (e.g. to enable form autocompletion or dynamic page UI) during what would otherwise be the Web user's "think time". On the one hand, these techniques generate extra work on the server, much of it speculative, in contrast to Web 1.0 applications in which the server dedicated essentially *no* resources to a given user during that user's think times. On the other hand, often the AJAX features of an application result in less rendering work on the server. A benchmark should help quantify these differences.

1.2. Cloud Computing is Different

True pay-as-you-go storage and compute services such as Amazon's EC2 and S3 changes the economics of service deployment in two important ways. One is the much lower cost of "instant" incremental scalability. Another is because capacity can be quickly un-deployed to save money, developers need not provision for very large peaks, nor waste money on idle capacity during nonpeaks. The net effect is that linear scaling and stress testing at high load, until recently the purview of heavily-capitalized corporations, are

now part of the operational landscape for independent developers as well. Even in "locked down" cloud computing environments such as EC2 where the developer has little control over network topology or hardware platform, understanding the performance bottlenecks imposed by the offered infrastructure is valuable.

1.3. Toward a Web 2.0 Workload, Application & Tools

The *Cloudstone* toolkit addresses these requirements with two components. The first is Sun's open source *Web2.0kit*[1], which consists of an example web application (a social events calendar) and a sophisticated application-specific distributed workload generator and data collector, Faban, that can scale to thousands of simulated users and supports fine-grained time-varying workloads. *Web2.0kit* includes three implementations of this application: PHP, Java EE and Rails. All three implementations provide the same Web 2.0 features (user-generated metadata, social networking functions, and a rich AJAX-based GUI) and adhere to each development stack's idioms. Each implementation exposes architectural as well as implementation-specific deployment and tuning choices, such as caching alternatives and database tuning parameters. Scripts to prepopulate the database are included as well, so the tools are ready to use "out of the box".

The second Cloudstone component is a set of automation tools to allow *Web2.0kit* to be used to run large experiments on cloud computing environments such as Amazon Elastic Compute Cloud (EC2).

Cloudstone consists entirely of 100% open source components connected in an architecture representative of datacenter-based deployment, and supplied as a set of virtual machine images for Amazon EC2 that readers are invited to download immediately (see instructions in section 0). In this paper we include some preliminary results of running the Cloudstone application on both a modern many-core enterprise server (Sun Niagara 2) and conventional datacenter hardware. It is our expectation that CloudStone can be used to systematically investigate questions such as: How do different development stacks trade off single-node performance for code complexity and programmer productivity? What is the relative performance difference between hosting a Web 2.0 application on a large number of modest-capacity servers vs. a smaller number of heavily-provisioned many-core servers? How do two different dynamic provisioning algorithms respond to workload peaks? What is the cost of deployment per user?

2. CloudStone Overview

2.1. Goals and Non-Goals

CloudStone's goal is to capture "typical" Web 2.0 functionality in a datacenter or cloud computing environment, provide a realistic workload generator, allow flexibility in deployment to mirror a range of typical best practices for caching and database tuning, and allow for testing and data collection of a variety of scenarios, including stress testing, linear scaling, "hockey stick" dynamic scaling, etc.

A *non-goal* of CloudStone is to argue for any one development stack over another. Many factors influence the choice of a development stack, and at best CloudStone will help developers quantify some of the effects of those choices.

Another *non-goal* is emulating legacy or Web 1.0 applications; our choice of application features and development stacks reflects popular design points for Web 2.0 application design today.

A further *non-goal* is to investigate the question of database sharding, partitioning, or scaling. Many projects are investigating how to improve the scalability of databases, and it would be far beyond the scope of this project to experiment with all of them. However, the application is written in such a way that "plugging" an alternative database architecture into it should be possible.

Lastly, it is a *non-goal* to provide the "best" (fastest, most memory-efficient, most elegant, etc.) implementation of this particular application in any particular stack. Instead we strive to code in a way that is generally representative of each platform's idioms and takes advantage of each platform's relative strengths as a competent developer on that platform would do.

2.2. Components and Typical Workflow

Cloudstone follows the now-canonical three-tier Web application architecture: a stateless Web server tier, a stateless or soft-state (caching or affinitized) application server tier, and a persistence tier. A typical experiment consists of:

1) Choose a deployment architecture and arrange for Cloudstone's included scripts to deploy the components

2) Prepare a workload profile for the workload generator

3) Run the experiment, deploying the workload generator to one or more machines distinct from those on which the application is deployed

4) Collect the resulting data

[1] `http://cooltools.sunsource.net/Web2.0kit`

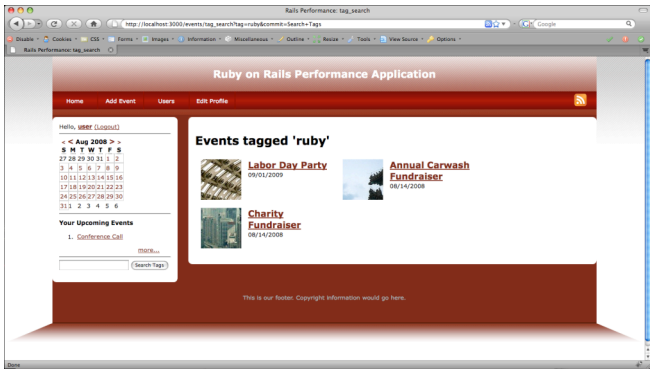

Figure 1. The *Web2.0kit* "social events" application's functionality and implementation are representative of Web 2.0 in all three implementations: Ruby on Rails, Java EE, and PHP.

Cloudstone provides various AMI's (virtual machine image files compatible with Amazon's Elastic Compute Cloud) that conveniently bundle the components to facilitate this workflow. We describe each of the components briefly.

2.3. Application

Figure 1 shows screenshots of the *Web2.0kit* social-events application. Users can browse events by date or tag, and see embedded maps to event locations; logged-in users can create events, tag events, attend an event, and add comments and ratings to an event. AJAX is used to make the UI streamlined and responsive; the same CSS stylesheets, XHTML markup, and RESTful [cite] URL's used to navigate the site are common to all three implementations, allowing the same workload generator and data collection tools to be used with any of them. From the user's point of view, all implementations behave identically. The data is stored in a relational database according to a simple snowflake schema; Cloudstone makes use of MySQL and includes scripts to populate the database with dummy data up to a desired size. The data is created in a deterministic manner that will always be the same for every run.

2.4. Workload Generation

Faban is a Markov-chain, closed-loop [cite Mor's paper], session-based [krishnamurthy] synthetic workload generator. (See [4] for an overview of approaches to Web load testing.) Unlike simpler workload generators such as *ab* or *httperf*, a Markov-chain based workload generator distinguishes N discrete application workflows, each consisting of a short sequence of related HTTP roundtrips to the server to accomplish some task ("add tag", "log in", etc.). A corresponding $N \times N$ matrix M gives the probability M_{ij} that workflow j will follow workflow i; this matrix can be derived from site-specific estimates [cite uRB] or by clustering information in web server logs [cite Menasce]. Many parallel Faban agents on different machines under the control of a central coordinator; Faban is designed to minimize coordinator-to-worker communication to avoid interfering with the network behavior of the test run. The number of simulated users can be changed up to twice a minute during the course of a run according to a text file specifying a workload profile. Faban does not use a standard "think-time" between requests, but opts for a constant spacing of all requests regardless of response time in the previous request.

2.5. Collecting and Analyzing Results

During a run, Faban records the response times of each request made by the load generator, from the time the request is issued by Faban until the time at which the last byte of the response is received. The request rate is measured between the initiations of successive operations. From these metrics, Faban calculates the mean, maximum, and 90th percentile of response times for each operation type. Faban also records utilization data by running external tools such as *iostat, mpstat, vmstat, netstat*, etc. periodically during a benchmark (the interval and set of additional tools to run is configurable) and graphs the results. All test data can be exported (e.g. to Comma-Separated Values) for further analysis.

2.6. Automation Support

Cloudstone includes Capistrano [cite] scripts to dynamically control the deploy process in our computing environment. The scripts provide *deploy, undeploy, restart* and *configure* actions for databases, web servers, application servers, and load balancers. The scripts are invoked by a central controller that passes them all necessary configuration and setup parameters. The scripts are currently setup for Amazon EC2, but can be easily modified for use in other environments as well via Capistrano's existing extension mechanisms.

3. The Challenges of Web 2.0 Benchmarking

Performing "fair" comparisons of different deployments is fraught with difficulty. A *stacks comparison* compares different implementations of the same functionality on different software stacks (e.g. Rails vs. PHP); a challenge is that the available tuning mechanisms are often quite different for each platform, being matched to each platform's development abstractions. A *platform comparison* compares the behavior of the same piece of software in different hardware environments, e.g. manycore vs. conventional datacenter server hardware; the radically different hardware topologies complicate this comparison. To help potential users of Cloudstone, we outline our basic tuning methodol-

ogy and point out caveats where the choice of platform or other tuning can trump other effects or otherwise distort results.

Multiple versions of the application will be made available with different tuning strategies in place. Currently the Rails application has two branches, one with caching and one without. Providing multiple tuning options allows developers to compare different strategies and determine how best to implement these strategies in their applications.

Traditional three-tier applications eventually bottleneck on the persistence tier, which is usually some kind of database. Hence there are generally three degrees of freedom involved in horizontal scaling:

(1) deploying additional web server, application server, etc. components and balancing the relative number and placement of these components to improve hardware utilization, until the database becomes the bottleneck;

(2) tuning the database to improve its performance;

(3) deploying caching to reduce the load on the database or application servers.

Our message in this paper is that Cloudstone as a framework is agnostic to these choices; we have made specific choices for our initial experiments to reflect what we understand to be contemporary practice, but the Cloudstone scripts can easily be modified to use alternatives.

3.1. Database Tuning

Database tuning is complex and we do not discuss it here, though we distinguish two general classes of optimization:

(1) Stack-independent techniques such as adding secondary indices, rewriting or combining queries, re-normalizing tables, modifying configuration parameters, and exploiting replication (master-slave, single writer/multiple readers, clustering, etc.). For example, we have found that MySQL is very sensitive to configuration: Every change in database size and hardware configuration requires a change to the configuration file to achieve optimal performance.

(2) Stack-specific techniques matched to each stack's database access model. For example, PHP requires the developer hand code all SQL queries, which allows more optimizations for experienced developers but increases the burden on less-experienced developers (who may write suboptimal queries). In contrast, Ruby on Rails and similar MVC frameworks provide object-relational mapping layers that insulate the developer from interacting directly with the database, making it less likely for inexperienced developers to do harm but also limiting the extent of query optimizations. Even within a framework, different versions may require changes to database query strategy; for example, the synthesis of queries that combine multiple tables changed significantly from Rails version 2.0 to 2.1.

Of course, the choice of database itself has performance implications. The two most popular choices for Web 2.0 deployments are MySQL and PostgreSQL; we use MySQL without loss of generality, but Cloudstone is agnostic to the type of database.

3.2. Deploying Additional Web & Application Servers

Despite the standardization of Web components such as Apache, different stacks often have different "preferred" deployment strategies. For example, since Rails processes run best in a dedicated application server rather than as a part of a full-featured web server, the preferred Rails deployment topology consists of an Apache or *lightppd* Web server acting as a reverse proxy load balancer, one or more single-threaded application server processes, a database, and optionally a caching server. We use the *Thin* application server[2], an optimized version of the popular *mongrel* server written in C, to efficiently dispatch requests to Rails. We did not investigate JVM-based deployment options using JRuby, but this could be easily handled by the Cloudstone tools.

In contrast, for PHP the most common deployment is a simple L4/L7 load balancer connected to several Apache web servers each integrating the *mod_php* plugin that executes PHP code. The operator sets the *maximum* allowed number of worker processes and lets Apache decide dynamically how many workers to keep running.

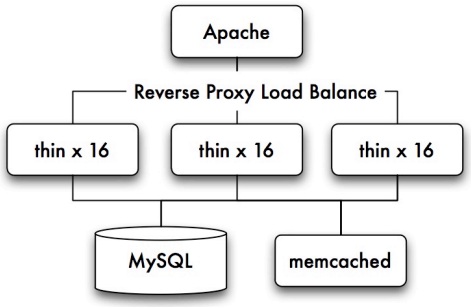

Figure 2. The preferred RoR deployment uses a single logical Apache process as a load balancer and separate Rails application servers (we use *thin*).

[2] `http://code.macournoyer.com/thin`

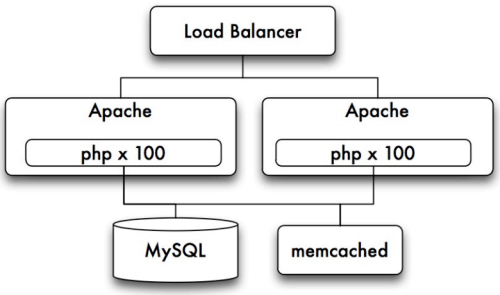

Figure 3. In contrast to Rails, the preferred PHP deployment uses Apache's built-in *mod_php* which can spawn large numbers of worker processes, rather than separate PHP application servers.

In either scenario, whenever multiple worker processes are deployed there is a need for a load balancer. The default *mod_proxy* load balancer built into Apache is fairly simplistic and does not allow for dynamic reconfiguration. More sophisticated alternatives include *haproxy, pound,* and Nginx, to name a few. We did not make use of an SSL accelerator, a component that offloads SSL certificate negotiation and encryption/decryption from the server, as our current operation matrix does not account for SSL operations. Most web 2.0 applications do not collect or store any personal information that requires encryption. For SaaS or retail application application SSL would be a requirement.

3.3. Caching

The easiest way to increase database performance is to avoid accessing it. This can be done by caching queries and web content and restructuring queries to reduce joins or round trips. Usually, the process of adding business-logic-specific caching cannot be automated since it requires the developer to specify cache policies on a per-page basis. Furthermore, the choice of software stack may dictate caching options.

The degrees of freedom for caching are generally:

(1) What is cached? Rails provides up to three levels of built-in caching. Caching full pages allows them to be served directly from a Web (asset) server, completely bypassing the application server and database, but is rarely effective for Web 2.0 applications due to the high degree of page customization. Caching rendered page fragments reduces the time associated with the rendering of that portion of the page, but additional techniques such as lazy loading of database results are required to take the database completely out of the loop in this case. Action caching offers a middle ground, reducing database access by serving the entire content of the page from cache while still allowing filters to be run to enforce authentication and other validations. Action and fragment caching are a natural fit for Rails' abstractions; in contrast, PHP does not provide built-in abstractions for caching, leaving it to each developer.

(2) Where are cached objects stored? The most popular choices for Web 2.0 stacks are in local RAM of each application server, in a file, or using *memcached*, a distributed RAM-based cooperative cache. *Memcached* has many deployment options for replication to provide redundancy and higher performance. It has no "native" object model so it can be used to store rendered content, query results, and user session data. Since *memcached* also has a concept of object lifecycle, it is well suited for storage of data whose validity is time-limited.

4. Example Measurements and Discussion

NOTE TO REVIEWERS: We expect to have many more benchmarking results by the workshop date and would be able to change the paper content and/or presentation accordingly. In addition to using Cloudstone for our research on datacenter automation, we will be using it for an undergraduate course on Web development, tuning and scaling.

As the previous section illustrates, the many deployment options and components make exact comparisons between frameworks difficult. In our example measurements, we attempt to keep as many deployment parameters the same as possible, including caching strategy, database, web server, and cache server.

We measured the following hardware platforms:

EC2: A single "extra-large compute instance" on EC2: a 64-bit, x86 architecture platform with 15 GB RAM, 4 virtual cores with 2 EC2 Compute Units each (Amazon describes 1 Compute Unit as "the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor"), and 1.7TB of local storage.

N2: A Sun Niagara 2 enterprise server configured with 32GB RAM, 8 UltraSPARC T2 cores at 1.4 GHz with 8 hardware threads per core, 50 GB of local storage.

We chose to answer a seemingly simple question: how many concurrent (simulated) users can be supported by a fixed amount of hardware under baseline conditions (no special database tuning, caching, etc.)? Our goal is not to promote one or the other platform but to illustrate Cloudstone's utility in addressing the issues that arise in cross-platform comparisons. We chose a service-level agreement (SLA) response time thresholds of 1 second, and we defined success as "90% of requests meeting their SLA's."[3]

[3] Large volume sites usually use 99% or even 99.9% as the SLA compliance threshold; we are working on adding instrumentation to Faban to collect these as well.

Figure 4 shows the result of deploying the Rails *Web2.0kit* application on EC2; Figure 5 shows the same application on N2. In both configurations, the single server hosts a MySQL database, an Apache load balancer, and between 1 and 32 *thin* processes. In both graphs, the height of each bar shows the number of concurrent users that can be "comfortably" served by a given number of server processes, where "comfortably" means 90% compliance withe SLA response time threshold. Note that Faban requires a minimum of 25 simulated users; as Figure 5 shows, on N2 we needed to run 6 copies of the *thin* server to accommodate 25 users.

Figure 4 suggests that an extra-large EC2 instance may be a better fit, but this conclusion may be premature. This baseline run does not enable caching, and with N2's greater RAM and some database tuning, we expect the gap to shrink. ***Note to reviewers:*** *We will have these results by the workshop date*.

By way of an initial comparison, earlier measurements [7] of the PHP implementation, using a deployment architecture similar to that shown in Figure 3, indicated that it could support up to 200 users on a comparable N2 server. However, that deployment made extensive use of caching. We expect to have measurements for both caching and non-caching versions of the Rails and PHP applications by the workshop deadline, which is the closest we believe we can come to an "apples to apples" framework comparison

5. Download & Acknowledgments

NOTE TO REVIEWERS: We are in the process of "packaging" EC2 AMI's allowing others to download all of this software, all of which will be Open Source.

Thanks to Peter Bodík for his help with the variable-workload modifications to Faban and the deployment scripts.

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Cisco Systems, Hewlett-Packard, IBM, Network Appliance, Oracle, Siemens AB, and VMWare, and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

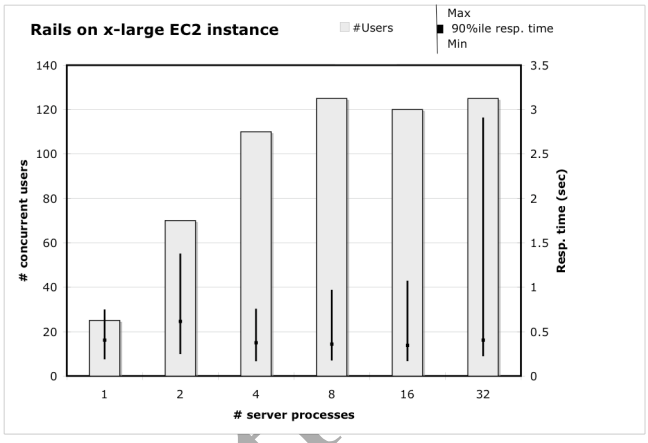

Figure 4. Number of users vs. number of server processes with all components shown in figure 2 on an EC2 "extra large" instance.

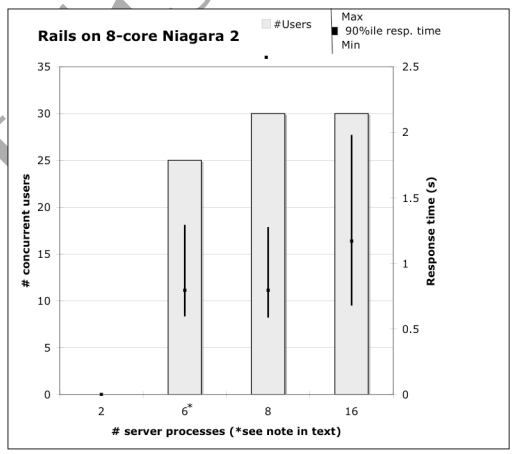

Figure 5. Same measurements on 8-core Niagara 2. ***NOTE:*** A minimum of 6 server processes was needed to serve 25 simulated users, the minimum allowed by Faban's load generator.

References

[1] George Candea et al., *Microreboot—A Technique for Cheap Recovery*. Proc. 6th OSDI, San Francisco, CA, Dec. 2004

[2] Roy T. Fielding and Richard N. Taylor, *Principled Design of the Modern Web Architecture.* ACM Trans. on Internet Technology 2(2): 115–150

[3] D. Krishnamurthy et al., *A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems,* IEEE Trans. on Software Eng. 32(11), Nov. 2006

[4] Daniel Menascé. *Load Testing of Web Sites.* IEEE Internet Computing 6(4), July/August 2002

[5] Tim O'Reilly. *What is Web 2.0?* http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html, Sept. 2005

[6] Bianca Schroeder, Adam Wierman, Mor Harchol-Balter. *Open vs. Closed: A Cautionary Tale.* Proc. NSDI 2006.

[7] Akara Sucharitakul and Shanti Subramanyam. *Cadillac or Nascar? A Non-Religious Investigation of Modern Web Technologies.* Proc.O'Reilly Velocity'08 Conference (http://en.oreilly.com/velocity2008)